Lecture Notes in Computer Science 2241

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Michael Jünger Denis Naddef (Eds.)

Computational Combinatorial Optimization

Optimal or Provably Near-Optimal Solutions

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Michael Jünger
Universität zu Köln, Institut für Informatik
Pohligstr. 1, 50969 Köln, Germany
E-mail:mjuenger@informatik.uni-koeln.de

Denis Naddef
ENSIMAG - antenne de Montbonnot
Laboratoire Informatique et Distribution, ZIRST
51, avenue Jean Kuntzmann, 38330 Montbonnot Saint Martin, France
E-mail: denis.naddef@imag.fr

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Computational combinatorial optimization : optimal or provably near optimal solutions / Michael Jünger ; Denis Naddef (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2001
(Lecture notes in computer science ; 2241)
ISBN 3-540-42877-1

CR Subject Classification (1998):G.1.6, G.2.1, G.2, F.2, J.1, E.1

ISSN 0302-9743
ISBN 3-540-42877-1 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2001
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN 10845868 06/3142 5 4 3 2 1 0

Preface

Increasing competition in the industrial and service sectors has led to a demand for optimal or provably near-optimal solutions to large scale optimization problems, and the exploration of a much larger range of alternatives in the race to improve productivity and efficiency. Many such decision problems involve the choice between a finite set of alternatives. Combinatorial optimization involves careful modeling of such problems, the mathematical analysis of the resulting discrete structures, the development of algorithms and their analysis, and the implementation of software to produce practical solutions.

DONET is a European research network with the participation of research groups in Belgium, England, France, Germany, Italy, Portugal, Switzerland, and The Netherlands, funded by the European Union in the program "Training and Mobility of Researchers (TMR)". The consortium's objectives include:

- consolidation of the network as a cohesive unit with a free flow of information and expertise between member institutions,
- elimination of the gaps between theoreticians and practitioners in discrete optimization,
- provision of a thorough training program for students, which includes strong theoretical and applied components, thus preparing them for entrance into both academic and industrial careers.

In this spirit we organized a

SPRING SCHOOL ON
COMPUTATIONAL COMBINATORIAL OPTIMIZATION
Schloß Dagstuhl, Germany, 15–19 May 2000

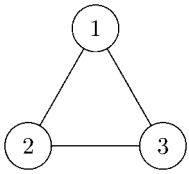
in which leading experts from and outside the DONET consortium gave lectures and organized exercises (often involving computer implementations), for half a day at a time, typically split into two units, per topic.

Schloß Dagstuhl provided an excellent atmosphere for teaching and learning. We are grateful that Egon Balas, Vašek Chvátal, Matthias Elf, Carsten Gutwenger, Claude Lemaréchal, Alexander Martin, Yves Pochet, Ted Ralphs, Giovanni Rinaldi, and Leslie Trotter followed our invitation to teach at the school and that we (or rather: they) were able to attract 57 students from 10 countries whose active participation made this school a productive and also a fun event.

We gladly accepted the invitation of Springer-Verlag to publish written versions (some with additional co-authors) of the courses taught at the school. During the school, the teachers agreed to write up their contributions within about a year. Just like the lectures, the written versions reflect the different individual styles of the teachers, and represent not a packing but rather a covering of the subjects taught at the school. We think that the resulting overlaps in the material presented are useful rather than annoying. After all, this is not a textbook

but a collection of lectures on computational combinatorial optimization. Therefore, we restricted our editing of the material provided to the maintenance of a common format and appropriate cross-referencing (indicated by [⟶ ...] in the text). The lectures appear in the order in which they were presented at the school. Even though they are independent, we recommend reading them in the given order.

The cover illustration shows a tetrahedron contained in a cube. For the triangle graph



there are four ways to partition the node set into two subsets. Below, we list these four partitions along with those edges that connect nodes in different subsets. Such edge sets are called "cuts" of the given graph. Next to the cuts, we list their characteristic vectors $(\chi_{12}, \chi_{13}, \chi_{23})$ where $\chi_{ij} = 1$ if the edge connecting nodes i and j is in the cut and $\chi_{ij} = 0$ otherwise.

node partition	cut edges	$(\chi_{12}, \chi_{13}, \chi_{23})$
$\{1,2,3\}$, $\emptyset$	$\emptyset$	$(0,0,0)$
$\{1,2\}$, $\{3\}$	$\{[1,3],[2,3]\}$	$(0,1,1)$
$\{1,3\}$, $\{2\}$	$\{[1,2],[2,3]\}$	$(1,0,1)$
$\{2,3\}$, $\{1\}$	$\{[1,2],[1,3]\}$	$(1,1,0)$

The four resulting 3-dimensional 0-1-vectors are the vertices of the shown tetrahedron, which is the "cut-polytope" for the triangle graph. It can be obtained from the unit cube by "chopping off" the four vertices that are not characteristic vectors of cuts. Polytopes and the "chopping off" process (which is called "separation") play an important rôle in this book. The cut polytope itself is also considered in some detail.

We would like to thank all authors for their productive cooperation, Constantin Hellweg and Ramin Sahamie for producing the cover illustration, and especially Matthias Elf for his help in the technical editing of this book.

September 2001

Michael Jünger and Denis Naddef
Organizers and Editors

Participants

Teachers

Egon Balas (Carnegie Mellon University)
Vašek Chvátal (Rutgers University)
Matthias Elf (University of Cologne)
Carsten Gutwenger (`caesar` Foundation Bonn)
Michael Jünger (University of Cologne)
Claude Lemarechal (INRIA Rhone-Alpes)
Alexander Martin (Technical University of Darmstadt)
Yves Pochet (University of Louvain)
Ted Ralphs (Lehigh University)
Giovanni Rinaldi (IASI-CNR Rome)
Leslie Trotter (Cornell University)

Students

Agostinho Agra (University of Lisboa)
Dino Ahr (University of Heidelberg)
Hernan Alperin (MIT)
Ernst Althaus (MPI Saarbrücken)
Enrico Angelelli (University of Brescia)
Miguel Anjos (University of Waterloo)
Claudia Archetti (University of Brescia)
Oliver Bastert (Technical University of Munich)
Pietro Belotti (Politecnico di Milano)
Andreas Bley (ZIB Berlin)
Ulrich Brenner (University of Bonn)
Christoph Buchheim (University of Cologne)
Gianfranco Ciaschetti (University of L'Aquila)
Paolo Detti (University of Rome III)
Andreas Fest (Technical University of Berlin)
Bernard Fortz (University of Brussels)
Jan Willem Goossens (Maastricht University)
Birgit Grohe (ZIB Berlin)
Leila Grouz Djerbi (France Telecom)
Gerald Gruber (University of Klagenfurt)
Mattias Grönkvist (Chalmers University of Technology)
Steffen Heber (DKFZ Heidelberg)
Stephan Held (University of Bonn)
Raymond Hemmecke (University of Duisburg)
Riko Jacob (Aarhus University)

Olaf Jahn (Technical University of Berlin)
Gunnar Klau (Technical University of Vienna)
Iris Klick (University of Bonn)
Joachim Kupke (University of Cologne)
Katharina Langkau (University of Bonn)
Frauke Liers (University of Cologne)
Andrea Lodi (University of Bologna)
Quentin Louveaux (University of Louvain)
Lucas Létocart (CNAM Paris)
Jean Maeght (University of Toulouse)
Fabrizio Marinelli (University of L'Aquila)
Isabel Martins (University of Lisboa)
Carlo Meloni (University of Rome III)
Michele Monaci (University of Bologna)
Susanne Moritz (Technical University of Darmstadt)
Ioannis Mourtos (London School of Economics)
Andreas Märkert (University of Duisburg)
Francisco Ortega (University of Louvain)
Marcus Oswald (University of Heidelberg)
Severine Petat (University of Paris VI)
Marc Pfetsch (Technical University of Berlin)
Diana Poensgen (ZIB Berlin)
Andrea Scozzari (University of Rome "La Sapienza")
Thomas Stidsen (Technical University of Denmark)
Nicolas Stier (MIT)
Chotiros Surapholchai (University of Heidelberg)
Tuomo Takkula (Chalmers University of Technology)
Dagmar Tenfelde-Podehl (University of Kaiserslautern)
Mathieu Van Vyve (University of Louvain)
Paolo Ventura (IASI-CNR Rome)
René Weiskircher (Technical University of Vienna)
Maria Zrikem (Electricité de France)

Table of Contents

General Mixed Integer Programming: Computational Issues for Branch-and-Cut Algorithms 1
Alexander Martin (Darmstadt University of Technology)

Projection and Lifting in Combinatorial Optimization 26
Egon Balas (Carnegie Mellon University)

Mathematical Programming Models and Formulations for Deterministic Production Planning Problems 57
Yves Pochet (Université Catholique de Louvain)

Lagrangian Relaxation 112
Claude Lemaréchal (INRIA Rhône-Alpes)

Branch-and-Cut Algorithms for Combinatorial Optimization and Their Implementation in ABACUS 157
Matthias Elf (University of Cologne), Carsten Gutwenger (caesar Foundation Bonn), Michael Jünger (University of Cologne), Giovanni Rinaldi (IASI-CNR Rome)

Branch, Cut, and Price: Sequential and Parallel 223
Laszlo Ladányi (IBM T.J.Watson Research Center), Ted K. Ralphs (Lehigh University), Leslie E. Trotter, Jr. (Cornell University)

TSP Cuts Which Do Not Conform to the Template Paradigm 261
David Applegate (AT&T Labs), Robert Bixby (Rice University), Vašek Chvátal (Rutgers University), William Cook (Princeton University)

Author Index 305

Author Index

Applegate, David 261

Balas, Egon 26
Bixby, Robert 261

Chvátal, Vašek 261
Cook, William 261

Elf, Matthias 157

Gutwenger, Carsten 157

Jünger, Michael 157

Ladányi, Laszlo 223
Lemaréchal, Claude 112

Martin, Alexander 1

Pochet, Yves 57

Ralphs, Ted K. 223
Rinaldi, Giovanni 157

Trotter Jr., Leslie E. 223

General Mixed Integer Programming: Computational Issues for Branch-and-Cut Algorithms

Alexander Martin

Darmstadt University of Technology, Department of Mathematics,
Schlossgartenstr. 7, D-64289 Darmstadt, Germany

Abstract. In this paper we survey the basic features of state-of-the-art branch-and-cut algorithms for the solution of general mixed integer programming problems. In particular we focus on preprocessing techniques, branch-and-bound issues and cutting plane generation.

1 Introduction

A general mixed integer program (MIP) is the problem of optimizing a linear objective function subject to a system of linear constraints, where some or all of the variables are required to be integer. The solution of general mixed integer programs is one of the challenging problems in discrete optimization. The problems that can be modeled as mixed integer programs arise, for instance, in science, technology, business, and environment, and their number is tremendous. It is therefore no wonder that many solution methods and codes exist for the solution of mixed integer programs, and not just a few of them are business oriented, see [64] for a survey on commercial linear and integer programming solvers.

One of the most successful methods to solve mixed integer programming problems are branch-and-cut algorithms. In Section 2 we outline the principle structure of a branch-and-cut algorithm. The main ingredients are preprocessing, the solution of the underlying linear programs, branch-and-bound issues, cut generation, and primal heuristics. Our aim is to give sort of a survey on the features that state-of-the-art branch-and-cut solvers for mixed integer programs include. Most of the issues presented are pretty much standard, but our intention is to use this paper more as a text book and to give the unfamiliar reader of this subject an impression on how mixed integer programs are solved today. In detail we focus on preprocessing in Section 3, branch-and-bound issues in Section 4, and cut generation in Section 5.

The software package that we use as a basic reference in this paper is `SIP`, which is currently developed at our institute and ZIB [49]. As mentioned most of the described issues are common to basically all state-of-the-art solvers and there are many other comparable codes that contain many of the described features. Among them are in particular `ABACUS`, developed at the University of Cologne

M. Jünger and D. Naddef (Eds.): Computat. Comb. Optimization, LNCS 2241, pp. 1–25, 2001.
© Springer-Verlag Berlin Heidelberg 2001

[66,40] [⟶ Elf/Gutwenger/Jünger/Rinaldi], `bc-opt`, developed at CORE [19], `CPLEX`, developed at ILOG [39], `MIPO`, developed at Columbia University [6], `MINTO`, developed at Georgia Institute of Technology [53], `SYMPHONY`, developed at Cornell University and Lehigh University [⟶ Ladányi/Ralphs/Trotter], and `XPRESS-MP`, developed at DASH [21].

It is common to use the library `Miplib` [10] as test set to evaluate certain features of a MIP code. `Miplib` is a collection of real-world mixed integer programming problems. From time to time we will refer to some instances from this library to explain certain phenomena. However, we will not give computational results here. The reader interested in concrete running times will find them, for instance, in [11,19,45,49].

Of course, branch-and-cut algorithms are not the only successful way to solve general mixed integer programs. For an excellent survey on alternative approaches, including test sets, Gomory's group approach and basis reduction, see [1].

2 Branch-and-Cut Algorithms

In this section we sketch the main ideas of a branch-and-cut algorithm. More details and references on this subject can be found in the annotated bibliography [14]. Suppose we want to solve a mixed integer program

$$\begin{aligned} &\min\ c^T x \\ &\text{s.t.}\quad Ax \le b, \end{aligned} \tag{1}$$

where $A \in \mathbb{Q}^{m\times n}$, $c \in \mathbb{Q}^n$, $b \in \mathbb{Q}^m$; the variables $x_i\,(i = 1,\dots,n)$ might be binary ($x_i \in \{0,1\}$), integer ($x_i \in \mathbb{Z}$), or continuous ($x_i \in \mathbb{R}$). Let $P_{\text{IP}} = \text{conv}\{x \in \mathbb{R}^n : x \text{ is feasible for (1)}\}$. The first step of the algorithm is to consider a relaxation of (1) by choosing a set $P' \subseteq \mathbb{R}^n$ with $P_{\text{IP}} \subseteq P'$ and to optimize the linear objective function over P'. For example, this relaxation might be the linear programming relaxation $\min\{c^T x : Ax \le b\}$ or a semidefinite relaxation. We only consider linear relaxations, hence, the set P' is always a polyhedron.

Let $\bar{x}$ be an optimal solution for the linear relaxation. If $\bar{x}$ is integer and all inequalities of $Ax \le b$ are satisfied by $\bar{x}$, we have found an optimal solution for (1). Otherwise, there exists a hyperplane $\{x \in \mathbb{R}^n : a^T x = \alpha\}$ such that $a^T\bar{x} > \alpha$ and $P_{\text{IP}} \subseteq \{x \in \mathbb{R}^n : a^T x \le \alpha\}$. Such a hyperplane is called a *cutting plane*. The problem of finding such a hyperplane is called the *separation problem*. More precisely,

> given $\bar{x} \in \mathbb{R}^n$. Decide, whether $\bar{x} \in P_{\text{IP}}$. If not, find some valid inequality $a^T x \le \alpha$ for P_{IP} such that $a^T\bar{x} > \alpha$.

It is well known that the separation problem for P_{IP} and the optimization problem $\min\{c^T x : x \in P_{\text{IP}}\}$ are polynomially equivalent, see [30,31]. Sometimes, the separation problem is restricted to a certain class of inequalities, in which

case we are searching for a violated inequality of that class. If we are able to find such a cutting plane, we can strengthen the relaxation and continue, for details see Section 5. This process is iterated until $\bar{x}$ is a feasible solution or no more violated inequalities are found. In the latter case this so-called *cutting plane phase* is embedded into an enumeration scheme. This is commonly done by picking some fractional variable $\bar{x}_i$ that must be binary or integer and creating two subproblems, one where one requires $x_i \geq \lceil \bar{x}_i \rceil$, and one where $x_i \leq \lfloor \bar{x}_i \rfloor$, see also the discussions in Section 4. The following algorithm summarizes the whole procedure.

Algorithm 1. (Branch-and-Cut Algorithm)
(1) Let L be a list of unsolved problems. Initialize L with (1).
(2) **Repeat**
(3) Choose a problem Π from L and delete it from L.
(4) **Repeat** *(iterate)*
(5) Solve the (linear) relaxation of Π. Let $\bar{x}$ be an optimal solution.
(6) If $\bar{x}$ is feasible for Π, Π is solved; **goto** (10).
(7) Look for violated inequalities and add them to the LP.
(8) **Until** there are no violated inequalities
(9) Split Π into subproblems and add them to L.
(10) **Until** $L = \emptyset$.
(11) Print the optimal solution.
(12) **STOP.**

The list L is usually organized as a binary tree, the so-called *branch-and-bound tree*. Each (sub)problem Π corresponds to a node in the tree, where the unsolved problems are the *leaves* of the tree and the node that corresponds to the entire problem (1) is the *root*. In the remainder of this section we discuss some issues that can be found in basically every state-of-the-art branch-and-cut implementation.

LP-Management. We assume that the reader is familiar with linear programming techniques. A comprehensive treatment of this subject can be found in [17, 59]. The method that is commonly used to solve the LPs within a branch-and-cut algorithm is the dual simplex algorithm, because an LP basis stays dual feasible when adding cutting planes. There are fast and robust linear programming solvers available, see, for instance, [39,21].

Nevertheless, one major aspect in the design of a branch-and-cut algorithm is to control the size of the linear programs. To this end, inequalities are often assigned an "age" (at the beginning the age is set to 0). Each time the inequality is not tight at the current LP solution, the age is increased by one. If the inequality gets too old, i. e., the age exceeds a certain limit, the inequality is eliminated from the LP. The value for this "age limit" varies from application to application.

Another issue of LP-management concerns the questions: When should an inequality be added to the LP? When is an inequality considered to be "violated"? And, how many and which inequalities should be added? The answers

to these questions again depend on the application. It is clear that one always makes sure that no redundant inequalities are added to the linear program.

A commonly used data structure in this context is the *pool*. Violated inequalities that are added to the LP are stored in this data structure. Also inequalities that are eliminated from the LP are restored in the pool. Reasons for the pool are to reconstruct the LPs when switching from one node in the branch-and-bound tree to another and to keep inequalities that were "expensive" to separate for an easier excess in the ongoing solution process.

Heuristics. Raising the lower bound using cutting planes is one important aspect in a branch-and-cut algorithm, finding good feasible solutions early to enable fathoming of branches of the search-tree is another. Primal heuristics strongly depend on the application. A very common way to find feasible solutions for general mixed integer programs is to "plunge" from time to time at some node of the branch-and-bound tree, i. e., to dive deeper into the tree and look for feasible solutions. This plunging is done by alternatingly rounding/fixing some variables and solving linear programs, until all variables are fixed, the LP is infeasible, a feasible solution has been found, or the LP value exceeds the current best solution. This rounding heuristic can be detached from the regular branch-and-bound enumeration phase or considered within the global enumeration phase. The complexity and the sensitivity to the change of the LP solutions influences the frequency in which the heuristics are called. Some more information hereto can be found, for instance, in [49,19,11].

Reduced Cost Fixing. The idea is to fix variables by exploiting the reduced costs of the current optimal LP solution. Let $\bar{z} = c^T \bar{x}$ be the objective function value of the current LP solution, z^{IP} be an upper bound on the value of the optimal solution, and $d = (d_i)_{i=1,\dots,n}$ the corresponding reduced cost vector. Consider a non-basic variable x_i of the current LP solution with finite lower and upper bounds l_i and u_i, and non-zero reduced cost d_i. Set $\delta = \frac{z^{\text{IP}} - \bar{z}}{|d_i|}$, rounded down in case x_j is a binary or an integer variable. Now, if x_i is currently at its lower bound l_i and $l_i + \delta < u_i$, the upper bound of x_i can be reduced to $l_i + \delta$. In case x_i is at its upper bound u_i and $u_i - \delta > l_i$, the lower bound of variable x_i can be increased to $u_i - \delta$. In case the new bounds l_i and u_i coincide, the variable can be fixed to its bounds and removed from the problem. This strengthening of the bounds is called *reduced cost fixing.* It was originally applied for binary variables [20], in which case the variable can always be fixed if the criterion applies. There are problems where by the reduced cost criterion many variables can be fixed, see, for instance, [24]. Sometimes, further variables can be fixed by logical implications, for example, if some binary variable x_i is fixed to one by the reduced cost criterion and it is contained in an SOS constraint (i. e., a constraint of the form $\sum_{j \in J} x_j \leq 1$ with non-negative variables x_j, $j \in J$), all other variables in this SOS constraint can be fixed to zero.

Enumeration Aspects. In our description of a branch-and-cut algorithm we left the questions open which problem to choose in Step (3) and how to split the problem in Step (9). We discuss these issues in detail in Section 4.

3 Preprocessing

Before entering the branch-and-cut phase as outlined in Section 2 there is usually a preprocessing step prefixed. Preprocessing aims at eliminating redundant information from the problem formulation given by the user and simultaneously tries to strengthen the formulation by logical implications. Preprocessing can be very effective and sometimes it might not be possible to solve certain problems without a good preprocessing. This includes, for instance, Steiner tree problems [43] or set partitioning problems [12]. Typically, preprocessing is applied only once at the beginning of the solution procedure, but sometimes it pays to run the preprocessing routine more often on different nodes in the branch-and-bound phase, see, for instance, [12,37,11]. There is always the question of the break even point between the running time for preprocessing and the savings in the solution time for the whole problem. There is no unified answer to this question. It depends on the individual problem, when intensive preprocessing pays and when not. In the following we discuss some preprocessing options and ways to implement them. Most of these options are incorporated in our code `SIP` and are drawn from [2,9,20,37,65].

We extend our definition of a mixed integer program in (1) slightly and consider it in the following more general form:

$$\begin{aligned} &\min\ c^T x \\ &\text{s.t.}\ \ Ax \left\{ \begin{matrix} \leq \\ = \\ \geq \end{matrix} \right\} b \\ &\qquad l \leq x \leq u \\ &\qquad x \in \mathbb{Z}^N \times \mathbb{R}^C, \end{aligned} \tag{2}$$

where M, N, and C are finite sets with N and C disjoint, $A \in \mathbb{R}^{M \times (N \cup C)}$, $c, l, u \in \mathbb{R}^{N \cup C}$, $b \in \mathbb{R}^M$. If some variable x_i, $i \in N$, is binary we have $l_i = 0$ and $u_i = 1$. If some variable x_j has no upper or lower bound, we assume that $l_j = -\infty$ or $u_j = +\infty$. Again we define $P_{\text{IP}} = \text{conv}\{x \in \mathbb{R}^n : x \text{ is feasible for (2)}\}$. Furthermore, we denote by $s_i \in \{\leq, =, \geq\}$ the sign of row i, i. e., (2) reads $\min\{c^T x : Ax\, s\, b,\ l \leq x \leq u,\ x \in \mathbb{Z}^N \times \mathbb{R}^C\}$. In order to avoid too many subcases in the following discussion we assume without loss of generality that there are no "greater than or equal" inequalities, i. e., $s_i \in \{\leq, =\}$. We consider the following cases:

Empty Rows. Suppose there is some row i with no non-zero entry. If

$$s_i = \left\{ \begin{matrix} \leq \\ = \end{matrix} \right\} \text{and} \left\{ \begin{matrix} b_i < 0 \\ |b_i| > 0 \end{matrix} \right\}$$

the problem is infeasible, otherwise row i can be removed.

Empty/Infeasible/Fixed Columns. For all columns j check the following: If

$$l_j > u_j,$$

the problem is infeasible. If

$$u_j = l_j,$$

fix column j to its lower (or upper) bound, update the right-hand side, and delete j from the problem.

Suppose some column j has no non-zero entry. If

$$\left\{ \begin{array}{l} l_j = -\infty \\ u_j = \quad \infty \end{array} \right\} \text{and} \left\{ \begin{array}{l} c_j > 0 \\ c_j < 0 \end{array} \right\}$$

the problem is unbounded (or infeasible in case no feasible solution exists). Otherwise, if

$$\left\{ \begin{array}{rl} l_j > & -\infty \\ u_j < & \infty \\ -l_j = u_j = & \infty \end{array} \right\} \text{and} \left\{ \begin{array}{l} c_j \geq 0 \\ c_j \leq 0 \\ c_j = 0 \end{array} \right\} \text{fix column } j \text{ to} \left\{ \begin{array}{l} l_j \\ u_j \\ 0 \end{array} \right\}.$$

Parallel Rows. Suppose we are given two rows $A_{i\cdot}x\, s_i\, b_i$ and $A_{j\cdot}x\, s_j\, b_j$. Row i and j are called parallel if there is some $\alpha \in \mathbb{R}$ such that $\alpha A_{i\cdot} = A_{j\cdot}$. The following situations might occur:

1. Conflicting constraints:
   a) $s_i =$ '$=$', $s_j =$ '$=$', and $\alpha b_i \neq b_j$
   b) $s_i =$ '$=$', $s_j =$ '$\leq$', and $\alpha b_i > b_j$
   c) $s_i =$ '$\leq$', $s_j =$ '$\leq$', and $\alpha b_i > b_j$ $(\alpha < 0)$

   In any of these cases the problem is infeasible.
2. Redundant constraints:
   a) $s_i =$ '$=$', $s_j =$ '$=$', and $\alpha b_i = b_j$
   b) $s_i =$ '$=$', $s_j =$ '$\leq$', and $\alpha b_i \leq b_j$
   c) $s_i =$ '$\leq$', $s_j =$ '$\leq$', and $\alpha b_i \leq b_j$ $(\alpha > 0)$
   d) $s_i =$ '$\leq$', $s_j =$ '$\leq$', and $\alpha b_i > b_j$ $(\alpha > 0)$

   In the first three cases row j is redundant, in (2d) row i.
3. Range constraints:
   a) $s_i =$ '$\leq$', $s_j =$ '$\leq$', and $\alpha b_i = b_j$ $(\alpha < 0)$
      The two inequalities can be aggregated into one equation.
   b) $s_i =$ '$\leq$', $s_j =$ '$\leq$', and $\alpha b_i < b_j$ $(\alpha < 0)$
      In this case both inequalities can be aggregated into one range constraint of the form $A_{i\cdot}x + u = b_i$ with $0 \leq u \leq b_i - \frac{b_j}{\alpha}$.

The question remains how to find parallel rows. Tomlin and Welsh [67] describe an efficient procedure, when the matrix A is stored columnwise, and Andersen and Andersen [2] slightly refine this approach. The idea is to use a hash function such that rows in different baskets are not parallel. Possible hash functions are the number of non-zeros, the index of the first and/or the last index of the row, the coefficient of the first non-zero entry, etc. In practice, the baskets are rather small so that rows inside one basket can be checked pairwise.

Duality Fixing. Suppose there is some column j with $c_j \geq 0$ that satisfies $a_{ij} \geq 0$ if $s_i =$ '$\leq$', and $a_{ij} = 0$ if $s_i =$ '=' for $i \in M$. If $l_j > -\infty$, we can fix column j to its lower bound. If $l_j = -\infty$ the problem is unbounded or infeasible. The same arguments apply to some column j with $c_j \leq 0$. Suppose $a_{ij} \leq 0$ if $s_i =$ '$\leq$', $a_{ij} = 0$ if $s_i =$ '=' for $i \in M$. If $u_j < \infty$, we can fix column j to its upper bound. If $u_j = \infty$ the problem is unbounded or infeasible.

Singleton Rows. If there is some row i that contains just one non-zero entry $a_{ij} \neq 0$, for some $j \in N \cup C$ say, then we can update the bound of column j in the following way. Initially let $-\bar{l}_j = \bar{u}_j = \infty$ and set

$$\bar{u}_j = b_i/a_{ij} \quad \text{if} \begin{cases} s_i = \text{'}\leq\text{'}, a_{ij} > 0 \text{ or} \\ s_i = \text{'}=\text{'} \end{cases}$$

$$\bar{l}_j = b_i/a_{ij} \quad \text{if} \begin{cases} s_i = \text{'}\leq\text{'}, a_{ij} < 0 \text{ or} \\ s_i = \text{'}=\text{'} \end{cases}$$

If $\bar{u}_j < \max\{l_j, \bar{l}_j\}$ or $\bar{l}_j > \min\{u_j, \bar{u}_j\}$ the problem is infeasible. Otherwise, we update the bounds by setting $l_j = \max\{l_j, \bar{l}_j\}$ and $u_j = \min\{u_j, \bar{u}_j\}$ and remove row i. In case variable x_j is integer (binary) we round down u_j to the next integer and l_j up to the next integer. If the new bounds coincide we can also delete column j after updating the right-hand side accordingly.

Singleton Columns. Suppose there is some column j with just one non-zero entry $a_{ij} \neq 0$, for some $i \in M$ say. Let x_j be a continuous variable with no upper and lower bounds. If $s_i =$ '$\leq$' we know after *duality fixing* has been applied that either $c_j \leq 0$ and $a_{ij} > 0$ or $c_j \geq 0$ and $a_{ij} < 0$. In both cases, there is an optimal solution satisfying row i with equality. Thus we can assume that $s_i =$ '='. Now, we delete column j and row i. After solving the reduced problem we assign to variable x_j the value

$$x_j = \frac{b_i - \sum_{k \neq j} a_{ik} x_k}{a_{ij}} \ .$$

Forcing and Dominated Rows. Here, we exploit the bounds on the variables to detect so-called forcing and dominated rows. Consider some row i and let

$$\begin{aligned} L_i &= \sum_{j \in P_i} a_{ij} l_j + \sum_{j \in N_i} a_{ij} u_j \\ U_i &= \sum_{j \in P_i} a_{ij} u_j + \sum_{j \in N_i} a_{ij} l_j \end{aligned} \tag{3}$$

where $P_i = \{j : a_{ij} > 0\}$ and $N_i = \{j : a_{ij} < 0\}$. Obviously, $L_i \leq \sum_{j=1}^{n} a_{ij} x_j \leq U_i$. The following cases might come up:

1. Infeasible row:
   a) $s_i = \text{`}=\text{'}$, and $L_i > b_i$ or $U_i < b_i$
   b) $s_i = \text{`}\leq\text{'}$, and $L_i > b_i$
   In these cases the problem is infeasible.
2. Forcing row:
   a) $s_i = \text{`}=\text{'}$, and $L_i = b_i$ or $U_i = b_i$
   b) $s_i = \text{`}\leq\text{'}$, and $L_i = b_i$
   Here, all variables in P_i can be fixed to its lower (upper) bound and all variables in N_i to its upper (lower) bound when $L_i = b_i$ ($U_i = b_i$). Row i can be deleted afterwards.
3. Redundant row:
   a) $s_i = \text{`}\leq\text{'}$, and $U_i < b_i$.

This row bound analysis can also be used to strengthen the lower and upper bounds of the variables. Compute for each variable x_j

$$\bar{u}_{ij} = \begin{cases} (b_i - L_i)/a_{ij} + l_j, & \text{if } a_{ij} > 0 \\ (b_i - U_i)/a_{ij} + l_j, & \text{if } a_{ij} < 0 \text{ and } s_i = \text{`}=\text{'} \\ (L_i - U_i)/a_{ij} + l_j, & \text{if } a_{ij} < 0 \text{ and } s_i = \text{`}\leq\text{'} \end{cases}$$

$$\bar{l}_{ij} = \begin{cases} (b_i - U_i)/a_{ij} + u_j, & \text{if } a_{ij} > 0 \text{ and } s_i = \text{`}=\text{'} \\ (L_i - U_i)/a_{ij} + u_j, & \text{if } a_{ij} > 0 \text{ and } s_i = \text{`}\leq\text{'} \\ (b_i - L_i)/a_{ij} + u_j, & \text{if } a_{ij} < 0. \end{cases}$$

Let $\bar{u}_j = \min_i \bar{u}_{ij}$ and $\bar{l}_j = \max_i \bar{l}_{ij}$. If $\bar{u}_j \leq u_j$ and $\bar{l}_j \geq l_j$, we speak of an *implied free variable*. The simplex method might benefit from not updating the bounds but treating variable x_j as a free variable (note, setting the bounds of j to $-\infty$ and $+\infty$ will not change the feasible region). Free variables will always be in the basis and are thus useful in finding a starting basis. For mixed integer programs however, it is better in general to update the bounds by setting $u_j = \min\{u_j, \bar{u}_j\}$ and $l_j = \max\{l_j, \bar{l}_j\}$, because the search region of the variable within an enumeration scheme is reduced. In case x_j is an integer (or binary) variable we round u_j down to the next integer and l_j up to the next integer. As an example consider the following inequality (taken from `mod015` from the `Miplib`):

$$-45x_6 - 45x_{30} - 79x_{54} - 53x_{78} - 53x_{102} - 670x_{126} \leq -443$$

Since all variables are binary, $L_i = -945$ and $U_i = 0$. For $j = 126$ we obtain $\bar{l}_{ij} = (-443 + 945)/-670 + 1 = 0.26$. After rounding up it follows that x_{126} must be one.

Note that with these new lower and upper bounds on the variables it might pay to recompute the row bounds L_i and U_i, which again might result in tighter bounds on the variables.

Coefficient Reduction. The row bounds in (3) can also be used to reduce coefficients of binary variables. Consider some row i with $s_i = \text{`}\leq\text{'}$ and let x_j be a binary variable with $a_{ij} \neq 0$.

$$\text{If} \begin{cases} a_{ij} < 0, U_i + a_{ij} < b_i, \text{ set } & a'_{ij} = b_i - U_i, \\ a_{ij} > 0, U_i - a_{ij} < b_i, \text{ set } & \begin{cases} a'_{ij} = U_i - b_i, \\ b_i = U_i - a_{ij}, \end{cases} \end{cases} \tag{4}$$

where a'_{ij} denotes the new reduced coefficient. Consider the following inequality of example `p0033` from the `Miplib`:

$$-230x_{10} - 200x_{16} - 400x_{17} \leq -5$$

All variables are binary, $U_i = 0$, and $L_i = -830$. We have $U_i + a_{i,10} = -230 < -5$ and we can reduce $a_{i,10}$ to $b_i - U_i = -5$. The same can be done for the other coefficients, and we obtain the inequality

$$-5x_{10} - 5x_{16} - 5x_{17} \leq -5$$

Note that the operation of reducing coefficients to the value of the right-hand side can also be applied to integer variables if all variables in this row have negative coefficients and lower bound zero. In addition, we may compute the greatest common divisor of the coefficients and divide all coefficients and the right-hand side by this value. In case all involved variables are integer (or binary) the right-hand side can be rounded down to the next integer. In our example, the greatest common divisor is 5, and dividing by that number we obtain the set covering inequality

$$-x_{10} - x_{16} - x_{17} \leq -1.$$

Aggregation. In mixed integer programs very often equations of the form

$$a_{ij}x_j + a_{ik}x_k = b_i$$

appear for some $i \in M$, $k, j \in N \cup C$. In this case, we may replace one of the variables, x_k say, by

$$\frac{b_i - a_{ij}x_j}{a_{ik}} \,. \tag{5}$$

In case x_k is binary or integer, the substitution is only possible, if the term (5) is guaranteed to be binary or integer as well. If this is true or x_k is a continuous variable, we aggregate the two variables. The new bounds of variable x_j are $l_j = \max\{l_j, (b_i - a_{ik}l_k)/a_{ij}\}$ and $u_j = \min\{u_j, (b_i - a_{ik}u_k)/a_{ij}\}$ if $a_{ik}/a_{ij} < 0$, and $l_j = \max\{l_j, (b_i - a_{ik}u_k)/a_{ij}\}$ and $u_j = \min\{u_j, (b_i - a_{ik}l_k)/a_{ij}\}$ if $a_{ik}/a_{ij} > 0$.
Of course, aggregation can also be applied to equations whose support is greater than two. However, this might cause additional fill in the matrix. Hence, aggregation is usually restricted to constraints and columns with small support.

Disaggregation. Disaggregation of columns is to our knowledge not an issue in preprocessing of mixed integer programs, since this usually blows up the

solution space. It is however applied in interior point algorithms for linear programs, because dense columns result in dense blocks in the Cholesky decomposition and are thus to be avoided [29].
On the other hand, disaggregation of rows is an important issue for mixed integer programs. Consider the following inequality (taken from the `Miplib`-problem `p0282`)

$$x_{85} + x_{90} + x_{95} + x_{100} + x_{217} + x_{222} + x_{227} + x_{232} - 8x_{246} \leq 0 \qquad (6)$$

where all variables involved are binary. The inequality says that whenever one of the variables x_i with $i \in S := \{85, 90, 95, 100, 217, 222, 227, 232\}$ is one, x_{246} must also be one. This fact can also be expressed by replacing (6) by the following eight inequalities:

$$x_i - x_{246} \leq 0 \qquad \text{for all } i \in S. \qquad (7)$$

Concerning the LP-relaxation, this formulation is tighter. Whenever any variable in S is one, x_{246} is forced to one as well, which is not guaranteed in the original formulation. On the other hand, one constraint is replaced by many (in our case 8) inequalities, which might blow up the constraint matrix. However within a cutting plane procedure this problem is not really an issue, because the inequalities in (7) can be generated on demand.

Probing. Probing is sometimes used in general mixed integer programming codes, see, for instance, [65]. The idea is to set some binary variable temporarily to zero or one and try to deduce further fixings from that. These implications can be expressed in inequalities as follows:

$$\begin{aligned} (x_j = 1 \Rightarrow x_i = \alpha) &\Rightarrow \begin{cases} x_i \geq l_i + (\alpha - l_i)x_j \\ x_i \leq u_i - (u_i - \alpha)x_j \end{cases} \\ (x_j = 0 \Rightarrow x_i = \alpha) &\Rightarrow \begin{cases} x_i \geq \alpha - (\alpha - l_i)x_j \\ x_i \leq \alpha + (u_i - \alpha)x_j \end{cases} \end{aligned} \qquad (8)$$

As an example, suppose we set in (6) variable x_{246} temporary to zero. This implies that $x_i = 0$ for all $i \in S$. Applying (8) we deduce the inequality

$$x_i \leq 0 + (1 - 0)x_{246} = x_{246}$$

for all $i \in S$ which is exactly (7).

In general, all these tests are iteratively applied until all of them fail. In other words, the original formulation is strengthened as far as possible. Our computational experiences [49] show that presolve reduces the problem sizes in terms of number of rows, columns, and non-zeros by around 10%. The time spent in presolve is neglectable (below one per mill). Interesting to note is also that for some problems presolve is indispensable for their solution. For example, problem `fixnet6` from the `Miplib` is an instance, where most solvers fail without preprocessing, but with presolve the instance turns out to be very easy.

4 Branch-and-Bound Strategies

In the general outline of a branch-and-cut algorithm, see Algorithm 1 in Section 2, there are two steps in the branch-and-bound part that leave some choices. In Step (3) of Algorithm 1 we have to select the next problem (node) from the list of unsolved problems to work on next, and in Step (9) we must decide on how to split the problem into subproblems. Popular strategies are to branch on a variable that is closest to 0.5 and to choose a node with the worst dual bound. In this section we briefly discuss some more alternatives. We will see that they sometimes outperform the mentioned standard strategy. For a comprehensive study of branch-and-bound strategies we refer to [44,45] and the references therein. We assume in this section that a general mixed integer program of the form (2) is given.

4.1 Node Selection

In the following we discuss three different strategies to select the node to be processed next, see Step (3) of Algorithm 1.

1. Best First Search (bfs).
   Here, a node is chosen with the worst dual bound, i. e., a node with lowest lower bound, since we are minimizing in (2). The goal is to improve the dual bound. However, if this fails early in the solution process, the branch-and-bound tree tends to grow considerably resulting in large memory requirements.
2. Depth First Search (dfs).
   This rule chooses the node that is "deepest" in the branch-and-bound tree, i. e., whose path to the root is longest. The advantages are that the tree tends to stay small, since always one of the two sons are processed next, if the node could not be fathomed. This fact also implies that the linear programs from one node to the next are very similar, usually the difference is just the change of one variable bound and thus the reoptimization goes fast. The main disadvantage is that the dual bound basically stays untouched during the solution process resulting in bad solution guarantees.
3. Best Projection.
   When selecting a node the most important question is, where are the good (optimal) solutions hidden in the branch-and-bound tree? In other words, is it possible to guess at some node whether it contains a better solution? Of course, this is not possible in general. But, there are some rules that evaluate the nodes according to the potential of having a better solution. One such rule is *best projection*. The earliest reference we found for this rule is a paper of Mitra [51] who gives the credit to J. Hirst. Let $z(p)$ be the dual bound of some node p, $z(\text{root})$ the dual bound of the root node, $\bar{z}_{\text{IP}}$ the value of the current best primal solution, and $s(p)$ the sum of the infeasibilities at node p, i. e., $s(p) = \sum_{i \in N} \min\{\bar{x}_i - \lfloor \bar{x}_i \rfloor, \lceil \bar{x}_i \rceil - \bar{x}_i\}$, where $\bar{x}$ is the optimal LP

solution of node p and N the set of all integer variables. Let

$$\varrho(p) = z(p) + \frac{\bar{z}_{\text{IP}} - z(\text{root})}{s(\text{root})} \cdot s(p). \tag{9}$$

The term $\frac{\bar{z}_{\text{IP}} - z(\text{root})}{s(\text{root})}$ can be viewed as a measure for the change in the objective function per unit decrease in infeasibility. The *best projection* rule selects the node that minimizes $\varrho(\cdot)$.

The computational tests in [49] show that *dfs* finds by far the maximal number of feasible solutions. This indicates that feasible solutions tend to lie deep in the branch-and-bound tree. In addition, the number of simplex iterations per LP is on average much smaller (around one half) for *dfs* than using *bfs* or *best projection*. This confirms our statement that reoptimizing a linear program is fast when just one variable bound is changed. However, *dfs* forgets to work on the dual bound. For many more difficult problems the dual bound is not improved resulting in very bad solution guarantees compared to the other two strategies. *Best projection* and *bfs* are doing better in this respect. There is no clear winner between the two, sometimes *best projection* outperforms *bfs*, but on average *bfs* is the best. Linderoth and Savelsbergh [45] compare further node selection strategies and come to a similar conclusion that there is no clear winner and that a sophisticated MIP solver should allow many different options for node selection.

4.2 Variable Selection

In this section we discuss rules on how to split a problem into subproblems, if it could not be fathomed in the branch-and-bound tree, see Step (9) of Algorithm 1. The only way to split a problem within an LP based branch-and-bound algorithm is to branch on linear inequalities in order to keep the property of having an LP relaxation at hand. The easiest and most common inequalities are *trivial inequalities*, i. e., inequalities that split the feasible interval of a singleton variable. To be more precise, if j is some variable with a fractional value $\bar{x}_j$ in the current optimal LP solution, we obtain two subproblems, one by adding the trivial inequality $x_j \leq \lfloor \bar{x}_j \rfloor$ (called the *left subproblem* or *left son*) and one by adding the trivial inequality $x_j \geq \lceil \bar{x}_j \rceil$ (called the *right subproblem* or *right son*). This rule of branching on trivial inequalities is also called *branching on variables*, because it actually does not require to add an inequality, but only to change the bounds of variable j. Branching on more complicated inequalities or even splitting the problem into more than two subproblems are rarely incorporated into general solvers, but turn out to be effective in special cases, see, for instance, [13,18,52]. In the following we present three variable selection rules.

1. Most Infeasibility.
   This rule chooses the variable that is closest to 0.5. The heuristic reason behind this choice is that this is a variable where the least tendency can be recognized to which "side" (up or down) the variable should be rounded. The hope is that a decision on this variable has the greatest impact on the LP relaxation.

2. Pseudo-costs.
This is a more sophisticated rule in the sense that it keeps a history of the success of the variables on which one has already branched. To introduce this rule, which goes back to [8], we need some notation. Let $\mathcal{P}$ denote the set of all problems (nodes) except the root node that have already been solved in the solution process. Initially, this set is empty. $\mathcal{P}^+$ denotes the set of all right sons, and $\mathcal{P}^-$ the set of all left sons, where $\mathcal{P} = \mathcal{P}^+ \cup \mathcal{P}^-$. For some problem $p \in \mathcal{P}$ let
   - $f(p)$ be the father of problem p.
   - $v(p)$ be the variable that has been branched on to obtain problem p from the father $f(p)$.
   - $x(p)$ be the optimal solution of the final linear program at node p.
   - $z(p)$ be the optimal objective function value of the final linear program at node p.

   The *up pseudo-cost* of variable $j \in N$ is

$$\Phi^+(j) = \frac{1}{|P_j^+|} \sum_{p \in P_j^+} \frac{z(p) - z(f(p))}{\lceil x_{v(p)}(f(p)) \rceil - x_{v(p)}(f(p))}, \tag{10}$$

   where $P_j^+ \subseteq \mathcal{P}^+$. The *down pseudo-cost* of variable $j \in N$ is

$$\Phi^-(j) = \frac{1}{|P_j^-|} \sum_{p \in P_j^-} \frac{z(p) - z(f(p))}{x_{v(p)}(f(p)) - \lfloor x_{v(p)}(f(p)) \rfloor}, \tag{11}$$

   where $P_j^- \subseteq \mathcal{P}^-$ and $\frac{z(p)-z(f(p))}{\lceil x_{v(p)}(f(p)) \rceil - x_{v(p)}(f(p))}$ and $\frac{z(p)-z(f(p))}{x_{v(p)}(f(p)) - \lfloor x_{v(p)}(f(p)) \rfloor}$, respectively, measure the change in the objective function per unit decrease of infeasibility of variable j. There are many suggestions made on how to choose the sets P_j^+ and P_j^-, for a survey see [45]. To name one possibility, following the suggestion of Eckstein [22] one could choose $P_j^+ := \{p \in \mathcal{P}^+ : v(p) = j\}$ and $P_j^- := \{p \in \mathcal{P}^- : v(p) = j\}$, if j has already been considered as a branching variable, otherwise set $P_j^+ := \mathcal{P}^+$ and $P_j^- := \mathcal{P}^-$. It remains to discuss how to weight the *up* and *down pseudo-costs* against each other to obtain the final *pseudo-costs* according to which the branching variable is selected. Here one typically sets

$$\Phi(j) = \alpha_j^+ \Phi^+(j) + \alpha_j^- \Phi^-(j), \tag{12}$$

   where α_j^+, α_j^- are positive scalars. A variable that maximizes (12) is chosen to be the next branching variable. As formula (12) shows, the rule takes the previously obtained success of the variables into account when deciding on the next branching variable. The weakness of this approach is that at the very beginning there is no information available, and $\Phi(\cdot)$ is almost identical for all variables. Thus, at the beginning where the branching decisions are usually the most critical the pseudo-costs take no effect. This drawback is tried to overcome in the following rule.

3. Strong Branching.
   The idea of *strong branching*, invented by CPLEX [38] (see also [3]), is before actually branching on some variable to test whether it indeed gives some progress. This testing is done by fixing the variable temporarily to its up and down value, i. e., to $\lceil \bar{x}_j \rceil$ and $\lfloor \bar{x}_j \rfloor$ if $\bar{x}_j$ is the fractional LP value of variable j, performing a certain fixed number of dual simplex iterations for each of the two settings, and measuring the progress in the objective function value. The testing is done, of course, not only for one variable but for a certain set of variables. Thus, the parameters of *strong branching* to be specified are the size of the candidate set, the maximum number of dual simplex iterations to be performed on each candidate variable, and a criterion according to which the candidate set is selected. Needless to say that each MIP solver has its own parameter settings, all are of heuristic nature and that their justification are based only on experimental results.

The computational experiences in [49] show that branching on a *most infeasible* variable is by far the worst, measured in CPU time, in solution quality as well as in the number of branch-and-bound nodes. Using *pseudo-costs* gives much better results. The power of *pseudo-costs* becomes in particular apparent if the number of solved branch-and-bound nodes is large. In this case the function $\Phi(\cdot)$ properly represents the variables that are qualified for branching. In addition, the time necessary to compute the *pseudo-costs* is basically for free. The statistics change when looking at *strong branching*. *Strong branching* is much more expensive than the other two strategies. This comes as no surprise, since in general the average number of dual simplex iterations per linear program is very small (for the `Miplib`, for instance, below 10 on average). Thus, the testing of a certain number of variables (even if it is small) in *strong branching* is relatively expensive. On the other hand, the number of branch-and-bound nodes is much smaller (around one half) compared to the *pseudo-costs* strategy. This decrease, however, does not completely compensate the higher running times for selecting the variables in general. Thus, *strong branching* is normally not used as a default strategy, but can be a good choice for some hard instances. A similar report is given in [45], where Linderoth and Savelsbergh conclude that there is no branching rule that clearly dominates the others, though pseudo-cost strategies are essential to solve many instances.

5 Cutting Planes

In this section we discuss cutting planes known from the literature that are incorporated in general MIP solvers. Cutting planes for integer programs may be classified with regard to the question whether their derivation requires knowledge about the structure of the underlying constraint matrix. In Section 5.1 we describe cutting planes that do not exploit any structure. An alternative approach to obtain cutting planes for a mixed integer program follows essentially the scheme to derive relaxations associated with certain substructures of the

underlying constraint matrix, and tries to find valid inequalities for these relaxations. Crowder, Johnson and Padberg [20] pioneered this methodology by interpreting each single row of the constraint matrix as a knapsack relaxation and strengthening the integer program by adding violated knapsack inequalities. This will be the topic of Section 5.2.

5.1 Cutting Planes Independent of Any Problem Structure

Examples of families of cutting planes that do not exploit the structure of the constraint matrix are mixed integer Gomory cuts [25,27,28,16,63], mixed integer rounding cuts [55], and lift-and-project cuts [5]. Marchand [46] describes the merits of applying mixed integer rounding cuts, see also [⟶ Pochet]. Lift-and-project cuts are investigated in [5,6] and are comprehensively discussed in the article by Egon Balas in this book [⟶ Balas].

In this section we concentrate on Gomory's mixed integer cuts. As a warm-up we start with the pure integer case. We will see that this approach (based on a rounding argument) fails if continuous variables are involved. In the general mixed integer case a disjunctive argument saves us.

Pure Integer Programs

Consider a pure integer program in the form $\min\{c^T x : Ax = b, x \in \mathbb{Z}^n_+\}$ with A, b integer. Set $P_{\text{IP}} = \text{conv}\{x \in \mathbb{Z}^n_+ : Ax = b\}$. Let $\bar{x}$ be an optimal solution of the LP relaxation $\min\{c^T x : x \in P\}$ with $P = \{x \in \mathbb{R}^n_+ : Ax = b\}$ and $B \subseteq \{1, \ldots, n\}$ be a basis of A with $\bar{x}_B = A_B^{-1}b - A_B^{-1}A_N x_N$ and $\bar{x}_N = 0$, where $N = \{1, \ldots, n\} \setminus B$.

If $\bar{x}$ is integer, we terminate with an optimal solution for $\min\{c^T x : x \in P_{\text{IP}}\}$. Otherwise, one of the values $\bar{x}_B$ must be fractional. Let $i \in B$ be some index with $\bar{x}_i \notin \mathbb{Z}$. Since every feasible integral solution $x \in P_{\text{IP}}$ satisfies $x_B = A_B^{-1}b - A_B^{-1}A_N x_N$,

$$A_{i\cdot}^{-1}b - \sum_{j \in N} A_{i\cdot}^{-1}A_{\cdot j}x_j \in \mathbb{Z}. \tag{13}$$

The term on the left remains integral when adding integer multiples of x_j, $j \in N$, or an integer to $A_{i\cdot}^{-1}b$. We obtain

$$f(A_{i\cdot}^{-1}b) - \sum_{j \in N} f(A_{i\cdot}^{-1}A_{\cdot j})x_j \in \mathbb{Z}, \tag{14}$$

where $f(\alpha) = \alpha - \lfloor \alpha \rfloor$, for $\alpha \in \mathbb{R}$. Since $0 \le f(\cdot) < 1$ and $x \ge 0$, we conclude that

$$f(A_{i\cdot}^{-1}b) - \sum_{j \in N} f(A_{i\cdot}^{-1}A_{\cdot j})x_j \le 0,$$

or equivalently,

$$\sum_{j \in N} f(A_{i\cdot}^{-1} A_{\cdot j}) x_j \geq f(A_{i\cdot}^{-1} b) \tag{15}$$

is valid for P_{IP}. Moreover, it is violated by the current linear programming solution $\bar{x}$, since $\bar{x}_N = 0$ and $f(A_{i\cdot}^{-1} b) = f(\bar{x}_i) > 0$. After subtracting $x_i + \sum_{j \in N} A_{i\cdot}^{-1} A_{\cdot j} x_j = A_{i\cdot}^{-1} b$ from (15) we obtain

$$x_i + \sum_{j \in N} \lfloor A_{i\cdot}^{-1} A_{\cdot j} \rfloor x_j \leq \lfloor A_{i\cdot}^{-1} b \rfloor, \tag{16}$$

which is, when the right-hand side is not rounded, a supporting hyperplane with integer left-hand side. Moreover, adding this inequality to the system $Ax = b$ preserves the property that all data are integral. Thus, the slack variable that is to be introduced for the new inequality can be required to be integer as well and the whole procedure can be iterated. In fact, Gomory [28] proves that with a particular choice of the generating row such cuts lead to a finite algorithm, i. e., after adding a finite number of inequalities, an integer optimal solution is found.

Later Chvátal [16,63] found a distinct but closely related way of finding a linear description of P_{IP}. He showed when using all supporting hyperplanes with integer left-hand side (an example of such an hyperplane is given in (16)) and rounding the right-hand sides yields again a polyhedron that contains P_{IP}. In addition, he proved that iterating this process a finite number of times provides P_{IP}.

Mixed Integer Programs

The two approaches discussed so far fail when both integer and continuous variables are present. Chvátal's approach fails because the right-hand side of a supporting hyperplane cannot be rounded down. Gomory's approach fails since it is no longer possible to add integer multiples to continuous variables to derive (14) from (13). For instance, $\frac{1}{3} + \frac{1}{3} x_1 - 2x_2 \in \mathbb{Z}$ with $x_1 \in \mathbb{Z}_+, x_2 \in \mathbb{R}_+$ has a larger solution set than $\frac{1}{3} + \frac{1}{3} x_1 \in \mathbb{Z}$. As a consequence, we cannot guarantee that the coefficients of the continuous variables are non-negative and therefore show the validity of (15). Nevertheless, it is possible to derive valid inequalities using the following *disjunctive argument.*

Property 1. Let $(a^k)^T x \leq \alpha^k$ be a valid inequality for a polyhedron P^k for $k = 1, 2$. Then,

$$\sum_{i=1}^{n} \min(a_i^1, a_i^2) x_i \leq \max(\alpha^1, \alpha^2)$$

is valid for both $P^1 \cup P^2$ and $\text{conv}(P^1 \cup P^2)$.

This property applied in different ways yields valid inequalities for the mixed integer case. We present Gomory's mixed integer cuts here, the other two, mixed integer rounding cuts and lift-and-project-cuts are both more or less also based on Property 1, see [⟶ Balas] and [⟶ Pochet].

Consider again the situation in (13), where $x_i, i \in B$, is required to be integer. We use the following abbreviations $\bar{a}_j = A_{i\cdot}^{-1} A_{\cdot j}$, $\bar{b} = A_{i\cdot}^{-1} b$, $f_j = f(\bar{a}_j)$, $f_0 = f(\bar{b})$, and $N^+ = \{j \in N : \bar{a}_j \geq 0\}$ and $N^- = N \setminus N^+$. Expression (13) is equivalent to $\sum_{j\in N} \bar{a}_j x_j = f_0 + k$ for some $k \in \mathbb{Z}$. We distinguish two cases, $\sum_{j\in N} \bar{a}_j x_j \geq f_0 \geq 0$ and $\sum_{j\in N} \bar{a}_j x_j \leq f_0 - 1 < 0$. In the first case,

$$\sum_{j\in N^+} \bar{a}_j x_j \geq f_0$$

must hold. In the second case, we have $\sum_{j\in N^-} \bar{a}_j x_j \leq f_0 - 1$, which is equivalent to

$$-\frac{f_0}{1-f_0} \sum_{j\in N^-} \bar{a}_j x_j \geq f_0.$$

Now we apply Property 1 to the disjunction $P^1 = P_{\text{IP}} \cap \{x : \sum_{j\in N} \bar{a}_j x_j \geq 0\}$ and $P^2 = P_{\text{IP}} \cap \{x : \sum_{j\in N} \bar{a}_j x_j \leq 0\}$ and obtain the valid inequality

$$\sum_{j\in N^+} \bar{a}_j x_j - \frac{f_0}{1-f_0} \sum_{j\in N^-} \bar{a}_j x_j \geq f_0. \tag{17}$$

This inequality may be strengthened in the following way. Observe that the derivation of (17) remains unaffected when adding integer multiples to integer variables. By doing this we may put each integer variable either in the set N^+ or N^-. If a variable is in N^+, the final coefficient in (17) is $\bar{a}_j$ and thus the best possible coefficient after adding integer multiples is $f_j = f(\bar{a}_j)$. In N^- the final coefficient in (17) is $-\frac{f_0}{1-f_0}\bar{a}_j$ and thus $\frac{f_0(1-f_j)}{1-f_0}$ is the best choice. Overall, we obtain the best possible coefficient by using $\min(f_j, \frac{f_0(1-f_j)}{1-f_0})$. This yields Gomory's mixed integer cut [26]

$$\begin{aligned} &\sum_{\substack{j:\, f_j \leq f_0 \\ j \text{ integer}}} f_j x_j + \sum_{\substack{j:\, f_j > f_0 \\ j \text{ integer}}} \frac{f_0(1-f_j)}{1-f_0} x_j + \\ &\sum_{\substack{j\in N^+ \\ j \text{ non-integer}}} \bar{a}_j x_j - \sum_{\substack{j\in N^- \\ j \text{ non-integer}}} \frac{f_0}{1-f_0} \bar{a}_j x_j \geq f_0. \end{aligned} \tag{18}$$

Gomory [26] shows that an algorithm based on iteratively adding these inequalities solves $\min\{c^T x : x \in X\}$ with $X = \{x \in \mathbb{Z}_+^p \times \mathbb{R}_+^{n-p} : Ax = b\}$ in a finite number of steps provided $c^T x \in \mathbb{Z}$ for all $x \in X$.

Note that Gomory's mixed integer cuts can always be applied, the separation problem for the optimal LP solution is easy. However, adding these inequalities might cause numerical difficulties, see the discussion in [60]. In [7,11] it is shown how useful Gomory cuts are if they are incorporated in the right way.

5.2 Cutting Planes Exploiting Structure

In this section we follow a different route to derive cutting planes and analyze the structure of the constraint matrix. The idea is to identify some substructure of $Ax \leq b$ and use the polyhedral knowledge about this substructure to strengthen the original formulation. Let $A_{IJ}x_J \leq b_I$ with $I \subseteq M, J \subseteq N \cup C$ be such a subsystem of (2). If $J = N \cup C$, we have that $P_{\text{IP}} \subseteq \{x \in \mathbb{Z}^N \times \mathbb{R}^C : A_{I\cdot}x \leq b_I\} =: P_{\text{rel}}$ and any cutting plane valid for P_{rel} is also valid for P_{IP}. Thus the task is to identify some substructure where one knows (part of) the polyhedral structure and to find violated cutting planes for this substructure. This approach was initiated by Crowder, Johnson and Padberg [20] for 0/1 integer programs, where each row of the constraint matrix was interpreted as a knapsack problem. Since this approach is still very common to many MIP solvers and is still very successful, we describe some of the cutting planes known for the 0/1 knapsack polytope that are used to strengthen general mixed integer programs. In case J is a proper subset of $N \cup C$, a valid inequality for P_{rel} is not necessarily valid for P_{IP}. In this case we have to resort to *lifting*. The main idea of lifting will be described at the end of this section. As we will see lifting is also useful to strengthen valid inequalities.

Knapsack Relaxations

Consider the following polytope

$$P_K(N, f, F) := \text{conv}\{x \in \{0,1\}^N : \sum_{i \in N} f_j x_j \leq F\} \tag{19}$$

with some finite set N, weights $f_j \in \mathbb{Q}$, $j \in N$, and capacity $F \in \mathbb{Q}$. $P_K(N, f, F)$ is called the 0/1 *knapsack polytope*. We obtain a knapsack relaxation from our integer program (2) by taking some row i and setting $f_j = a_{ij}$ and $F = b_i$, where we assume that all involved variables are binary. Thus any valid inequality for $P_K(N, f, F)$ is also valid for P_{IP}. In the following we summarize some of the inequalities known for the 0/1 knapsack polytope that are also used for the solution of integer programs.

A set $S \subseteq N$ is called a *cover* if its weight exceeds the capacity, i. e., if $\sum_{i \in S} f_i > F$. With the cover S one can associate the *cover inequality*

$$\sum_{i \in S} x_i \leq |S| - 1$$

that is valid for the knapsack polyhedron $P_K(N, f, F)$. If the cover is *minimal*, i. e., if $\sum_{i \in S \setminus \{s\}} f_i \leq F$ for all $s \in S$, the inequality is called *minimal cover inequality (with respect to S)*. In [4,57,36,69] it was shown that the minimal cover inequality defines a facet of $P_K(S, f, F)$.

Another well-known class of knapsack inequalities are $(1, k)$-configuration inequalities that were introduced by Padberg [58]. A $(1, k)$-*configuration* consists

of a feasible set S, i. e., a set S such that $\sum_{i\in S} f_i \leq F$, plus one additional item z such that every subset of S of cardinality k, together with z, forms a minimal cover. A $(1,k)$-configuration $S \cup \{z\}$ gives rise to the inequality

$$\sum_{i\in S} x_i + (|S| - k + 1)x_z \leq |S|,$$

which is called a $(1,k)$-*configuration inequality (with respect to* $S \cup \{z\}$*)*. Note that a minimal cover S is a $(1, |S| - 1)$-configuration, and vice versa, a $(1,k)$-configuration inequality (with respect to $S \cup \{z\}$) that satisfies $k = |S|$ is a minimal cover. In [58] it was shown that the $(1,k)$-configuration inequality defines a facet of $P_K(S \cup \{z\}, f, F)$.

Inequalities derived from both covers and $(1,k)$-configurations are special cases of *extended weight inequalities* that have been introduced by Weismantel [68]. Consider a subset $T \subseteq N$ with $f(T) < F$ and let $r := F - f(T)$. The inequality

$$\sum_{i\in T} f_i x_i + \sum_{i\in N\setminus T} (f_i - r)^+ x_i \leq f(T). \tag{20}$$

is called *weight inequality with respect to* T. It is valid for $P_K(N, f, F)$. The name *weight inequality* reflects that the coefficients of the items in T equal their original weights and the number $r := F - f(T)$ corresponds to the residual capacity of the knapsack when $x_i = 1$ for all $i \in T$. There is a natural way to extend weight inequalities by (i) replacing the original weights of the items by relative weights and (ii) resorting to the method of sequential lifting.

Consider again some subset $T \subseteq N$ with $f(T) \leq F$, let $r = F - f(T)$ and denote by S the subset of $N \setminus T$ such that $f_i \geq r$ for all $i \in S$. The *(uniform) extended weight inequality* associated with T and some permutation $\pi_1, \ldots \pi_{|S|}$ of the set S is of the form

$$\sum_{i\in T} x_i + \sum_{i\in S} w_i x_i \leq |T|, \tag{21}$$

where $w_i, i \in S$, are the lifting coefficients obtained by applying Algorithm 2 on page 20. These (uniform) extended weight inequalities subsume the family of minimal cover and $(1,k)$-configuration inequalities. They can be generalized to inequalities with arbitrary weights in the starting set T, see [68].

The separation of minimal cover inequalities is widely discussed in the literature. The complexity of cover separation has been investigated in [23,42,32], whereas algorithmic and implementational issues are treated among others in [20,33,37,62,72]. The ideas and concepts suggested to separate cover inequalities basically carry over to extended weight inequalities. Typical features of a separation algorithm for cover inequalities are: fix all variables that are integer, find a cover (in the extended weight case some subset T), and lift the remaining variables sequentially.

Cutting planes derived from knapsack relaxations can sometimes be strengthened if special ordered set (SOS) inequalities $\sum_{i\in Q} x_i \leq 1$ for some $Q \subseteq N$ are

available. In connection with a knapsack inequality these constraints are also called *generalized upper bound constraints (GUBs)*. It is clear that by taking the additional SOS constraints into account stronger cutting planes may be derived. This possibility has been studied in [20,41,71,54,33].

Lifting

As outlined at the beginning of this section and as observed in the description and separation of knapsack inequalities we are often faced with the following problem.

We are given an inequality $\sum_{i\in I}\alpha_i x_i \le \alpha_0$ that is valid for $P_{\text{IP}} \cap \{x \in \mathbb{R}^n : x_j = 0 \text{ for all } j \in N\setminus I\}$ for some $I \subseteq N$. We would like to extend this inequality to a valid inequality of P_{IP} and, if possible, in such a way that it induces a high dimensional face of P_{IP}. Or in case $\sum_{i\in I}\alpha_i x_i \le \alpha_0$ is already facet-defining for the subpolytope (as for instance the minimal cover for $P_K(S,f,F)$), we would like to extend the inequality to a facet-defining inequality of P_{IP} (in the minimal cover case to a facet-defining inequality of $P_K(N,f,F)$). One way to solve this problem is the *method of sequential lifting*, see [57,69]. The algorithm proceeds in an iterative fashion. It takes into account step by step a variable $i \in N\setminus I$, computes an appropriate coefficient α_i for this variable and iterates. We assume in the following that $\pi_1,\dots,\pi_{n-|I|}$ is a permutation of the items in $N\setminus I$.

Algorithm 2. (Sequential lifting)
(1) **For** $k = 1$ to $n-|I|$ perform the following steps:
(2) **For** $l = 1$ to u_{π_k} perform the following steps:

$$\begin{aligned}\gamma(k,l) = \max \sum_{i\in I}\alpha_i x_i + \sum_{i\in\{\pi_1,\dots,\pi_{k-1}\}}\alpha_i x_i\\ \sum_{i\in I} A_{\cdot i}x_i + \sum_{i\in\{\pi_1,\dots,\pi_{k-1}\}} A_{\cdot i}x_i + A_{\cdot\pi_k} l \le b\\ 0\le x_i\le u_i, x_i\in\mathbb{Z} \text{ for } i\in I\cup\{\pi_1,\dots,\pi_{k-1}\}.\end{aligned}$$

(3) **End(For)**
(4) Set

$$\alpha_{\pi_k} := \min_{l=1,\dots,u_{\pi_k}} \frac{\alpha_0-\gamma(k,l)}{l}.$$

(5) **End(For)**
(6) **Stop.**

It can be shown by induction on k that the output $\sum_{i\in N}\alpha_i x_i \le \alpha_0$ of this algorithm is a valid inequality for P_{IP}. In case, for some $k \in \{\pi_1,\dots,\pi_{n-|I|}\}$, the integer program in (2) is infeasible, i. e., $\gamma(k,l) = -\infty$, for all $l = 1,\dots,u_k$, we may assign any value to α_k and the inequality stays valid. In fact, the following result is true.

Proposition 1. *Let $I \subseteq N$ and $\sum_{i\in I}\alpha_i x_i \le \alpha_0$ an inequality that defines a facet of $P_{IP}\cap\{x\in\mathbb{R}^n : x_j = 0$ for all $j\in N\setminus I\}$. After applying Algorithm 2, the inequality $\sum_{i\in N}\alpha_i x_i \le \alpha_0$ defines a facet of P_{IP}.*

The inequality that results from applying the lifting procedure is dependent on the permutation of the items in the set $N \setminus I$.

Example 1. Consider the knapsack polyhedron

$$P_{\text{IP}} = \text{conv}\{x \in \{0,1\}^6 : 5x_1 + 5x_2 + 5x_3 + 5x_4 + 3x_5 + 8x_6 \leq 17\}.$$

The inequality $x_1 + x_2 + x_3 + x_4 \leq 3$ is valid for $P_{\text{IP}} \cap \{x_5 = x_6 = 0\}$. Choosing the permutation $(5, 6)$ yields the inequality $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \leq 3$. If one chooses the permutation $(6, 5)$ of the items 5 and 6, the resulting inequality reads $x_1 + x_2 + x_3 + x_4 + 2x_6 \leq 3$. Both inequalities are facet-defining for P_{IP}.

Note that in order to perform the lifting procedure one needs to solve a couple of integer programs that - at first sight - appear as difficult as the original problem. Sometimes they are not. For instance, if the integer program is a 0/1 knapsack problem and the starting inequality $\sum_{i \in I} \alpha_i x_i \leq \alpha_0$ is a minimal cover or $(1, k)$-configuration inequality, the lifting coefficients can be computed in polynomial time, see [72]. Sometimes it is possible to determine the exact lifting coefficient without solving integer programs, as was observed by Balas [4] for minimal cover inequalities and extended by Weismantel [68] to extended weight inequalities. It is however true that for many general mixed integer programs the lifting procedure can hardly be implemented in the way we presented it, because computing the coefficients step by step is just too expensive. In such cases, one resorts to lower bounds on the coefficients that one obtains from heuristics. Another way is to look for conditions under which simultaneous lifting of variables is possible. This leads to the study of superadditive functions [70, 35].

We note that lifting can, of course, also be applied if a variable x_i is currently at its upper bound u_i. In this case, we first "complement" variable x_i by replacing it by $u_i - x_i$, apply the same Algorithm 2 and resubstitute the variable afterwards. Lifting (sequential or simultaneous) has also been applied to general mixed integer programs, see, for instance, [34,48] or in connection with lift-and-project cuts, see [7,5] and [⟶ Balas].

Computational results about the success of knapsack inequalities with or without GUB constraints are given, for instance, in [20,11,19,33,49]. The papers consistently show that knapsack cuts are crucial for the solution of integer programs that contain knapsack problems as a substructure.

Of course, knapsack relaxations are not the only ones considered in mixed integer programming solvers. An analysis of other important relaxations of an integer program allows to incorporate odd hole and clique inequalities for the stable set polyhedron [56] or flow cover inequalities for certain mixed integer models [61,62]. Further recent examples of this second approach are given in [15, 48]. More than one knapsack constraint at a time are considered in [50]. Cordier et al. [19] give a nice survey on which of the mentioned cutting planes help to solve which problems from the `Miplib`. A comprehensive survey on cutting planes used to solve integer and mixed integer programs is given in [47].

6 Conclusions

In this paper we have discussed the basic features of current branch-and-cut algorithms for the solution of mixed integer programs. We have especially seen that preprocessing, though most of the ideas are straight-forward, is often very important to solve certain mixed integer programs. We have also observed that there are various alternative and better strategies for node and variable selection within the branch-and-bound enumeration scheme than the classical choices of selecting some node deepest in the tree and selecting a variable closest to one half. We also got to know some cutting planes that are incorporated into todays software. Of course, we could just touch the surface of these topics in this survey. The interested reader is most welcome to get deeper into the field through the cited literature.

References

1. K. Aardal, R. Weismantel, and L.A. Wolsey. Non-standard approaches to integer programming. Technical Report CORE DP2000/2, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, 2000.
2. E. D. Andersen and K. D. Andersen. Presolving in linear programming. *Mathematical Programming*, 71:221 – 245, 1995.
3. D. Applegate, R. E. Bixby, V. Chvátal, and W. Cook. Finding cuts in the TSP. Technical Report 95-05, DIMACS, March 1995.
4. E. Balas. Facets of the knapsack polytope. *Mathematical Programming*, 8:146 – 164, 1975.
5. E. Balas, S. Ceria, and G. Cornuéjols. A lift-and-project cutting plane algorithm for mixed $0-1$ programs. *Mathematical Programming*, 58:295–324, 1993.
6. E. Balas, S. Ceria, and G. Cornuéjols. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42:1229 – 1246, 1996.
7. E. Balas, S. Ceria, G. Cornuéjols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19:1 – 9, 1996.
8. M. Benichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, and O. Vincent. Experiments in mixed-integer programming. *Mathematical Programming*, 1:76 – 94, 1971.
9. R. E. Bixby. *Lectures on Linear Programming.* Rice University, Houston, Texas, Spring 1994.
10. R. E. Bixby, S. Ceria, C. McZeal, and M. W. P. Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. Information on the paper and the problems are available at WWW Page: http://www.caam.rice.edu/∼bixby/miplib/miplib.html, 1998.
11. R.E. Bixby, M. Fenelon, Z. Guand E. Rothberg, and R. Wunderling. MIP: Theory and practice closing the gap. Technical report, ILOG Inc., Paris, France, 1999.
12. R. Borndörfer. *Aspects of Set Packing, Partitioning, and Covering.* PhD thesis, Technische Universität Berlin, 1998.
13. R. Borndörfer, C. E. Ferreira, and A. Martin. Decomposing matrices into blocks. *SIAM Journal on Optimization*, 9:236 – 269, 1998.
14. A. Caprara and M. Fischetti. Branch-and-cut algorithms. In M. Dell'Amico, F. Maffioli, and S. Martello, editors, *Annotated Bibliographies in Combinatorial Optimization*, pages 45–63. John Wiley & Sons Ltd, Chichester, 1997.

15. S. Ceria, C. Cordier, H. Marchand, and L. A. Wolsey. Cutting planes for integer programs with general integer variables. *Mathematical Programming*, 81:201 – 214, 1998.
16. V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4:305 – 337, 1973.
17. V. Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983.
18. J.M. Clochard and D. Naddef. Using path inequalities in a branch-and-cut code for the symmetric traveling salesman problem. In Lawrence Wolsey and Giovanni Rinaldi, editors, *Proceedings on the Third IPCO Conference*, pages 291–311, 1993.
19. C. Cordier, H. Marchand, R. Laundy, and L. A. Wolsey. bc – opt: a branch-and-cut code for mixed integer programs. *Mathematical Programming*, 86:335 – 354, 1999.
20. H. Crowder, E. Johnson, and M. W. Padberg. Solving large-scale zero-one linear programming problems. *Operations Research*, 31:803–834, 1983.
21. DASH Associates, Blisworth House, Blisworth, Northants NN7 3BX, United Kingdom. *XPRESS-MP Optimisation Subroutine Library*, 2001. Information available at URL http://www.dash.co.uk.
22. J. Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5. *SIAM Journal on Optimization*, 4:794 – 814, 1994.
23. C. E. Ferreira. *On Combinatorial Optimization Problems Arising in Computer System Design*. PhD thesis, Technische Universität Berlin, 1994.
24. C. E. Ferreira, A. Martin, and R. Weismantel. Solving multiple knapsack problems by cutting planes. *SIAM Journal on Optimization*, 6:858 – 877, 1996.
25. R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Society*, 64:275 – 278, 1958.
26. R. E. Gomory. An algorithm for the mixed integer problem. Technical Report RM-2597, The RAND Cooperation, 1960.
27. R. E. Gomory. Solving linear programming problems in integers. In R. Bellman and M. Hall, editors, *Combinatorial analysis, Proceedings of Symposia in Applied Mathematics*, volume 10, Providence RI, 1960.
28. R. E. Gomory. An algorithm for integer solutions to linear programming. In R. L. Graves and P. Wolfe, editors, *Recent Advances in Mathematical Programming*, pages 269 – 302, New York, 1969. McGraw-Hill.
29. J. Gondzio. Presolve analysis of linear programs prior to apply an interior point method. *INFORMS Journal on Computing*, 9:73 – 91, 1997.
30. M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169 – 197, 1981.
31. M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer, 1988.
32. Z. Gu, G. L. Nemhauser, and M. W. P. Savelsbergh. Cover inequalities for $0 - 1$ linear programs: complexity. *INFORMS Journal on Computing*, 11:117 – 123, 1998.
33. Z. Gu, G. L. Nemhauser, and M. W. P. Savelsbergh. Cover inequalities for $0 - 1$ linear programs: computation. *INFORMS Journal on Computing*, 10:427 – 437, 1998.
34. Z. Gu, G. L. Nemhauser, and M. W. P. Savelsbergh. Lifted flow cover inequalities for mixed 0-1 integer programs. *Mathematical Programming*, 85:439 – 468, 1999.
35. Z. Gu, G. L. Nemhauser, and M. W. P. Savelsbergh. Sequence independent lifting in mixed integer programming. *Journal on Combinatorial optimization*, 4:109 – 129, 2000.
36. P. L. Hammer, E. .L. Johnson, and U. N. Peled. Facets of regular 0-1 polytopes. *Mathematical Programming*, 8:179 – 206, 1975.

37. K. L. Hoffman and M. W. Padberg. Improved LP-representations of zero-one linear programs for branch-and-cut. *ORSA Journal on Computing*, 3:121–134, 1991.
38. ILOG CPLEX Division, 889 Alder Avenue, Suite 200, Incline Village, NV 89451, USA. *Using the CPLEX Callable Library*, 1997. Information available at URL http://www.cplex.com.
39. ILOG CPLEX Division, 889 Alder Avenue, Suite 200, Incline Village, NV 89451, USA. *Using the CPLEX Callable Library*, 2000. Information available at URL http://www.cplex.com.
40. Jünger, M., Thienel, S.: The ABACUS System for Branch and Cut and Price Algorithms in Integer Programming and Combinatorial Optimization. Software Practice and Experience **30**, 1325–1352, 2000.
41. E. Johnson and M. W. Padberg. A note on the knapsack problem with special ordered sets. *Operations Research Letters*, 1:18 – 22, 1981.
42. D. Klabjan, G. L. Nemhauser, and C. Tovey. The complexity of cover inequality separation. *Operations Research Letters*, 23:35 – 40, 1998.
43. T. Koch and A. Martin. Solving Steiner tree problems in graphs to optimality. *Networks*, 32:207 – 232, 1998.
44. A. Land and S. Powell. Computer codes for problems of integer programming. *Annals of Discrete Mathematics*, 5:221 – 269, 1979.
45. J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies for mixed integer programming. Technical Report LEC-97-12, Georgia Institute of Technology, 1997.
46. H. Marchand. *A Polyhedral Study of the Mixed Knapsack Set and its Use to Solve Mixed Integer Programs*. PhD thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, 1998.
47. H. Marchand, A. Martin, R. Weismantel, and L.A. Wolsey. Cutting planes in integer and mixed integer programming. Technical Report CORE DP9953, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, 1999.
48. H. Marchand and L. A. Wolsey. The $0-1$ knapsack problem with a single continuous variable. *Mathematical Programming*, 85:15 – 33, 1999.
49. A. Martin. Integer programs with block structure. Habilitations-Schrift, Technische Universität Berlin, 1998.
50. A. Martin and R. Weismantel. The intersection of knapsack polyhedra and extensions. In R. E. Bixby, E. A. Boyd, and R. Z. Ríos-Mercado, editors, *Integer Programming and Combinatorial Optimization*, Proceedings of the 6th IPCO Conference, pages 243 – 256, 1998.
51. G. Mitra. Investigations of some branch and bound strategies for the solution of mixed integer linear programs. *Mathematical Programming*, 4:155 – 170, 1973.
52. D. Naddef. Polyhedral theory and branch-and-cut algorithms for the symmetric tsp. In G. Gutin and A. Punnen, editors, *The Traveling Salesman Problem and its Variations*. Kluwert, 2001. To appear.
53. G. L. Nemhauser, M. W. P. Savelsbergh, and G. C. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47 – 58, 1994.
54. G. L. Nemhauser and P. H. Vance. Lifted cover facets of the $0-1$ knapsack polytope with GUB constraints. *Operations Research Letters*, 16:255 – 263, 1994.
55. G. L. Nemhauser and L. A. Wolsey. A recursive procedure to generate all cuts for $0-1$ mixed integer programs. *Mathematical Programming*, 46:379 – 390, 1990.
56. M. W. Padberg. On the facial structure of set packing polyhedra. *Mathematical Programming*, 5:199–215, 1973.
57. M. W. Padberg. A note on zero-one programming. *Operations Research*, 23:833–837, 1975.

58. M. W. Padberg. $(1, k)$-configurations and facets for packing problems. *Mathematical Programming*, 18:94–99, 1980.
59. M. W. Padberg. *Linear Optimization and Extensions*. Springer, 1995.
60. M. W. Padberg. Classical cuts for mixed-integer programming and branch-and-cut. Technical report, New York University, 2000. To appear in MMOR.
61. M. W. Padberg, T. J. Van Roy, and L. A. Wolsey. Valid inequalities for fixed charge problems. *Operations Research*, 33:842 – 861, 1985.
62. T. J. Van Roy and L. A. Wolsey. Solving mixed integer programming problems using automatic reformulation. *Operations Research*, 35:45 – 57, 1987.
63. A. Schrijver. On cutting planes. *Annals of Discrete Mathematics*, 9:291 – 296, 1980.
64. Ramesh Sharda. Linear programming solver software for personal computers: 1995 report. *OR/MS Today*, 22(5):49 – 57, 1995.
65. Uwe H. Suhl and R. Szymanski. Supernode processing of mixed-integer models. *Computational Optimization and Applications*, 3:317 – 331, 1994.
66. Stefan Thienel. *ABACUS A Branch-And-CUt System*. PhD thesis, Universität zu Köln, 1995.
67. J. A. Tomlin and J. S. Welsh. Finding duplicate rows in a linear program. *Operations Research Letters*, 5:7 – 11, 1986.
68. R. Weismantel. On the 0/1 knapsack polytope. *Mathematical Programming*, 77:49–68, 1997.
69. L. A. Wolsey. Faces of linear inequalities in 0-1 variables. *Mathematical Programming*, 8:165 – 178, 1975.
70. L. A. Wolsey. Valid inequalities and superadditivity for 0/1 integer programs. *Mathematics of Operations Research*, 2:66 – 77, 1977.
71. L. A. Wolsey. Valid inequalities for 0-1 knapsacks and MIPs with generalized upper bound constraints. *Discrete Applied Mathematics*, 29:251–261, 1990.
72. E. Zemel. Easily computable facets of the knapsack polytope. *Mathematics of Operations Research*, 14:760 – 764, 1989.

Projection and Lifting in Combinatorial Optimization

Egon Balas

Carnegie Mellon University, Mathematical Sciences Department,
Wean Hall, Pittsburgh, PA 15213, USA

Abstract. This is an overview of the significance and main uses of projection and lifting in integer programming and combinatorial optimization. Its first four sections deal with those basic properties of projection that make it such an effective and useful bridge between problem formulations in different spaces, i.e. different sets of variables. They discuss topics like the integrality-preserving property of projection, the dimension of projected polyhedra, when do facets of a polyhedron project into facets of its projection, and so on. They also describe the use of projection for comparing the strength of different formulations of the same problem, and for proving the integrality of polyhedra by using extended formulations. The next section of the survey deals with disjunctive programming, or optimization over unions of polyhedra, whose most important incarnation are mixed 0-1 programs and their partial relaxations. It discusses the compact representation of the convex hull of a union of polyhedra through extended formulation, the connection between the projection of the latter and the polar of the convex hull, as well as the sequential convexification of facial disjunctive programs, among them mixed 0-1 programs, with the related concept of disjunctive rank. Finally, the last two sections review the recent developments in disjunctive programming, namely lift-and-project cuts, the construction of cut generating linear programs, techniques for lifting and for strengthening disjunctive cuts, and the embedding of these cuts into a branch-and-cut framework, along with a brief outline of computational results.

1 Basic Properties of Projection

Most combinatorial optimization problems tend to have several alternative formulations, some of which are easier to handle than others. Projection provides a connection between different formulations. Since most combinatorial optimization problems are solved by some combination of polyhedral methods and enumeration, and since the latter typically involves repeated solution of variants of the problem's linear programming relaxation, it is crucial to be able to compare the strength of the LP relaxations of different formulations of the same problem.

Given a polyhedron of the form

$$Q := \{(u, x) \in \mathbb{R}^p \times \mathbb{R}^q : Au + Bx \leq b\},$$

M. Jünger and D. Naddef (Eds.): Computat. Comb. Optimization, LNCS 2241, pp. 26–56, 2001.
© Springer-Verlag Berlin Heidelberg 2001

where A, B and b have m rows, the *projection of* Q *onto* $\mathbb{R}^q$, or *onto the* x*-space*, is defined as

$$\mathrm{Proj}_x(Q) := \{x \in \mathbb{R}^q : \exists u \in \mathbb{R}^p : (u, x) \in Q\}.$$

Thus projecting the set Q onto $\mathbb{R}^q$ can be viewed as finding a matrix $C \in \mathbb{R}^{m \times q}$ and a vector $d \in \mathbb{R}^m$ such that

$$\mathrm{Proj}_x(Q) = \{x \in \mathbb{R}^q : Cx \leq d\}.$$

This task can be accomplished as follows. Call the polyhedral cone

$$W := \{v : vA = 0, v \geq 0\}$$

the *projection cone* associated with $\mathrm{Proj}_x(Q)$. Then we have

Theorem 1.

$$Proj_x(Q) = \{x \in \mathbb{R}^q : (vB)x \leq vb, v \in \mathrm{extr}\, W\}$$

where $\mathrm{extr}\, W$ *denotes the set of extreme rays of* W*.*

Proof. W is a pointed cone (since $W \subseteq \mathbb{R}^m_+$), hence it is the conical hull of its extreme rays. Therefore the system of inequalities defining $\mathrm{Proj}_x(Q)$ holds for all $x \in \mathrm{extr}\, W$ if and only if it hold for all $x \in W$.

Now let $\bar{x} \in \mathrm{Proj}_x(Q)$. Then there exists $\bar{u} \in \mathbb{R}^p$ such that $A\bar{u} + B\bar{x} \leq b$. Premultiplying this system with any $v \in W$, $\bar{x}$ satisfies $(vB)x \leq vb$ for all $v \in W$.

Conversely, suppose $\hat{x} \in \mathbb{R}^q$ satisfies $(vB)x \leq vb$ for all $v \in W$. Then there exists no $v \geq 0$ satisfying $vA = 0$ and $v(B\hat{x} - b) > 0$. But then, by Farkas' Lemma, there exists $\hat{u}$ satisfying $A\hat{u} \leq b - B\hat{x}$, i.e. $\hat{x} \in \mathrm{Proj}_x(Q)$.

□

Several comments are in order.

First, if some of the inequalities defining Q are replaced with equations, the corresponding components of v become unrestricted in sign. In such a case, if the projection cone W is not pointed, then it is not the convex hull of its extreme rays. Nevertheless, like any polyhedral cone, W is finitely generated, and any finite set $\widehat{W}$ of its generators can play the role previously assigned to $\mathrm{extr}\, W$.

Second, the set Q may be defined, besides the system of inequalities involving the variables u to be projected out, by any additional constraints – not necessarily linear – not involving u: such constraints will become part of the projection, without any change. In other words, for an arbitrary set $S \subseteq \mathbb{R}^q$, the projection of

$$Q := \{(u, x) \in \mathbb{R}^p \times \mathbb{R}^q : Au + Bx \leq b, x \in S\}$$

on $\mathbb{R}^q$ is

$$\mathrm{Proj}_x(Q) := \{x \in \mathbb{R}^q : (vB)x \leq vb \text{ for all } v \in W, x \in S\}$$

where W is the projection cone introduced earlier.

Third, $\text{Proj}_x(Q)$ as given by Theorem 1 may have redundant inequalities, even when the latter come only from extreme rays of W; i.e., an extreme ray of W does not necessarily give rise to a facet of $\text{Proj}_x(Q)$. It would be nice to be able to identify the extreme rays of W which give rise to facets of $\text{Proj}_x(Q)$, but unfortunately this in general is not possible: whether an inequality $(vB)x \leq vb$ is or is not facet defining for $\text{Proj}_x(Q)$ depends on B and b as well as on v. We will return to this question later.

Fourth, we have the simple fact following from the definitions, that if $x \in \mathbb{R}^q$ is an extreme point of $\text{Proj}_x(Q)$, then there exists $u \in \mathbb{R}^p$ such that (u, x) is an extreme point of Q. This has the following important consequence:

Proposition 1. *If Q is an integral polyhedron, then $Proj_x(Q)$ is an integral polyhedron.*

Well Known Special Cases. There are two well known algorithms whose iterative steps are special cases of projection:

If the matrix A in the definition of Q has a single column a, i.e., if

$$Q := \{(u_0, x) : au_0 + Bx \leq b\},$$

then

$$\text{extr}\, W = \{v^k : a_k = 0\} \cup \{v^{ij} : a_i \cdot a_j < 0\},$$

where

$$v^k := e_k, \quad v^{ij} := a_i e_j - a_j e_i,$$

with e_k the k-th unit vector, and we have one step of *Fourier Elimination* (see e.g. [39]).

If Q is of the form

$$Q := \left\{ (u, x, x_0) \,\middle|\, \begin{array}{ll} -cu - dx + x_0 & \leq 0 \\ A'u + B'x & \leq b' \end{array} \right\},$$

then

$$W = \{(v_0, v) : -v_0 c + vA' = 0; v_0, v \geq 0\},$$

$$\text{Proj}_x(Q) := \left\{ (x_0, x) \,\middle|\, \begin{array}{r} x_0 + (vB' - d)x \leq vb',\ \forall v : (1, v) \in \text{extr}\, W \\ (vB')x \leq vb',\ \forall v : (0, v) \in \text{extr}\, W \end{array} \right\}$$

and we have the constraint set of the *Benders Master Problem* (see e.g. [33]).

Projection and Restriction. Projection is not to be confused with *restriction*, another operation that relates a higher dimensional object to one in a subspace. Referring to the same set $Q \subseteq \mathbb{R}^p \times \mathbb{R}^q$, its restriction to the subspace defined by $u = 0$ is

$$\{(u, x) \in Q : u = 0\} = \{x \in \mathbb{R}^q : (0, x) \in Q\}.$$

Notice that, unlike in the case of projection, the restriction of Q "to $\mathbb{R}^q$," or "to the x-space," is not well defined, since it may mean several things, of which the above is just one. Another one is, for some $t \in \mathbb{R}^p$,

$$\{(u, x) \in Q : u = t\} = \{x \in \mathbb{R}^q : (t, x) \in Q\}.$$

Clearly, the restriction of Q through some value assignment to each component of u is always a subset of $\text{Proj}_x(Q)$.

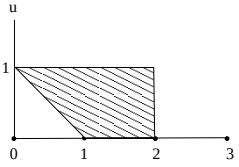


Figure 1a: Q

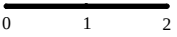


Figure 1b: $\text{Proj}_x Q$

Figure 1c: $\{(x, u) \in Q : u = 0\}$

Fig. 1.

To illustrate the difference between projection and restriction, consider the set $Q := \{(x, u) \in \mathbb{R}^2 : x + u \geq 1,\ 0 \leq x \leq 2,\ 0 \leq u \leq 1\}$. Its projection on the x-space is $\{x \in \mathbb{R} : 0 \leq x \leq 2\}$, whereas its restriction to the subspace of $\mathbb{R}^2$ defined by $u = 0$ is $\{x \in \mathbb{R} : 1 \leq x \leq 2\}$, as shown in Fig. 1.

While both projection and restriction map polyhedra from a higher dimensional space to a lower dimensional one, sometimes we want to go into the opposite direction. The operation, in a sense reverse to projection, in which one goes from a given polyhedral formulation of a problem to a higher dimensional one, involving some new variables, is called *extended formulation*. Sometimes going to an extended formulation is referred to as lifting. For instance, many problems defined on graphs that are usually formulated in terms of arc variables, can also be formulated in the higher dimensional space of arc- and node-variables; and we will see examples where this is advantageous. Typically, extended formulations are not unique, and it takes some insight to recognize those which offer some advantage. One possible advantage might be that the extended formulation gives rise to a polyhedron with desirable properties.

The reverse of restriction of course is relaxation, i.e. the removal of the restricting constraint. But this operation becomes less trivial than it sounds when looked at in the following context. Suppose we know a valid inequality, $\alpha x \leq \alpha_0$, for a polyhedron $P \cap S$ that is a restriction to a subspaces of a higher dimensional polyhedron P, and we would like to calculate the strongest (tightest) extension of the inequality, say $\alpha x + \beta y \leq \alpha_0$, to P. This is a well defined and important operation called *lifting*. It consists of calculating the missing coefficients β_j of the inequality $\alpha x + \beta y \leq \alpha_0$ valid for P. There is a well known procedure for doing this, called *sequential lifting* (see e.g. [33]), which calculates the coefficients β_j one by one, and under certain conditions guarantees that if $\alpha x \leq \alpha_0$ defines a facet of $P \cap S$, then $\alpha x + \beta y \leq \alpha_0$ defines a facet of P.

Exercise 1. Consider the following variations of Q:

$$Q_1 := \{(u,x) \in \mathbb{R}^p \times \mathbb{R}^q : Au + Bx = b\}$$
$$Q_2 := \{(u,x) : \mathbb{R}^p \times \mathbb{R}^q : Au + Bx \leq b,\ u \geq 0\}$$
$$Q_3 = Q \text{ with } B = \begin{pmatrix} I \\ 0 \end{pmatrix}, \text{where } I \text{ is the } q \times q \text{ identity matrix and } 0 \text{ is the } (m-q) \times q \text{ zero matrix (assume } m > q).$$

For each Q_i, $i = 1, 2, 3$ write down the projection cone associated with $\text{Proj}_x(Q_i)$.

Exercise 2. Prove Proposition 1.

2 Dimensional Aspects of Projection

When we have a polyhedral representation of some combinatorial object, we would like it to be nonredundant. This makes us interested in identifying among the inequalities defining the polyhedron, say Q, those which are facet inducing, since the latter provide a minimal representation. Further, when we project Q onto a subspace, we would like to know whether the facets of Q project into facets of the projection. To be able to answer this and similar questions, we need to look at the relationship between the dimension of a polyhedron and the dimension of its projections. This was done in the paper [12], on which most of this section is based.

Consider the polyhedron $Q := \{(u,x) \in \mathbb{R}^p \times \mathbb{R}^q : Au + Bx \leq b\} \neq \emptyset$, where A, B and b have m rows, and let us partition (A, B, b) into $(A^=, B^=, b^=)$ and $(A^\leq, B^\leq, b^\leq)$, where $A^=u + B^=x = b^=$ is the *equality subsystem* of Q, the set of equations corresponding to the inequalities satisfied at equality by every $(u,x) \in Q$. W.l.o.g., we may assume that the equality subsystem is of full row rank (otherwise the redundant rows can be removed), and let $r := \text{rank}(A^=, B^=) = \text{rank}(A^=, B^=, b^=)$, where the last equality follows from $Q \neq \emptyset$.

Let $\dim(Q)$ denote the dimension of Q. It is well known that $\dim(Q) = p + q - r$, and that Q is full-dimensional, i.e. $\dim(Q) = p + q$, if and only if the equality subsystem is vacuous. When this is the case, then $\dim(\text{Proj}_x(Q)) = q$, for otherwise $\text{Proj}_x(Q)$ has a nonempty equality subsystem that must also be valid for Q, contrary to the assumption that the latter is full dimensional.

Consider now the general situation, when Q is not necessarily full dimensional. Recall that $r = \text{rank}(A^=, B^=)$, and define $r^* := \text{rank}(A^=)$. Clearly, $0 \leq r^* \leq \min\{r, p\}$. It can then be shown that

Theorem 2. [12] $\dim(Proj_x(Q)) = \dim(Q) - p + r^*$.

It follows that $\dim(\text{Proj}_x(Q)) = \dim(Q)$ if and only if $r^* = p$, i.e., the projection operation is dimension-preserving if and only if the matrix $A^=$ is of full column rank.

When Is the Projection of a Facet a Facet of the Projection? We now turn to the question of what happens to the facets of Q under projection. Since a facet of a polyhedron is itself a polyhedron, the dimension of its projection can be deduced from Theorem 2.

Let $\alpha u + \beta x \leq \beta_0$ be a valid inequality for Q, and suppose

$$F := \{(u, x) \in Q : \alpha u + \beta x = \beta_0\}$$

is a facet of Q. Let $\binom{\alpha}{A}^{=} u + \binom{\beta}{B}^{=} x = \binom{\beta_0}{b}^{=}$ be the equality subsystem of F, and let $r_F := \operatorname{rank}\left(\binom{\alpha}{A}^{=}, \binom{\beta}{B}^{=}\right)$. Notice that $r_F - r = 1$, since by assumption $\dim(F) = \dim(Q) - 1$. Further, denote $r_F^* := \operatorname{rank}\left(\binom{\alpha}{A}^{=}\right)$. We may interpret $r_F^* - r^*$ as the difference between the number of dimensions "lost" in projecting Q (which is $p - r^*$) and in projecting F (which is $p - r_F^*$). From Theorem 2 we then have

Corollary 1. $\dim(Proj_x(F)) = \dim(Proj_x(Q)) - 1 + (r_F^* - r^*)$

Indeed, $\dim(\mathrm{Proj}_x(F)) = \dim(F) - p + r_F^* = \dim(Q) - 1 - p + r_F^*$, and substituting for $\dim(Q)$ its value given by Theorem 2 yields the Corollary.

We are now in a position to state what happens to the facets of Q under projection.

Corollary 2. *Let F be a facet of Q. Then $Proj_x(F)$ is a facet of $Proj_x(Q)$ if and only if $r_F^* = r^*$.*

Proof outline. From Corollary 1 it is clear that $\mathrm{Proj}_x(F)$ has the dimension of a facet of $\mathrm{Proj}_x(Q)$ if and only if $r_F^* = r^*$. However, this by itself does not amount to a proof, unless we can guarantee that $\mathrm{Proj}_x(F)$ is a face of $\mathrm{Proj}_x(Q)$, which is far from obvious: in general, the projection of a face need not be a face of the projection. Indeed, think of a 3-dimensional pyramid, and its projection on its base: neither the top vertex of the pyramid, nor the 3 edges incident with it, become faces of the projection as a result of the operation.

However, if $\mathrm{Proj}_x(F)$ is a set of the form $\{x \in \mathrm{Proj}_x(Q) : \beta x = \beta_0\}$ for some valid inequality $\beta x \leq \beta_0$ for $\mathrm{Proj}_x(Q)$, then clearly $\mathrm{Proj}_x(F)$ is a face of $\mathrm{Proj}_x(Q)$. This is the case in our situation: since F is a facet of Q, it has a defining inequality $\alpha u + \beta x \leq \beta_0$. Now if $r_F^* = r^*$, then there exists a vector λ such that $\alpha = \lambda A^{=}$; hence subtracting $\lambda A^{=} u + \lambda B^{=} x = \lambda b^{=}$ from this defining inequality yields $(\beta - \lambda B^{=})x \leq \beta_0 - \lambda b^{=}$, which can be written as $\beta' x \leq \beta_0'$. Thus $\mathrm{Proj}_x(F) = \{x \in \mathrm{Proj}_x(Q) : \beta' x = \beta_0'\}$, i.e. $\mathrm{Proj}_x(F)$ is a face of $\mathrm{Proj}_x(Q)$; hence a facet.

□

Another consequence of Theorem 2 which is not hard to prove is this:

Corollary 3. *Let $r^* = p$, i.e. let $A^{=}$ be of full column rank, and let $0 \leq d \leq \dim(Q) - 1$. Then every d-dimensional face of Q projects into a d-dimensional face of $Proj_x(Q)$.*

Projection with a Minimal System of Inequalities. As discussed in section 1, the inequalities of the system defining $\text{Proj}_x(Q)$ are not necessarily facet inducing, even though they are in 1-1 correspondence with the extreme rays of the projection cone W. In other words, the system $(vB)x \leq vb$, $\forall v \in \text{extr}\, W$, is not necessarily minimal. In fact, this system often contains a large number of redundant inequalities. It has been shown however [6], that if a certain linear transformation is applied to Q, which replaces the coefficient matrix B of x with the identity matrix I, the resulting polyhedron Q^0 has a projection cone W^0 that is pointed, and the projection of Q^0 is of the form

$$\text{Proj}_x(Q^0) = \{x \in \mathbb{R}^q : vx \leq v_0 \text{ for all } (v, v_0) \in \mathbb{R}^{m+1} \text{ such that } (v, w, v_0) \in \text{extr}\, W^0\},$$

with $\text{Proj}_x(Q^0) = \text{Proj}_x(Q)$. Here w is a set of auxiliary variables generated by the above transformation. Furthermore, we have the following

Proposition 2. **[6]** *Let $Proj_x(Q)$ be full dimensional. Then the inequality $vx \leq v_0$ defines a facet of $Proj_x(Q)$ if and only if (v, v_0) is an extreme ray of the cone $Proj_{(v,v_0)}(W^0)$.*

The linear transformation that takes Q into Q^0 is of low complexity (namely $O(\max\{m, q\}^3)$, where B is $m \times q$). The need to project the cone W^0 onto the subspace (v, v_0) arises only when B is not of full row rank. In the important case when B is of full row rank, we have the following stronger result:

Corollary 4. **[6]** *Let $Proj_x(Q)$ be full dimensional and* $\text{rank}(B) = m$. *Then the inequality $vx \leq v_0$ defines a facet of $Proj_x(Q)$ if and only if (v, v_0) is an extreme ray of W^0.*

In many important cases the matrix B is of the form $B = \binom{I}{0}$, which voids the need for the transformation discussed above and leads to a projection cone whose extreme rays yield facet inducing inequalities for $\text{Proj}_x(Q)$. This is the case encountered, for instance, in the characterization of the perfectly matchable subgraph polytope of a graph to be discussed in section 4; as well as in the projection used in the convex hull characterization of a disjunctive set discussed in section 5.

Exercise 3. Give an example of a 3-dimensional polyhedron Q and one of its 2-dimensional projections $\text{Proj}(Q)$ such that $\text{Proj}(Q)$ has (a) fewer facets than Q; (b) more facets than Q; (c) the same number of facets as Q (visual representation will suffice).

3 Comparing Different Formulations

Combinatorial optimization problems tend to have several formulations, and choosing the most convenient one is often a nontrivial exercise. While the number of variables and constraints in the different formulations do have some importance, the most relevant criterion of the comparison is the strength of the

linear programming relaxation. This is so because most solution procedures involve some branch and bound component, and the bounding usually comes from the LP relaxation: the tighter this relaxation, the stronger the bound.

Typically the comparisons are made between two problem formulations, say P and Q, such that P is expressed in terms of variables $x \in \mathbb{R}^n$, while Q is in terms of variables $(x, y) \in \mathbb{R}^n \times \mathbb{R}^q$. To compare the strength of the two LP relaxations, one has to express both P and Q in terms of the same variables, and so one uses projection to eliminate y and re-express Q in terms of x, i.e., as $\text{Proj}_x(Q)$. One then compares the LP relaxation of this latter set to that of P.

Before we give several examples of how this can be done, we wish to point out that projection can be used to compare two different formulations of the same problem even when they are expressed in completely different (nonoverlapping) sets of variables, provided the two sets are related by an affine transformation. Suppose, for instance, that we have two equivalent formulations of a problem whose feasible sets are P and Q:

$$P := \{x \in \mathbb{R}^n : Ax \leq a_0\}, \quad Q := \{y \in \mathbb{R}^q : By \leq b_0\}$$

where A is $m \times n$, B is $p \times q$, and

$$x = Ty + c \tag{1}$$

for some $T \subset \mathbb{R}^n \times \mathbb{R}^q$ and $c \in \mathbb{R}^n$.

Define

$$Q^+ := \left\{ (x, y) \in \mathbb{R}^{n+q} \,\middle|\, \begin{array}{r} x - Ty = c \\ By \leq b_0 \end{array} \right\}$$

and project Q^+ onto $\mathbb{R}^n$. Using the projection cone

$$W := \{(v, w) \in \mathbb{R}^{n+p} : -vT + wB = 0, w \geq 0\},$$

we obtain

$$\text{Proj}_x(Q^+) := \{x \in \mathbb{R}^n : vx \leq vc + wb_0, \forall (v, w) \in W\}.$$

Then to compare P with Q, we compare P with $\text{Proj}_x(Q^+)$.

The affine transformation (1) can also be used to derive a different analytical comparison between P and Q that does not make use of projection [34].

The Traveling Salesman Problem. Consider the constraint set of the TSP defined on the complete digraph G on $n+1$ nodes in two well known formulations:

Dantzig - Fulkerson - Johnson [24]

$$\begin{aligned}
\sum (x_{ij} : j = 0, \ldots, n) &= 1, && i = 0, \ldots, n \\
\sum (x_{ij} : i = 0, \ldots, n) &= 1, && j = 0, \ldots, n \\
\sum (x_{ij} : i \in S, j \in S) &\leq |S| - 1, && S \subseteq \{0, \ldots, n\}, \quad 2 \leq |S| \leq \frac{n+1}{2} \\
x_{ij} &\in \{0, 1\}, && i, j = 0, \ldots, n
\end{aligned}$$

Miller - Tucker - Zemlin [32]

$$\begin{aligned}
\sum(x_{ij} : j = 0, \dots, n) &= 1 \qquad && i = 0, \dots, n \\
\sum(x_{ij} : i = 0, \dots, n) &= 1 \qquad && j = 0, \dots, n \\
u_i - u_j + nx_{ij} &\leq n - 1 \quad && i, j = 1, \dots, n, \quad i \neq j \\
x_{ij} &\in \{0, 1\} \quad && i, j = 0, \dots, n
\end{aligned}$$

The M-T-Z formulation introduces n node variables, but replaces the exponentially large set of subtour elimination constraints by $n(n-1)$ new constraints that achieve the same goal. However, projecting the constraint set of the M-T-Z formulation into the subspace of the arc variables by using the cone

$$W = \{v \mid vA = 0, \quad v \geq 0\}, \tag{2}$$

where A is the transpose of the node-arc incidence matrix of G, yields the inequalities

$$\sum(x_{ij} : (i, j) \in C) \leq \frac{n-1}{n}|C|$$

for every directed cycle C of G. These inequalities are strictly weaker than the corresponding subtour elimination inequalities of the D-F-J formulation, so the latter provides a tighter LP relaxation.

The Set Covering Problem. Consider the set covering problem

$$\min\{cx \mid Ax \geq e, \quad x \in \{0, 1\}^n\} \tag{SC}$$

where $e = (1, \dots, 1)$ and A is a matrix of 0's and 1's. Let M and N index the rows and columns of A, respectively, and

$$M_j = \{i \in M \mid a_{ij} = 1\},\ j \in N, \qquad N_i = \{j \in N \mid a_{ij} = 1\},\ i \in M.$$

The following uncapacitated plant location (UPL) problem is known to be equivalent to (SC):

$$\begin{aligned}
\min \textstyle\sum(c_j x_j : j \in N) & \\
\textstyle\sum(u_{ij} : j \in N_i) \geq 1, & \qquad i \in M \\
-\textstyle\sum(u_{ij} : i \in M_j) + |M_j| x_j \geq 0, & \qquad j \in N \\
u_{ij} \geq 0,\ i \in M,\ j \in N;\ x_j \in \{0, 1\},\ j \in N &
\end{aligned}$$

If we now project this constraint set onto the x-space by using the cone

$$W = \{v | v_i - v_j \leq 0,\ j \in N_i,\ i \in M :\ v_i \geq 0,\ i \in M\},$$

we obtain the inequalities

$$\begin{aligned}
\textstyle\sum(|M_j| x_j : j \in N_S) \geq |S|, & \qquad \forall S \subseteq M; \\
x_j \geq 0, & \qquad j \in N
\end{aligned}$$

(where $N_S := \cup(N_i : i \in S)$, each of which is dominated (strictly if $S \neq M$) by the sum of the inequalities of (SC) indexed by S. Hence the LP relaxation of the UPL formulation is weaker than that of (SC).

If, on the other hand, we use the so called "strong" formulation of the uncapacitated plant location problem, namely

$$\begin{aligned} \min \textstyle\sum(c_j x_j : j \in N) & \\ \textstyle\sum(u_{ij} : j \in N_i) \geq 1 & \quad i \in M \\ -u_{ij} + x_j \geq 0 & \quad i \in M,\ j \in N; \\ u_{ij} \geq 0,\ x_j \in \{0,1\}, & \quad i \in M,\ j \in N; \end{aligned}$$

then the projected inequalities include

$$\begin{aligned} \textstyle\sum(x_j : j \in N_i) \geq 1, & \quad i \in M; \\ x_j \geq 0 & \quad j \in N \end{aligned}$$

thus yielding the same LP relaxation as that of (SC).

Nonlinear 0-1 Programming. Consider the nonlinear inequality

$$\begin{aligned} \sum_{j \in N} a_j (\underset{i \in Q_j}{\pi} x_i) \leq b,\ a_j > 0,\ j \in N & \\ x_i \in \{0,1\},\ i \in Q_j,\ j \in N & \end{aligned}$$

where π denotes product.

It is a well known linearization technique due to R. Fortet [25] to replace this inequality by

$$\begin{aligned} \sum_{j \in N} a_j y_j \qquad\qquad & \leq b \\ -y_j + \sum_{i \in Q_j} x_i & \leq |Q_j| - 1 \qquad j \in N \\ y_j - \qquad x_i & \leq 0 \qquad\quad i \in Q_j,\ j \in N \\ y_j \geq 0,\ x_i \in \{0,1\},\ i \in Q_j,\ j \in N & \end{aligned}$$

When the number of nonlinear terms is small relative to the number of variables, this is perfectly satisfactory. However, often the number of terms is a high-degree polynomial in the number of variables, in which case it is desirable to eliminate the new variables y_j, $j \in N$.

The cone needed for projecting the above constraint set onto the x-space has for every $M \subseteq N$ an extreme direction vector

$$v_M = (1; w_M; 0),$$

where 1 is a scalar, 0 is the zero vector with $\sum(|Q_j| : j \in N)$ components, and $w_M \in \mathbb{R}^n$ is defined by

$$w_j = \begin{cases} a_j \text{ if } j \in M \\ 0 \ \text{ if } j \in N - M. \end{cases}$$

The corresponding projected inequalities are

$$\sum_{i \in Q_M} (\sum_{j: i \in Q_j} a_j) x_i \leq b - \sum_{j \in M} a_j (|Q_j| - 1), \ M \subseteq N,$$

where $Q_M := \cup (Q_j : j \in M)$.

But this is precisely the linearization of Balas and Mazzola [11], arrived at by other means.

Exercise 4. Show that the extreme rays of the cone W defined in (2) are the incidence vectors of the directed cycles of G.

4 Proving the Integrality of Polyhedra

One of the important uses of projection in combinatorial optimization is to prove the integrality of certain polyhedra. It often happens that the LP relaxation of a certain formulation, say P, does not satisfy any of the known sufficient conditions for it to have the integrality property; but there exists a higher dimensional formulation whose LP relaxation, say Q, satisfies such a condition (for the relevant variables). In such a case all we need to do is to show that P is the projection of Q onto the subspace of the relevant variables. It then follows from Proposition 1 that P is integral.

We will illustrate the procedure on several examples.

Perfectly Matchable Subgraphs of a Bipartite Graph. Let $G = (V, E)$ be a bipartite graph with bipartition $V = V_1 \cup V_2$, let $G(W)$ denote the subgraph induced by $W \subseteq V$, and let X be the set of incidence vectors of vertex sets W such that $G(W)$ has a perfect matching.

The Perfectly Matchable Subgraph (PMS-) polytope of G is then $\text{conv}(X)$, the convex hull of X. Its linear characterization [14] can be obtained by projection as follows.

Fact (from the König-Hall Theorem)

$G(W)$ has a perfect matching if and only if

$$|W \cap V_1| = |W \cap V_2|$$

and for every $S \subseteq W \cap V_1$

$$|S| \leq |N(S)|,$$

where

$$N(S) := \{ j \in N \,|\, (i, j) \in E \text{ for some } i \in S \}.$$

Theorem 3. [14] *The PMS polytope of the bipartite graph G is defined by the system*

$$\begin{aligned} 0 \leq x_i \leq 1 \ \ & i \in V \\ x(V_1) - x(V_2) = 0 & \\ x(S) - x(N(S)) \leq 0 \ \ & S \subseteq V_1. \end{aligned} \tag{3}$$

If $0 \leq x_i \leq 1$ is replaced by $x_i \in \{0,1\}$, the theorem is simply a restatement of the Kónig-Hall condition in terms of incidence vectors. So the proof of the theorem amounts to showing that the polytope defined by (3) is integral.

Note that the coefficient matrix of (3) is not totally unimodular.

To prove the theorem, we restate the constraint set in terms of vertex and edge variables, x_i and u_{ij}, respectively, i.e., in a higher dimensional space. We then obtain the system

$$\begin{array}{lll} u(i,N(i)) - x_i = 0 & i \in V_1 \\ u(N(j),j) - x_j = 0 & j \in V_2 \\ x(V_1) - x(V_2) = 0 & \\ u_{ij} \geq 0,\ (i,j) \in E;\ 0 \leq x_i \leq 1, & i \in V \end{array} \tag{4}$$

whose coefficient matrix is totally unimodular. Here $u(i,N(i)) := \sum(u_{ij} : j \in N(i))$ and $u(N(j),j) := \sum(u_{ij} : i \in N(j))$. Thus the polyhedron defined by (4) is integral, and if it can be shown that its projection is the polyhedron defined by (3), this is proof that the latter is also integral. This is indeed the case (see [14] for a proof). Moreover, the projection yields the system (3) with the condition $S \subseteq V_1$ replaced by

$$S \subseteq V_1 \text{ such that } G(S \cup N(S)) \text{ and } G((K_1 \setminus S) \cup (K_2 \setminus N(S))) \text{ are connected}$$

(where K is the component of G containing $S \cup N(S)$, and $K_i = K \cap V_i$, $i = 1,2$).

This in turn allows one to weaken the "if" requirement of the König-Hall Theorem to "if $|S| \leq |N(S)|$ for all $S \subseteq V_1$ such that $G(S \cup N(S))$ and $G((K_1 \setminus S) \cup (K_2 \setminus N(S)))$ are connected."

Assignable Subgraphs of a Digraph. The well known *Assignment Problem* (AP) asks for assigning n people to n jobs. When represented on a digraph, an assignment, i.e. a solution to AP, consists of a collection of arcs that forms a node-disjoint union of cycles (a cycle decomposition). A digraph $G = (V,A)$ is *assignable* (admits a cycle decomposition) if the assignment problem on G has a solution. The Assignable Subgraph Polytope of a digraph is the convex hull of incidence vectors of node sets W such that $G(W)$ is assignable.

Let $\deg^+(v)$ and $\deg^-(v)$ denote the outdegree and indegree, respectively, of v and for $S \subset V$, let $\Gamma(S) := \{j \in V | (i,j) \in A \text{ for some } i \in S\}$.

Projection can again be used to prove the following

Theorem 4. **[4]** *The Assignable Subgraph Polytope of the digraph G is defined by the system*

$$\begin{array}{ll} 0 \leq x_i \leq 1 & i \in V \\ x(S \setminus \Gamma(S)) - x(\Gamma(S) \setminus S) \leq 0, & S \subseteq V. \end{array}$$

s-t Path Decomposable Subgraphs of an Acyclic Digraph. An acyclic digraph $G = (V,A)$ with two distinguished nodes, s and t, is said to admit an *s-t path decomposition* if there exists a collection of interior node disjoint s-t paths

that cover all the nodes of G. The s-t Path Decomposable Subgraph Polytope of G is then the convex hull of incidence vectors of node sets $W \subseteq V - \{s,t\}$ such that $G(W \cup \{s,t\})$ admits an s-t path decomposition.

For $S \subseteq V$, define

$$\Gamma^*(S) := \begin{cases} (\Gamma(S) \setminus \{t\}) \cup \Gamma(s) & \text{if } t \in \Gamma(S) \\ \Gamma(S) & \text{if } t \notin \Gamma(S) \end{cases}$$

Projection can then be used to prove the following

Theorem 5. **[4]** *The s-t Path Decomposable Subgraph Polytope of the acyclic digraph $G = (V, A)$ is defined by the system*

$$\begin{aligned} 0 \leq x_i \leq 1 \quad & i \in V \\ x(S \setminus \Gamma^*(S)) - (\Gamma^*(S) \setminus S) \leq 0 \quad & S \subseteq V - \{s,t\} \end{aligned}$$

Perfectly Matchable Subgraphs of an Arbitrary Graph. For an arbitrary undirected graph $G = (V, E)$, the perfectly matchable subgraph (PMS-) polytope, defined as before, can also be characterized by projection, but this is a considerably more arduous task than in the case of a bipartite graph. The difficulty partly stems from the fact that the projection cone in this case is not pointed and thus instead of the extreme rays one has to work with a finite set of generators. On the other hand, an interesting feature of the technique used in this case is that a complete set of generators did not have to be found; it was sufficient to identify a set of generators that produce all facet defining inequalities of the PMS-polytope.

For $W \subset V$, let $G(W)$ be the subgraph of G induced by W, let $c(W)$ be the number of components of $G(W)$, and let $N(W)$ be the set of neighbors of W, i.e. $N(W) := \{j \in V \setminus W) : (i,j) \in E$ for some $i \in W\}$.

Theorem 6. **[15]** *The PMS polytope of an arbitrary graph $G = (V, E)$ is defined by the system*

$$\begin{aligned} 0 \leq x_i \leq 1 \;\; i \in V \\ x(S) - x(N(S)) \leq |S| - c(S) \end{aligned} \tag{5}$$

for all $S \subseteq V$ such that every component of $G(S)$ consists of a single node or else is a nonbipartite graph with an odd number of nodes.

For further details and a proof see [15].

Exercise 5. Discuss the connection between Theorem 6 and Tutte's condition for a graph to have a perfect matching.

5 Disjunctive Programming

Disjunctive programming is optimization over unions of polyhedra. While polyhedra are convex sets, their unions of course are not. The name reflects the fact that the objects investigated by this theory can be viewed as the solution sets of systems of linear inequalities joined by the logical operations of conjunction, negation (taking of complement) and disjunction, where the nonconvexity is due to the presence of disjunctions. Pure and mixed integer programs, in particular pure and mixed 0-1 programs can be viewed as disjunctive programs; but the same is true of a host of other problems, like for instance the linear complementarity problem. It is clear that if, for instance, a linear program over a feasible set F is amended with the condition that variable x_j has to be an integer between 0 and k, which can be written as $(x_j = 0) \vee (x_j = 1) \ldots \vee (x_j = k)$ (with "$\vee$" the logical "or" symbol), then it becomes an optimization problem over a union of polyhedra $P_0 \cup P_1 \cup \ldots \cup P_k$, where $P_i := \{x \in F : x_j = i\}$ for $i = 0, 1, \ldots, k$. The main application of disjunctive programming has so far been to integer and, in particular, 0-1 programming, where it has served as the source of a rich family of cutting planes and in the early 90's has produced a computationally successful variant known as lift-and-project [7].

The foundations of disjunctive programming were laid in a July 1974 technical report, published 24 years later as an invited paper [1] with a foreword. For additional work on disjunctive programming in the seventies and eighties see [2,3,10,16,19,20,26,27,28,38]. In particular, [2] contains a detailed account of the origins of the disjunctive approach and the relationship of disjunctive cuts to Gomory's mixed integer cut, intersection cuts and others. Disjunctive programming received a new impetus in the early nineties from the work on matrix cones by Lovász and Schrijver [31], see also Sherali and Adams [37]. The version that led to the computational breakthroughs of the nineties is described in the two papers by Balas, Ceria and Cornuéjols [7,8], the first of which discusses the cutting plane theory behind the approach, while the second deals with the branch-and-cut implementation and computational testing. Related recent developments are discussed in [9,5,17,21,22,23,30,36,40,42,43].

A *disjunctive set*, i.e. the constraint set of a disjunctive program, can be expressed in many different forms, of which the following two extreme ones have special significance. Let

$$P_i := \{x \in \mathbb{R}^n : A^i x \geq b^i\}, \qquad i \in Q$$

be convex polyhedra, with Q a finite index set and (A^i, b^i) an $m_i \times (n+1)$ matrix, $i \in Q$, and let $P := \{x \in \mathbb{R}^n : Ax \geq b\}$ be the polyhedron defined by those inequalities (if any) common to all P_i, $i \in Q$. Then the disjunctive set $\bigcup_{i \in Q} P_i$ over which we wish to optimize some linear function can be expressed as

$$\{x \in \mathbb{R}^n : \bigvee_{i \in Q} (A^i x \geq b^i\}, \tag{6}$$

which is its *disjunctive normal form* (a disjunction whose terms do not contain further disjunctions). The same disjunctive set can also be expressed as

$$\{x \in \mathbb{R}^n : Ax \geq b, \bigvee_{h \in Q_j} (d^h x \geq d_0^h),\ j = 1, \ldots, t\}, \tag{7}$$

which is its *conjunctive normal form* (a conjunction whose terms do not contain further conjunctions). Here (d^h, d_0^h) is a $(n+1)$-vector for $h \in Q_j$, all j, where each set Q_j contains exactly one inequality of each system $A^i x \geq b^i$, $i \in Q$, and t is the number of all sets Q_j with this property. Thus the connection between (6) and (7) is that each term $A^i x \geq b^i$ of (6) contains $Ax \geq b$ and exactly one inequality $d^h x \geq d_0^h$ of each disjunction of (7) indexed by Q_j for $j = 1, \ldots, t$, and that all distinct systems $A^i x \geq b^i$ with this property are present among the terms of (6).

The lift-and-project approach relies mainly on the following two basic ideas (results) of disjunctive programming, the first one of which is derived from the form (6), while the second from the form (7):

1. There is a compact representation of the convex hull of a union of polyhedra in a higher dimensional space, which in turn can be projected back into the original space. The first step of this operation, i.e. the higher dimensional representation, may be viewed as lifting (or extended formulation), while the second step is projection. As a result one obtains the convex hull in the original space.

2. A large class of disjunctive sets, called *facial*, can be convexified sequentially, i.e. their convex hull can be derived by imposing the disjunctions one at a time, generating each time the convex hull of the current set.

We will discuss these two ideas in turn.

The Convex Hull of a Union of Polyhedra.

Theorem 7. **[1]** *Given polyhedra* $P_i := \{x \in \mathbb{R}^n : A^i x \geq b^i\} \neq \emptyset$, $i \in Q$, *the closed convex hull of* $\bigcup_{i \in Q} P_i$ *is the set of those* $x \in \mathbb{R}^n$ *for which there exist vectors* $(y^i, y_0^i) \in \mathbb{R}^{n+1}$, $i \in Q$, *satisfying*

$$\begin{aligned} x - \textstyle\sum(y^i : i \in Q) &= 0 \\ A^i y^i - b^i y_0^i &\geq 0 \\ y_0^i &\geq 0 \quad i \in Q \\ \textstyle\sum(y_0^i : i \in Q) &= 1. \end{aligned} \tag{8}$$

In particular, denoting by $P_Q := conv(\bigcup_{i \in Q} P_i)$ *the closed convex hull of* $\bigcup_{i \in Q} P_i$ *and by* $\mathcal{P}$ *the set of vectors* $(x, \{y^i, y_0^i\}_{i \in Q})$ *satisfying (8),*

(i) if x^* *is an extreme point of* P_Q, *then* $(\bar{x}, \{\bar{y}^i, \bar{y}_0^i\}_{i \in Q})$ *is an extreme point of* $\mathcal{P}$, *with* $\bar{x} = x^*$, $(\bar{y}^k, \bar{y}_0^k) = (x^*, 1)$ *for some* $k \in Q$, *and* $(\bar{y}^i, \bar{y}_0^i) = (0, 0)$ *for* $i \in Q \setminus \{k\}$.

(ii) if $(\bar{x}, \{\bar{y}^i, \bar{y}_0^i\}_{i \in Q})$ *is an extreme point of* $\mathcal{P}$, *then* $\bar{y}^k = \bar{x} = x^*$ *and* $\bar{y}_0^k = 1$ *for some* $k \in Q$, $(\bar{y}^i, \bar{y}_0^i) = (0, 0)$, $i \in Q \setminus \{k\}$, *and* x^* *is an extreme point of* P_Q.

Note that in this higher dimensional representation of P_Q, the number of variables and constraints is linear in the number $|Q|$ of polyhedra in the union, and so is the number of facets of $\mathcal{P}$. Note also that in any basic solution of the linear system (8), $y_0^i \in \{0,1\}$, $i \in Q$, automatically, without imposing this condition explicitly.

If we impose simultaneously *all* the integrality conditions of a mixed 0-1 program with p 0-1 variables, we have a disjunction with 2^p terms, one for every p-component 0-1 point. But if we impose only disjunctions that yield a set Q of manageable size, then this representation becomes extremely useful (such an approach is facilitated by the sequential convexifiability of facial disjunctive sets, see below).

In the special case of a disjunction of the form $x_j \in \{0,1\}$, when $|Q| = 2$ and

$$P_{j0} := \{x \in \mathbb{R}^n_+ : Ax \geq b,\, x_j = 0\},$$
$$P_{j1} := \{x \in \mathbb{R}^n_+ : Ax \geq b,\, x_j = 1\},$$

$P_Q := \text{conv}\,(P_{j0} \cup P_{j1})$ is the set of those $x \in \mathbb{R}^n$ for which there exist vectors $(y, y_0), (z, z_0) \in \mathbb{R}^{n+1}_+$ such that

$$\begin{array}{rrrrr} x - & y & - & z & = 0 \\ & Ay - by_0 & & & \geq 0 \\ & -y_j & & & = 0 \\ & & & Az - bz_0 \geq 0 & \\ & & & z_j - z_0 = 0 & \\ & & y_0 & + z_0 = 1 & \end{array} \qquad (8')$$

Unlike the general system (8), the system (8′), in which $|Q| = 2$, is of quite manageable size.

Projection and Polarity

In order to generate the convex hull P_Q, and more generally, to obtain valid inequalities (cutting planes) in the space of the original variables, we project $\mathcal{P}$ onto the x-space:

Theorem 8. **[1]** *$Proj_x(\mathcal{P}) = \{x \in \mathbb{R}^n : \alpha x \geq \beta$ for all $(\alpha, \beta) \in W_0\}$, where*

$$W_0 := \{(\alpha, \beta) \in \mathbb{R}^{n+1} : \alpha = u^i A^i,\, \beta \leq u^i b^i \text{ for some } u^i \geq 0,\, i \in Q\}.$$

□

Note that W_0 is not the standard projection cone of $\mathcal{P}$, introduced in section 1, which is (assuming each A^i is $m_i \times n$)

$$W := \{(\alpha, \beta, \{u^i\}_{i \in Q}) \in \mathbb{R}^n \times \mathbb{R} \times \mathbb{R}^{m_1} \times \ldots \times \mathbb{R}^{m_{|Q|}} : \alpha - u^i A^i = 0, \beta - u^i b^i \leq 0, u^i \geq 0, i \in Q\}.$$

Instead, $W_0 = \text{Proj}_{(\alpha,\beta)}(W)$, i.e. W_0 is the projection of W onto the (α, β)-space. In fact, W_0 can be shown to be the *reverse polar cone* P_Q^* of P_Q, i.e. the cone of all valid inequalities for P_Q.

Theorem 9. [1]

$$P_Q^* := \{(\alpha, \beta) \in \mathbb{R}^{n+1} : \alpha x \geq \beta \text{ for all } x \in P_Q\}$$

$$= \{(\alpha, \beta) \in \mathbb{R}^{n+1} : \alpha = u^i A^i,\ \beta \leq u^i b^i \text{ for some } u^i \geq 0,\ i \in Q\}.$$

□

To turn again to the special case of a disjunction of the form $x_j \in \{0, 1\}$, projecting the system (6) onto the x-space yields the polyhedron P_Q whose reverse polar cone is

$$\begin{aligned} P_Q^* = \{(\alpha, \beta) \in \mathbb{R}^{n+1} : \ & \alpha \geq uA - u_0 e_j \\ & \alpha \geq vA + v_0 e_j \\ & \beta \leq ub \\ & \beta \leq vb + v_0 \\ & u, v \geq 0\} \end{aligned}$$

(where e_j is the j-th unit vector).

One of the main advantages of the higher dimensional representation is that in projecting it onto the x-space we have a straightforward criterion to distinguish facets of P_Q from other valid inequalities:

Theorem 10. [1] *Assume P_Q is full dimensional. The inequality $\alpha x \geq \beta$ defines a facet of P_Q if and only if (α, β) is an extreme ray of the cone P_Q^*.*

Next we turn to the class of disjunctive programs that are sequentially convexifiable.

Sequential Convexification

A disjunctive set is called *facial* if every inequality in (7) induces a face of P, the polyhedron defined by the inequalities $Ax \geq b$ common to all terms of the disjunction. Zero-one programs (pure or mixed) are facial disjunctive programs, general integer programs are not. *Sequential convexifiability* is one of the basic properties that distinguish 0-1 programs from general integer programs.

Theorem 11. [1] *Let*

$$D := \{x \in \mathbb{R}^n : Ax \geq b,\ \bigvee_{h \in Q_j} (d^h x \geq d_0^h),\ j = 1, \ldots, t\},$$

where $1 \leq t \leq n$, $|Q_j| \geq 1$ for $j = 1, \ldots, t$, and D is facial. Let $P_D := conv(D)$.

Define

$$P^0(= P) := \{x \in \mathbb{R}^n : Ax \geq b\},$$

and for $j = 1, \ldots, t$,

$$P^j := conv\ (P^{j-1} \cap \{x : \bigvee_{h \in Q_j} (d^h x \geq d_0^h)\}).$$

Then

$$P^t = P_D.$$

While faciality is a sufficient condition for sequential convexifiability, it is not necessary. A necessary and sufficient condition is given in [16]. The most important class of facial disjunctive programs are mixed 0-1 programs, and for that case Theorem 11 asserts that if we denote

$$P_D := \text{conv } \{x \in \mathbb{R}^n_+ : Ax \geq b,\ x_j \in \{0,1\},\ j = 1, \ldots, p\},$$

$$P^0 := \{x \in \mathbb{R}^n_+ : Ax \geq b\},$$

and define recursively for $j = 1, \ldots, p$

$$P^j := \text{conv } (P^{j-1} \cap \{x : x_j \in \{0,1\}\}),$$

then

$$P^p = P_D.$$

Thus, in principle, a 0-1 program with p 0-1 variables can be solved in p steps. Here each step consists of imposing the 0-1 condition on one new variable and generating all the inequalities that define the convex hull of the set defined in this way.

Note that while pure and mixed 0-1 programs are facial disjunctive programs and therefore are sequentially convexifiable, general integer programs – whether pure or mixed – are not.

Disjunctive Rank

A useful concept in cutting plane theory is that of *rank* with respect to a cut generating procedure. If $Ax \geq b$ is the linear programming relaxation of some integer program, every valid inequality can be derived by (repeatedly) taking nonnegative combinations of $Ax \geq b$ and rounding down the resulting inequality, as in $\lfloor \lambda A \rfloor x \geq \lfloor \lambda b \rfloor$ with $\lambda \geq 0$. The minimum number of times this procedure has to be iterated in order to obtain a given valid inequality (or an inequality that dominates it) is known as the *Chvatal rank* of that inequality (see e.g. [33]).

Based on the sequential convexifiability of facial disjunctive programs, one can define the *disjunctive rank* of an inequality $\alpha x \geq \beta$ for a mixed 0-1 program as the smallest integer k for which there exists an ordering $\{i_1, \ldots, i_p\}$ of $\{1, \ldots, p\}$ such that $\alpha x \geq \beta$ is valid for P^k. In other words, an inequality is of rank k if it can be obtained by k, but not by fewer than k, applications of the recursive procedure defined above. Clearly, the disjunctive rank of a cutting plane for 0-1 programs is bounded by the number of 0-1 variables. It is known that the number of 0-1 variables is not a valid bound for the Chvatal rank of an inequality.

The above definition of the disjunctive rank is based on using the disjunctions $x_j \in \{0, 1\}$, $j = 1, \ldots, p$. Tighter bounds can be derived by using stronger disjunctions. For instance, a 0-1 program whose constraints include the generalized upper bounds $\sum(x_j : j \in Q_i) = 1$, $i = 1, \ldots, t$, with $|Q_i| = |Q_j| = q$, $Q_i \cap Q_j = \emptyset$, $i, j \in \{1, \ldots, t\}$, and $|\cup_{i=1}^{t} Q_i| = p$, can be solved as a disjunctive program with the disjunctions

$$\bigvee_{j \in Q_i} (x_j = 1), \qquad i = 1, \ldots, t\,(= p/q),$$

in which case the disjunctive rank of any cut is bounded by the number $t = p/q$ of *GUB* constraints.

Another Derivation of the Basic Results

The two basic ingredients of our approach, the lifting/projection technique and sequential convexification, can also be derived by the following procedure [7]. Define

$$P := \{x \in \mathbb{R}^n : \widetilde{A}x \geq \widetilde{b}\} \subseteq \mathbb{R}^n$$

and

$$P_D := \text{conv}\,(\{x \in P : x_j \in \{0, 1\}, \quad j = 1, \ldots, p\}),$$

with the inequalities $x \geq 0$ and $x_j \leq 1$, $j = 1, \ldots, p$, included in $\widetilde{A}x \geq \widetilde{b}$.

1. Select an index $j \in \{1, \ldots, p\}$. Multiply $\widetilde{A}x \geq \widetilde{b}$ with $1 - x_j$ and x_j to obtain the nonlinear system

$$\begin{aligned} (1 - x_j)(\widetilde{A}x - \widetilde{b}) &\geq 0 \\ x_j(\widetilde{A}x - \widetilde{b}) &\geq 0. \end{aligned} \tag{9}$$

2. Linearize (9) by substituting y_i for $x_i x_j$, $i = 1, \ldots, n$, $i \neq j$, and x_j for x_j^2.
3. Project the resulting polyhedron onto the x-space.

Theorem 12. **[7]** *The outcome of steps 1, 2, 3 is*

$$\textit{conv}\,(P \cap \{x : x_j \in \{0, 1\}\}).$$

Corollary 5. **[7]** *Repeating steps 1, 2, 3 for each* $j \in \{1, \ldots, p\}$ *in turn yields* P_D.

The fact that this procedure is isomorphic to the one introduced earlier can be seen by examining the outcome of step 2. In fact, the linearized system resulting from step 2 is precisely (5.3′), the higher dimensional representation of the disjunctive set defined by the constraint $x_j \in \{0, 1\}$ (see [7] for details).

The above 3-step procedure is a streamlined version of the matrix cone procedure of Lovász and Schrijver [31]. The latter involves in step 1 multiplication with $1 - x_j$ and x_j for every $j \in \{1, \ldots, p\}$ rather than just one. While obtaining P_D by the Lovász-Schrijver procedure still involves p iterations of steps 1, 2, 3, the added computational cost brings a reward: after each iteration, the coefficient matrix of the linearized system must be positive semidefinite, a condition that can be used in various ways to derive strong bounds or cuts (see [31], [9]).

Another similar procedure, due to Sherali and Adams [37], is based on multiplication with every product of the form $(\prod_{j \in J_1} x_j)(\prod_{j \in J_2} (1 - x_j))$, where J_1 and J_2 are disjoint subsets of $\{1, \ldots, p\}$ such that $|J_1 \cup J_2| = t$ for some $1 \leq t \leq p$. Linearizing the resulting nonlinear system leads to a higher dimensional polyhedron whose strength (tightness) is intermediate between P and P_D, depending on the choice of t: for $t = p$, the polyhedron becomes identical to $\mathcal{P}$ defined by the system (8) for a 0-1 polytope.

Disjunctive Cuts. Theorem 9 describes the set of valid inequalities for a union of polyhedra P_i as defined in Theorem 7. If we replace this definition by $P_i := \{x \in \mathbb{R}^n_+ : A^i x \geq b^i\}$, then the valid inequalities for $P_Q = \text{conv}(\bigcup_{i \in Q} P_i)$ are of the form $\alpha x \geq \beta$, where $(\alpha, \beta) \in \mathbb{R}^n \times \mathbb{R}$ satisfies

$$\alpha \geq u^i A^i, \ \beta \leq u^i b^i, \ i \in Q \tag{10}$$

for some $u^i \geq 0$, $i \in Q$.

Another way of stating condition (10) is

$$\alpha_j = \max_{i \in Q} u^i a^i_j, \ \beta \leq \min_{i \in Q} u^i b^i \tag{11}$$

where a^i_j is the j-th column of A^i, $i \in Q$.

Since (10) – or, alternatively, (11) – defines all valid inequalities for a disjunctive program, *every* valid cutting plane for a disjunctive program can be obtained from (11) by choosing suitable multipliers u^i, $i \in Q$. In other words, any cutting plane for a combinatorial optimization problem that can be stated as a disjunctive program, can be brought to the form (10) or (11). Once brought to this form, the cuts can be improved by appropriate changes in the multipliers u^i, $i \in Q$.

We will illustrate this on an example.

Example. Consider the mixed integer program whose constraint set is

$$\begin{aligned} x_1 &= 0.2 + 0.4(-x_3) + 1.3(-x_4) - 0.01(-x_5) + 0.07(-x_6) \\ x_2 &= 0.9 - 0.3(-x_3) + 0.4(-x_4) - 0.04(-x_5) + 0.1(-x_6) \\ x_j &\geq 0,\ j = 1, \ldots, 6,\ x_j \text{ integer},\ j = 1, \ldots, 4. \end{aligned}$$

This problem is taken from the paper [29], which also lists six cutting planes derived from the associated group problem:

$$\begin{aligned} 0.75x_3 + 0.875x_4 + 0.0125x_5 + 0.35x_6 &\geq 1, \\ 0.778x_3 + 0.444x_4 + 0.40x_5 + 0.111x_6 &\geq 1, \\ 0.333x_3 + 0.667x_4 + 0.033x_5 + 0.35x_6 &\geq 1, \\ 0.50x_3 + x_4 + 0.40x_5 + 0.25x_6 &\geq 1, \\ 0.444x_3 + 0.333x_4 + 0.055x_5 + 0.478x_6 &\geq 1, \\ 0.394x_3 + 0.636x_4 + 0.346x_5 + 0.155x_6 &\geq 1. \end{aligned}$$

The first two of these inequalities are the mixed-integer Gomory cuts derived from the row of x_1 and x_2 respectively. To show how they can be improved, we first derive them as they are. To do this, for a row of the form

$$x_i = a_{i0} + \sum_{j \in J} a_{ij}(-x_j),$$

with x_j integer-constrained for $j \in J_1 := \{1, \ldots, 4\}$, continuous for $j \in J_2 := \{5, 6\}$, one defines $f_{ij} = a_{ij} - [a_{ij}]$, $j \in J \cup \{0\}$, $\varphi_{i0} = f_{i0}$, and

$$\varphi_{ij} = \begin{cases} f_{ij}, & j \in J_1^+ = \{j \in J_1 | f_{i0} \geq f_{ij}\}, \\ f_{ij} - 1, & j \in J_1^- = \{j \in J_1 | f_{i0} < f_{ij}\}, \\ a_{ij}, & j \in J_2. \end{cases}$$

Then every x which satisfies the above equation and the integrality constraints on x_j, $j \in J_1 \cup \{i\}$, also satisfies the condition

$$y_i = \varphi_{i0} + \sum_{i \in J} \varphi_{ij}(-x_j), \quad y_i \text{ integer}.$$

For the two equations of the example, the resulting conditions are

$$\begin{aligned} y_1 &= 0.2 - 0.6(-x_3) - 0.7(-x_4) - 0.01(-x_5) + 0.07(-x_6), \quad y_1 \text{ integer}, \\ y_2 &= 0.9 + 0.7(-x_3) + 0.4(-x_4) - 0.04(-x_5) + 0.1(-x_6), \quad y_2 \text{ integer}. \end{aligned}$$

Since each y_i is integer-constrained, they have to satisfy the disjunction $y_i \leq 0 \vee y_i \geq 1$. Substituting $\varphi_{i0} + \sum_{j \in J} \varphi_{ij}(-x_j)$ for each y_i and applying the formula

(11) with multipliers $u_0^i = 1/\varphi_{i0}$ in the first term, and $u_0^i = 1/(1-\varphi_{i0})$ in the second term of each disjunction we obtain for $i = 1$ and $i = 2$ the two cuts

$$\frac{0.6}{0.8}x_3 + \frac{0.7}{0.8}x_4 + \frac{0.01}{0.8}x_5 + \frac{0.07}{0.2}x_6 \geq 1,$$

and

$$\frac{0.7}{0.9}x_3 + \frac{0.4}{0.9}x_4 + \frac{0.04}{0.1}x_5 + \frac{0.1}{0.9}x_6 \geq 1.$$

These are precisely the first two inequalities of the above list. Since all cuts discussed here are stated in the form ≥ 1, the smaller the j-th coefficient, the stronger is the cut in the direction j. We would thus like to reduce the size of the coefficients as much as possible.

Now suppose that instead of $y_1 \leq 0 \vee y_1 \geq 1$, we use the disjunction

$$\{y_1 \leq 0\} \vee \left\{ \begin{array}{l} y_1 \geq 1 \\ x_1 \geq 0 \end{array} \right\},$$

which of course is also satisfied by every feasible x.

Then, applying formula (11) with multipliers 5, 5 and 15 for $y_1 \leq 0$, $y_1 \geq 1$ and $x_1 \geq 0$ respectively, we obtain the cut whose coefficients are

$$\max\left\{\frac{5\times(-0.6)}{5\times 0.2}, \frac{5\times 0.6+15\times(-0.4)}{5\times 0.8+15\times(-0.2)}\right\} = -3,$$

$$\max\left\{\frac{5\times(-0.7)}{5\times 0.2}, \frac{5\times 0.7+15\times(-1.3)}{5\times 0.8+15\times(-0.2)}\right\} = -3.5,$$

$$\max\left\{\frac{5\times(-0.01)}{5\times 0.2}, \frac{5\times 0.01+15\times 0.01}{5\times 0.8+15\times(-0.2)}\right\} = 0.2,$$

$$\max\left\{\frac{5\times 0.07}{5\times 0.2}, \frac{5\times(-0.07)+15\times(-0.07)}{5\times 0.8+15\times(-0.2)}\right\} = 0.35$$

that is

$$-3x_3 - 3.5x_4 + 0.2x_5 + 0.35x_6 \geq 1.$$

The sum of coefficients on the left hand side has been reduced from 1.9875 to -5.95. This strengthening has been obtained by assigning a positive multiplier in (11) to the inequality $x_1 \geq 0$, which had a zero multiplier in the previous derivation.

Similarly, for the second cut, if instead of $y_2 \leq 0 \vee y_2 \geq 1$ we use the disjunction

$$\left\{ \begin{array}{l} y_2 \leq 0 \\ x_1 \geq 0 \end{array} \right\} \vee \{y_2 \geq 1\},$$

with multipliers 10, 40 and 10 for $y_2 \leq 0$, $x_1 \geq 0$ and $y_2 \geq 1$ respectively, we obtain the cut

$$-7x_2 - 4x_4 + 0.4x_5 - x_6 \geq 1.$$

Here the sum of left hand side coefficients has been reduced from 1.733 to -11.6. Again, the effect is obtained by assigning a positive multiplier to $x_1 \geq 0$, this time on the other side of the disjunction.

Exercise 6. Consider the example at the end of section 5. Why is it that to improve the first cut, we replace the disjunction $(y_1 \leq 0) \vee (y_1 \geq 1)$ by $(y_1 \leq 0) \vee \binom{y_1 \geq 1}{x_1 \geq 0}$, and to improve the second cut, we replace it by $\binom{y_1 \leq 0}{x_1 \geq 0} \vee (y_1 \geq 1)$? Why not the other way? Give a necessary condition and/or a sufficient condition for an inequality $x_i \geq 0$ (where x_i is a basic variable) to be usable in this role, i.e. to improve the cut from $y_1 \leq 0 \vee y_1 \geq 1$ when included in one of the terms of the disjunction.

6 Generating Lift-and-Project Cuts

The implementation of the disjunctive programming approach into a practical 0-1 programming algorithm had to wait until the early '90s. It required not only the choice of a specific version of disjunctive cuts, but also a judicious combination of cutting with branching, made possible in turn by the discovery of an efficient procedure for lifting cuts generated in a subspace (for instance, at a node of the search tree) to be valid in the full space (i.e. throughout the search tree).

Deepest Cuts

As mentioned earlier, if P_D is full dimensional, then facets of P_D correspond to extreme rays of the reverse polar cone P_D^*. To generate such extreme rays, for each 0-1 variable x_j that is fractional at the linear programming optimum, we solve a linear program over a normalized version of the cone P_D^* corresponding to the disjunction $x_j = 0 \vee x_j = 1$, with an objective function aimed at cutting off the linear programming optimum $\bar{x}$ by as much as possible. This "cut generating linear program" for the j-th variable is of the form

$$\begin{array}{lrcl} \min & \alpha\bar{x} - \beta & & \\ \text{s.t.} & \alpha - uA + u_0 e_j & \geq & 0 \\ & \alpha - vA - v_0 e_j & \geq & 0 \\ & -\beta + ub & = & 0 \\ & -\beta + vb + v_0 & = & 0 \\ & u, v \geq 0 & & \end{array} \qquad (\text{CGLP})_j$$

and (i) $\beta \in \{1, -1\}$, or (ii) $\sum_j |\alpha_j| \leq 1$. For details, see [7,8].

The normalization constraint (i) or (ii) has the purpose of turning the cone P_D^* into a polyhedron. Several other normalizations have been proposed later, with pro's and con's for each one, and their choice plays an important role in determining the optimum.

Solving $(\text{CGLP})_j$ yields a cut $\alpha x \geq \beta$, where

$$\alpha_k = \begin{cases} \max\{ua_k, va_k\} & k \in N \setminus \{j\} \\ \max\{ua_j - u_0, va_j + v_0\} & k = j, \end{cases} \tag{12}$$

with a_k the k-th column of A, and $\beta = \min\{ub, vb + v_0\}$. This cut maximizes the amount $\beta - \alpha\bar{x}$ by which $\bar{x}$ is cut off.

Cut Lifting

A cutting plane derived at a node of the search tree defined by a subset $F_0 \cup F_1$ of the 0-1 variables, where F_0 and F_1 index those variables fixed at 0 and 1, respectively, is valid at that node and its descendants in the tree (where the variables in $F_0 \cup F_1$ remain fixed at their values). Such a cut can in principle be made valid at other nodes of the search tree, where the variables in $F_0 \cup F_1$ are no longer fixed, by calculating appropriate values for the coefficients of these variables – a procedure called lifting and mentioned in section 1. However, calculating such coefficients is in general a daunting task, which may require the solution of an integer program for every coefficient. One important advantage of the cuts discussed here is that the multipliers u, u_0, v, v_0 obtained along with the cut vector (α, β) by solving $(\text{CGLP})_j$ can be used to calculate by closed form expressions the coefficients α_h of the variables $h \in F_0 \cup F_1$.

While this possibility of calculating efficiently the coefficients of variables absent from a given subproblem (i.e. fixed at certain values) is crucial for making it possible to generate cuts during a branch-and-bound process that are valid throughout the search tree, its significance goes well beyond this aspect. Indeed, most columns of A corresponding to nonbasic components of $\bar{x}$ typically play no role in determining the optimal solution of $(\text{CGLP})_j$ and can therefore be ignored. In other words, the cuts can be generated in a subspace involving only a subset of the variables, and then lifted to the full space. This is the procedure followed in [7,8], where the subspace used is that of the variables indexed by some $R \subset N$ such that R includes all the 0-1 variables that are fractional and all the continuous variables that are positive at the LP optimum. The lifting coefficients for the variables not in the subspace, which are all assumed w.l.o.g. to be at their lower bound, are then given by

$$\alpha_\ell := \max\{ua_\ell, va_\ell\}, \quad h \in N \setminus R$$

where u and v are the optimal vectors obtained by solving $(\text{CGLP})_j$.

These coefficients always yield a valid lifted inequality. If normalization (i) is used in $(\text{CGLP})_j$, the resulting lifted cut is exactly the same as the one that would have been obtained by applying $(\text{CGLP})_j$ to the problem in the full space. If other normalizations are used, the resulting cut may differ in some coefficients (lifting a cut does not in general have a unique outcome).

Cut Strengthening

The cut $\alpha x \geq \beta$ derived from a disjunction of the form $x_j \in \{0,1\}$ can be strengthened by using the integrality conditions on variables other than x_j, as shown in [10] (see also §7 of [2]). Indeed, if x_k is such a variable, the coefficient

$$\alpha_k := \max\{ua_k, va_k\}$$

can be replaced by

$$\alpha'_k := \min\{ua_k + u_0\lceil m_k\rceil,\ va_k - v_0\lfloor m_k\rfloor\},$$

where

$$m_k := \frac{va_k - ua_k}{u_0 + v_0}.$$

For a proof of this statement, see [2], [7] or [8]. The strengthening "works," i.e. produces an actual change in the coefficient, only if $|ua_k - va_k| \geq u_0 + v_0$. If this inequality does not hold, then either $ua_k > va_k$ and $0 \geq m_k > -1$, or $ua_k < va_k$ and $0 \leq m_k < 1$; in either case, $\alpha'_k = \alpha_k$. Furthermore, the larger the difference $|ua_k - va_k|$, the more room there is for strengthening the coefficient in question.

This strengthening procedure can also be applied to cuts derived from disjunctions other than $x_j \in \{0,1\}$, including disjunctions with more than two terms. In the latter case, however, the closed form expression for the values m_k used above has to be replaced by a procedure for calculating those values, whose complexity is linear in the number of terms in the disjunction (see [10] or [2] for details).

The Overall Cut Generating Procedure

To summarize briefly the above discussion, the actual cut generating procedure is not just "lift and project," but rather RLPLS, an acronym for

- RESTRICT the problem to a subspace defined from the LP optimum, and choose a disjunction;
- LIFT the disjunctive set to describe its convex hull in a higher dimensional space;
- PROJECT the polyhedron describing the convex hull onto the original (restricted) space, generating cuts;
- LIFT the cuts into the original full space;
- STRENGTHEN the lifted cuts.

Exercise 7. Generalize the procedure described in the subsection "Deepest cuts" of section 6 to the case where you want to use a stronger disjunction than $x_j = 0 \vee x_j = 1$, namely the one implied by imposing the 0-1 condition on two variables rather than just one: $x_i \in \{0,1\}$, $x_j \in \{0,1\}$:

(a) state the resulting (4-term) disjunction,
(b) formulate the corresponding $(\text{CGLP})_{ij}$,
(c) given an optimal solution to $(\text{CGLP})_{ij}$, state the associated cut and derive the expression corresponding to (12).

7 Branch-and-Cut and Computational Experience

No cutting plane approach known at this time can solve large, hard integer programs just by itself. Repeated cut generation tends to produce a flattening of the region of the polyhedron where the cuts are applied, as well as numerical instability which can only partly be mitigated by a tightening of the tolerance requirements. Therefore the successful use of cutting planes requires their combination with some enumerative scheme. One possibility is to generate cutting planes as long as that seems profitable, thereby creating a tighter linear programming relaxation than the one given originally, and then to solve the resulting problem by branch and bound. Another possibility, known as branch-and-cut, consists of branching and cutting intermittently; i.e., when the cut generating procedure "runs out of steam," move elsewhere in the feasible region by branching. This approach depends crucially on the ability to lift the cuts generated at different nodes of the search tree so as to make them valid everywhere.

The first successful implementation of lift-and-project for mixed 0-1 programming in a branch-and-cut framework was the MIPO (Mixed Integer Program Optimizer) code described in [8]. The procedure it implements can be outlined as follows.

Nodes of the search tree (subproblems created by branching) are stored along with their optimal LP bases and associated bounds. Cuts that are generated are stored in a pool. The cut generation itself involves the RLPLS process described earlier. Furthermore, cuts are not generated at every node, but at every k-th node, where k is a cutting frequency parameter.

At any given iteration, a subproblem with best (weakest) lower bound is retrieved from storage and its optimal LP solution $\bar{x}$ is recreated. Next, all those cuts in the pool that are tight for $\bar{x}$ or violated by it, are added to the constraints and $\bar{x}$ is updated by reoptimizing the LP.

At this point, a choice is made between generating cuts or skipping that step and going instead to branching. The frequency of cutting is dictated by the parameter k that is problem dependent, and calculated after generating cuts at the root node. Its value is a function of several variables believed to characterize the usefulness of cutting planes for the given problem, primarily the average depth of the cuts obtained. In our experiments, the most frequently used value of k was 8.

If cuts are to be generated, this happens according to the RLPLS scheme described above. First, a subspace is chosen by retaining all the 0-1 variables fractional and all the continuous variables positive at the LP optimum, and removing the remaining variables along with their lower and upper bounds. A lift and project cut is then generated from each disjunction $x_j \in \{0, 1\}$ for

j such that $0 < \bar{x}_j < 1$ (this is called a round of cuts). Each cut is lifted and strengthened; and if it differs from earlier cuts sufficiently (the difference between cuts is measured by the angle between their normals), it is added to the pool; otherwise it is thrown away. After generating a round of cuts, the current LP is reoptimized again.

If cuts are not to be generated (or have already been generated), a fractional variable is chosen for branching on a disjunction of the form $x_j \in \{0, 1\}$; i.e., two new subproblems are created, their lower bounds are calculated, and they are stored. The branching variable is chosen as the one with largest (in absolute value) cost coefficient among those whose LP optimal value is closest to 0.5.

Computational Results in Branch-and-Cut Mode

The above described procedure was implemented in the code MIPO, described in detail in [8]. This implementation of MIPO does not have its own linear programming routine; instead, it calls a simplex code whenever it has to solve or reoptimize an LP. In the experiments of [8] the LP solver used was that of CPLEX 2.1. The test bed consisted of 29 test problems from MIPLIB and other sources, ranging in size from about 30 to 9,000 0-1 variables, 0 to 500 continuous variables, and about 20 to 2,000 constraints. The large majority of these problems have a real world origin; they were contributed mostly by people who tried, not always successfully, to solve them. Of the MIPLIB problems, most of those not included into the testbed were omitted as too easily solved by straight branch-and-bound; two problems were excluded because their LP relaxation exceeded the dimensioning of the experiment. MIPO was compared with MINTO, OSL and CPLEXMIP 2.1 (the most advanced version available at the time of the experiments). The outcome (see [8] for detailed results) is best summarized by showing the number of times a code ranked first, and second, both in terms of search tree nodes and in terms of computing time. This is done in Table 1, whose two parts correspond to Tables 9 and 10 of [8].

Table 1.

Ranking by number of search tree nodes:

OSL		CPLEX		MINTO		MIPO	
First	Second	First	Second	First	Second	First	Second
15	2	0	4	4	10	11	10

Ranking by CPU time:

OSL		CPLEX		MINTO		MIPO	
First	Second	First	Second	First	Second	First	Second
2	3	6	6	10	3	12	14

In a sense, MIPO seemed to be the most robust among the four codes: it was the only one that managed to solve all 29 test problems, and it ranked first or second in computing time on 26 out of the 29 instances.

Other experiments with lift-and-project in an enumerative framework are reported on in [41], where S. Thienel compares the performance of ABACUS, an object oriented branch-and-cut code, in two different modes of operation, one using lift-and-project cuts and the other using Gomory cuts; with the outcome that the version with lift-and-project cuts is considerably faster on all hard problems, where hard means requiring at least 10 minutes.

Little experimentation has taken place so far with cuts derived from stronger disjunctions than the 0-1 condition on a single variable. In [9] the MIPO procedure was run on maximum clique problems, where the higher dimensional formulation used to generate cuts was the one obtained by multiplying the constraint set with inequalities of the form $1 - \sum(x_j : j \in S) \geq 0$, $x_j \geq 0$, $j \in S$, where S is a stable set. This is the same as the higher dimensional formulation derived from the disjunction

$$(x_j = 0,\, j \in S) \vee (x_{j_1} = 1,\, x_j = 0,\, j \in S \setminus \{j_1\}) \vee \ldots \\ \ldots \vee (x_{j_s} = 1,\, x_j = 0,\, j \in S \setminus \{j_s\}),$$

where $s = |S|$. As this disjunction is more powerful than the standard one, the cuts obtained were stronger; but they were also more expensive to generate, and without some specialized code to solve the highly structured cut generating LP's of this formulation, the trade-off between the strength of the cuts and the cost of generating them favored the weaker cuts from the standard disjunction.

Results in Cut-and-Branch Mode

Bixby, Cook, Cox and Lee [18] report on their computational experience with a parallel branch-and-bound code, run on 51 test problems after generating several types of cuts at the root node. One of the cut types used was disjunctive or lift-and-project cuts, generated essentially as in [8] with normalization (ii), but without restriction to a subspace and without strengthening. Since deriving these cuts in the full space is expensive, the routine generating them was activated only for 4 of the hardest problems. Their addition to the problem constraints reduced the integrality gap by 58.7%, 94.4%, 99.9% and 94.4%, respectively.

In [21], Ceria and Pataki report computational results with a disjunctive cut generator used in tandem with the CPLEX branch and bound code. Namely, the cut generator was used to produce 2 and 5 rounds of cuts from the 0-1 disjunctions for the 50 most promising variables fractional at the LP optimum, after which the resulting problem with the tightened LP relaxation was solved by the CPLEX 5.0 MIP code. This "cut-and-branch" procedure was tested on 18 of the hardest MIPLIB problems and the results were compared to those obtained by using CPLEX 5.0 without the cut generator. The outcome of the comparison can be summarized as follows (see Table 2 of [21] for details).

- The total running time of the cut-and-branch procedure was less than the time without cuts for 14 of the 18 problems; while the opposite happened for the remaining 4 problems.
- Two of the problems solved by cut-and-branch in 8 and 3 minutes respectively could not be solved by CPLEX alone in 20 hours.
- For 6 problems the gain in time was more than 4-fold.

Very good results were obtained on two difficult problems outside the above set. One of them, *set1ch*, could not be solved by CPLEX alone, which after exhausting its memory limitations stopped with a solution about 15% away from the optimum. On the other hand, running the cutting plane generator for 10 rounds on this problem produced a lower bound within 1.4% of the integer optimum, and running CPLEX on the resulting tightened formulation solved the problem to optimality in 28 seconds.

The second difficult problem, *seymour*, was formulated by Paul Seymour in an attempt to find a minimal irreducible configuration in the proof of the four color theorem. It was not solved to optimality until very recently. The value of the LP relaxation is 403.84, and an integer solution of 423.00 was known. The best previously known lower bound of 412.76 had been obtained by running a parallel computer with 16 processors for about 60 hours, using about 1400 megabytes of memory. The cut-and-branch procedure applied to this case generated 10 rounds of 50 cuts in about 10.5 hours, and produced a lower bound of 413.16, using less than 50 megabytes of memory. Running CPLEX for another 10 hours on this tightened formulation then raised the bound to 414.20. More recently, using the same lift-and-project cuts but a more powerful computer, the problem was solved to optimality [35].

References

1. E. Balas, "Disjunctive Programming: Properties of the Convex Hull of Feasible Points." Invited paper, with a foreword by G. Cornuéjols and W. Pulleyblank, *Discrete Applied Mathematics, 89,* 1998, 3-44 (originally MSRR# 348, Carnegie Mellon University, July 1974).
2. E. Balas, "Disjunctive Programming." *Annals of Discrete Mathematics, 5,* 1979, 3-51.
3. E. Balas, "Disjunctive Programming and a Hierarchy of Relaxations for Discrete Optimization Problems." *SIAM Journal on Algebraic and Discrete Methods, 6*, 1985, 466-485.
4. E. Balas, "The Assignable Subgraph Polytope of a Directed Graph." *Congressus Numerantium, 60,* 1987, 35-42.
5. E. Balas, "A Modified Lift-and-Project Procedure." *Mathematical Programming, 79,* 1997, 19-31.
6. E. Balas, "Projection with a Minimal System of Inequalities." *Computational Optimization and Applications, 10,* 1998, 189-193.
7. E. Balas, S. Ceria and G. Cornuéjols, "A Lift-and-Project Cutting Plane Algorithm for Mixed 0-1 Programs." *Mathematical Programming, 58,* 1993, 295-324.

8. E. Balas, S. Ceria and G. Cornuéjols, "Mixed 0-1 Programming by Lift-and-Project in a Branch-and-Cut Framework." *Management Science, 42,* 1996, 1229-1246.
9. E. Balas, S. Ceria, G. Cornuéjols and G. Pataki, "Polyhedral Methods for the Maximum Clique Problem." In D. Johnson and M. Trick (editors), *Clique, Coloring and Satisfiability: The Second DIMACS Challenge.* The American Mathematical Society, Providence, RI, 1996, 11-27.
10. E. Balas and R.G. Jeroslow, "Strengthening Cuts for Mixed Integer Programs." *European Journal of Operational Research, 4,* 1980, 224-234.
11. E. Balas and J.B. Mazzola, "Nonlinear 0-1 Programming: I. Linearization Techniques, II. Dominance Relations and Algorithms." *Mathematical Programming, 30,* 1984, 1-21 and 22-45.
12. E. Balas and M. Oosten, "On the Dimension of Projected Polyhedra." *Discrete Applied Mathematics, 87,* 1998, 1-9.
13. E. Balas and M. Oosten, "The Cycle Polytope of a Directed Graph." *Networks, 36,* 2000, 34-46.
14. E. Balas and W.R. Pulleyblank, "The Perfectly Matchable Subgraph Polytope of a Bipartite Graph." *Networks, 13,* 1983, 495-518.
15. E. Balas and W.R. Pulleyblank, "The Perfectly Matchable Subgraph Polytope of an Arbitrary Graph." *Combinatorica, 9,* 1989, 321-337.
16. E. Balas, J. Tama and J. Tind, "Sequential Convexification in Reverse Convex and Disjunctive Programming." *Mathematical Programming, 44,* 1989, 337-350.
17. N. Beaumont, "An Algorithm for Disjunctive Programming." *European Journal of Operational Research, 48,* 1990, 362-371.
18. R. Bixby, W. Cook, A. Cox and E. Lee, "Computational Experience with Parallel Mixed Integer Programming in a Distributed Environment." *Annals of Operations Research, 90,* 1999, 19-45.
19. C. Blair, "Two Rules for Deducing Valid Inequalities for 0-1 Programs." *SIAM Journal of Applied Mathematics, 31,* 1976, 614-617.
20. C. Blair, "Facial Disjunctive Programs and Sequences of Cutting Planes." *Discrete Applied Mathematics, 2,* 1980, 173-180.
21. S. Ceria and G. Pataki, "Solving Integer and Disjunctive Programs by Lift-and-Project." In R.E. Bixby, E.A. Boyd and R.Z. Rios-Mercado (editors), *IPCO VI, Lecture Notes in Computer Science, 1412,* Springer 1998, 271-283.
22. S. Ceria and J. Soares, "Disjunctive Cuts for Mixed 0-1 Programming: Duality and Lifting." GSB, Columbia University, 1997.
23. S. Ceria and J. Soares, "Convex Programming for Disjunctive Convex Optimization." *Mathematical Programming, 86,* 1999, 595-614.
24. G.B. Dantzig, R.D. Fulkerson and S.M. Johnson, "Solution of a Large-Scale Traveling Salesman Problem." *Operations Research, 2*, 1954, 393-410.
25. R. Fortet, "Applications de l'algèbre de Boolean recherche opérationnelle." *Revue Francaise de Recherche Opérationnelle, 4*, 1960, 17-25.
26. R.G. Jeroslow, "Cutting Plane Theory: Disjunctive Methods." *Annals of Discrete Mathematics, 1,* 1977, 293-330.
27. R.G. Jeroslow, "Representability in Mixed Integer Programming I: Characterization Results." *Discrete Applied Mathematics, 17,* 1987, 223-243.
28. R.G. Jeroslow, "Logic Based Decision Support: Mixed Integer Model Formulation." *Annals of Discrete Mathematics, 40,* 1989.
29. E.L. Johnson, "The Group Problem for Mixed-Integer Programming." *Mathematical Programming Study, 2,* 1974, 137-179.
30. A. Letchford, "On Disjunctive Cuts for Combinatorial Optimization." *Journal of Combinatorial Optimization, 5*, 2001 (Issue 3).

31. L. Lovász and A. Schrijver, "Cones of Matrices and Set Functions and 0-1 Optimization." *SIAM Journal of Optimization, 1,* 1991, 166-190.
32. C.E. Miller, A.W. Tucker and R.A. Zemlin, "Integer Programming Formulations and Traveling Salesman Problems." *Journal of the ACM, 7*, 1960, 326-329.
33. G. Nemhauser and L. Wolsey, *Integer and Combinatorial Optimization*, Wiley, 1986.
34. M. Padberg and Ting-Yi Sung, "An Analytical Comparison of Different Formulations of the Traveling Salesman Problem." *Math. Programming, 52,* 1991, 315-357.
35. G. Pataki, personal communication, March 2001.
36. M. Perregaard and E. Balas, "Generating Cuts from Multiple-Term Disjunctions." In K. Aardal and B. Gerards (Editors), *Integer Programming and Combinatorial Optimization* (8th IPCO Conference, Utrecht, 2001), LCNS No. 2081, Springer, 2001.
37. H. Sherali and W. Adams, "A Hierarchy of Relaxations Between the Continuous and Convex Hull Representations for Zero-One Programming Problems." *SIAM Journal on Discrete Mathematics, 3,* 1990, 411-430.
38. H. Sherali and C. Shetty, *Optimization with Disjunctive Constraints.* Lecture Notes in Economics and Mathematical Systems, 181, Springer, 1980.
39. J. Stoer and C. Witzgall, *Convexity and Optimization in Finite Dimensions I.* Springer, 1970.
40. R.A. Stubbs and S. Mehrotra, "A Branch-and-Cut Method for 0-1 Mixed Convex Programming." Department of Industrial Engineering, Northwestern University, 1996.
41. S. Thienel, "ABACUS: A Branch-and-Cut System." Doctoral Dissertation, Faculty of Mathematics and The Natural Sciences, University of Cologne, 1995.
42. M. Turkay and I.E. Grossmann, "Disjunctive Programming Techniques for the Optimization of Process Systems with Discontinuous Investment Costs-Multiple Size Regions." *Industrial Engineering Chemical Research, 35,* 1996, 2611-2623.
43. H.P. Williams, "An Alternative Explanation of Disjunctive Formulations." *European Journal of Operational Research, 72,* 1994, 200-203.

Mathematical Programming Models and Formulations for Deterministic Production Planning Problems*

Yves Pochet

CORE and IAG, Université Catholique de Louvain, Belgium
ypochet@core.ucl.ac.be

Abstract. We study in this lecture the literature on mixed integer programming models and formulations for a specific problem class, namely deterministic production planning problems. The objective is to present the classical optimization approaches used, and the known models, for dealing with such management problems.
We describe first production planning models in the general context of manufacturing planning and control systems, and explain in which sense most optimization solution approaches are based on the decomposition of the problem into single-item subproblems.
Then we study in detail the reformulations for the core or simplest subproblem in production planning, the single-item uncapacitated lot-sizing problem, and some of its variants. Such reformulations are either obtained by adding variables – to obtain so called extended reformulations – or by adding constraints to the initial formulation. This typically allows one to obtain a linear description of the convex hull of the feasible solutions of the subproblem. Such tight reformulations for the subproblems play an important role in solving the original planning problem to optimality.
We then review two important classes of extensions for the production planning models, capacitated models and multi-stage or multi-level models. For each, we describe the classical modeling approaches used.
Finally, we conclude by giving our personal view on some new directions to be investigated in modeling production planning problems. These include better models for capacity utilization and setup times, new models to represent the product structure – or recipes – in process industries, and the study of continuous time planning and scheduling models as opposed to the discrete time models studied in this review.

1 Introduction

Two chapters in this book present the general theory of mixed integer programming [⟶ Martin], and the lift-and-project approach to combinatorial op-

* Research carried out with financial support of the project TMR-DONET nr. ERB FMRX-CT98-0202 of the European Union, and within the framework of the Belgian Program on Interuniversity Poles of Attraction initiated by the Belgian State, Prime Minister's Office, Science Policy Programming.

M. Jünger and D. Naddef (Eds.): Computat. Comb. Optimization, LNCS 2241, pp. 57–111, 2001.
© Springer-Verlag Berlin Heidelberg 2001

timization [⟶ Balas]. They propose and study methods for improving the formulations and solving general mixed integer programs, without considering or exploiting the specific nature of the problems. In this chapter we take a different viewpoint. We study the literature on mixed integer programming models and formulations for a specific problem class, namely deterministic production planning problems. We illustrate how the specific structure of these problems can be exploited to obtain good or better formulations and algorithms for solving them. Implementation issues are not considered in this chapter, we refer the reader to two chapters [⟶ Elf/Gutwenger/Jünger/Rinaldi], and [⟶ Ladányi/Ralphs/Trotter].

In manufacturing environments, production planning problems deal with decisions about the size of the production lots of the different products to manufacture or to process, about the time at which those lots have to be produced, and sometimes about the machine or production facility where the production must take place. The production decisions are typically taken by looking at the best trade-off between financial and customer service or satisfaction objectives. In production planning and operations management, the financial objectives are usually represented by production costs – for machines, materials, manpower, startup costs, overhead costs, ... – and inventory costs – opportunity costs of the capital tied up in the stocks, insurances, ... –. Customer service objectives are represented by the ability to deliver the right product, in ordered quantity, at the promised date and place. The final aim of such modelling approaches is to provide tools allowing one to better plan and control the flow of materials and information within the firms.

These production planning problems abound in practice and the literature contains many heuristic and exact optimization algorithmic approaches to solve them. Our objective here is to present a classification of the mixed integer programming (MIP) optimization models, and mathematical formulations, used for dealing with such management problems.

In Section 2, we describe production planning models in the general context of manufacturing planning and control systems, including Material requirements planning (MRP-I), Manufacturing resource planning (MRP-II) and Hierarchical production planning (HPP). We describe briefly the elements of production planning models, and define a typology of the models encountered in practice. This part is a self contained introduction to production planning models, and to make the presentation more practical at this early stage, we describe example planning models such as the uncapacitated lot-sizing model, the multi-item master production scheduling (MPS) model and the materials requirements planning basic model.

Although practical models are multi-item, we explain in which sense most optimization solution approaches are based on the decomposition of the problem into single-item subproblems. This motivates the analysis of the formulations of a variety of single-item models in the next section. The goal is to obtain a complete linear description of the convex hull of the feasible solutions of the

subproblem. Such tight formulations play an important role in solving or finding good solutions for the original multi-item planning problem.

In Section 3, we present known results on the MIP formulation of the basic or core planning subproblem, namely the single-item uncapacitated lot-sizing subproblem (ULS), and some of its single-item variants including startup costs, backlogging, constant capacity restriction, Wagner-Within costs and profit maximization objective function. We illustrate on the ULS problem the classical approach used to study the reformulations of these polynomially solvable single item problems. The reformulations are either obtained by adding variables – to obtain so called extended reformulations – or by adding constraints to the initial formulation.

For the ULS problem, several extended formulations are described, including the shortest path formulation that can be derived directly from the polynomial time dynamic program solving the problem. Other classical extended reformulations such as multicommodity and facility location reformulations are presented.

The alternative approach is to work with the initial set of variables, and derive valid inequalities that describe the convex hull of the incidence vectors of feasible solutions. By solving the separation problem associated with such inequalities, it is possible to develop a branch and cut algorithm for solving these problems. For each model studied in this section, we describe the known valid inequalities, indicate whether or not they describe the convex hull of solutions.

In Sections 4 and 5, we review two important classes of extensions for the production planning models, capacitated models and multi-stage or multi-level models.

We separate capacitated planning models into big buckets (large time periods) and small buckets (small time periods) models. Big buckets capacitated models are used in order to control the capacity utilization in each period. We describe how to obtain classes of valid inequalities to strengthen the formulation of these models. Small buckets models are used when, in order to model accurately the capacity utilization, one has to control the production sequence of the different items – because there are variable sequence dependent change over costs and times –. We describe several formulations that are used to represent these production sequences using small buckets.

For multi-stage models, we explain the classical stage by stage decomposition used in MRP-I, and describe the well known echelon stock reformulation which is the basis of all MIP approaches for these problems.

In Section 6, we give our personal view on some new directions to be investigated in modeling production planning problems. These include better models for capacity utilization and setup times, new models to represent the product structure – or recipes – in process industries, and the study of continuous time planning and scheduling models as opposed to the discrete time models studied in this review.

We conclude with some challenges for the future of this field of research.

2 Production Planning

2.1 Manufacturing Planning and Control Systems

The production process can be defined as the transformation process of raw materials into end products, usually through a series of transformation steps producing and consuming intermediate products. These raw materials, intermediate and end products can often be inventoried, allowing one to produce and consume them at different moments and rates in time. Each transformation step may require several input products and may produce one or several outputs. The raw materials are purchased from suppliers, and the end products are sold to external customers. Sometimes, intermediate products are also sold to customers (spare parts, ...). This general definition of the production as a transformation process is illustrated in Figure 1, where materials inventories are represented by triangles, transformation processes are represented by circles, and flows of materials through the process (i.e. inputs into or outputs from transformations steps) are represented by arrows.

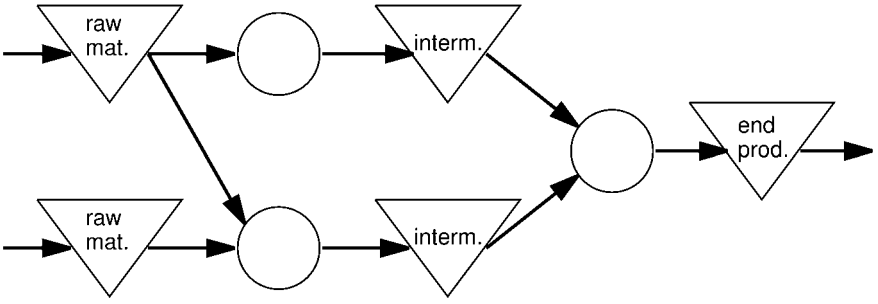


Fig. 1. The production process

Production planning is defined as the planning (that is the acquisition, time of usage, quantity used, ...) of the resources required to perform these transformation steps, in order to satisfy the customers in the most efficient or economical way. In other words, the production decisions are typically taken by looking at the best trade-off between financial objectives and customer service or satisfaction objectives. In production planning and operations management, the financial objectives are usually represented by production costs – for machines, materials, manpower, startup costs, overhead costs, ...– and inventory costs – opportunity costs of the capital tied up in the stocks, insurances, ...–. Customer service objectives are represented by the ability to deliver the right product, in ordered quantity, at the promised date and place.

According to Anthony [5] and Salomon [62] –among others–, the production planning problems can be classified into strategic, tactical and operational planning problems.

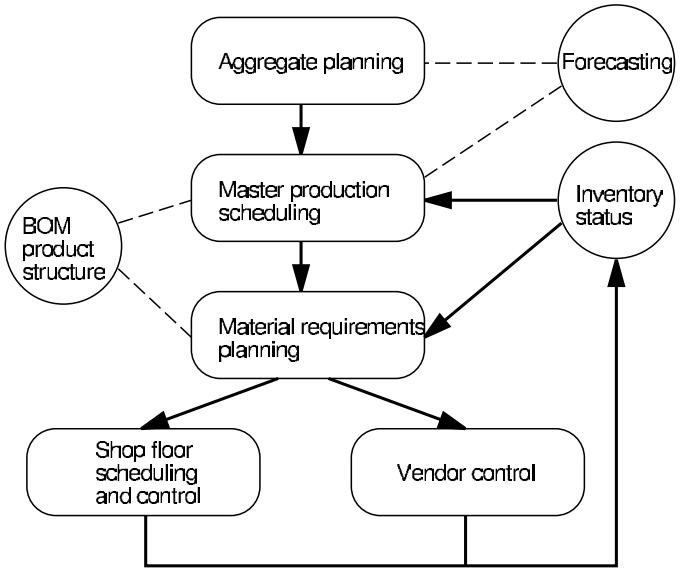


Fig. 2. An MRP-II system (adapted from [76])

Strategic problems deal with the management of change in the production process and the acquisition of the resources needed to produce. This includes for example product-mix, plan layout as well as location, supply chain design and investment decisions. The objective pursued in solving these strategic problems is to maintain the competitive advantage and capabilities, and to sustain the growth rate. This requires to model the decisions over a long term horizon using aggregate information.

Tactical planning problems analyse the resource utilization problems over a medium term planning horizon using aggregate information. This consists in making decisions about, for instance, materials flow, inventory, capacity utilization, maintenance planning. The usual objective at this stage is to improve the cost efficiency and the customer's satisfaction.

Operational Planning problems aim at planning and controling the execution of the production tasks. For instance, production sequencing and input/output analysis models fit into this category. Here the goal is to obtain an efficient and accurate execution of the plans, over a very short term horizon, and using very detailed information.

Manufacturing planning and control systems (MPC) (see Vollman, Berry, Whybark [76]) are developed to cope with these complex planning environments, and integrate these multi-level multi-horizon planning problems into a single integrated management system.

For instance, Figure 2 describes how the tactical and operational planning problems are integrated in classical manufacturing resources planning (MRP-II)

systems, an example of MPC system. In these systems, medium term production planning (PP) consists in deciding about capacity utilization and aggregate or global inventory levels to meet forecasted demand over a medium term horizon of about one year. A medium term horizon is needed to be able to take into account some seasonal pattern in the demand. Master production scheduling (MPS) consists in planning the detailed short term production of end-products in order to meet forecasted demand and firm customer orders, taking into account the capacity utilization and global inventory levels decided at the PP stage. Here the time horizon is usually expressed in weeks and corresponds to the duration of the production cycle. Materials requirements planning (MRP-I) establishes the short term production plans for all components (intermediate products and raw materials) from the production plan of end-products decided at the MPS stage, and from the product structure database (Bills of Materials, BOM). Then, shop-floor control systems (for manufactured components) and vendor follow-up systems (for purchased components) control the very short term execution of the plans decided at the MRP-I stage. The time horizon at this last stage is usually of a few days.

Other well known integrated production planning concepts and systems fit into this general manufacturing planning and control framework. For instance the MRP-II system represented in Figure 2 subsumes the original MRP-I system defined by Orlicky [49], and follows the hierarchical production planning (HPP) principles defined by Hax and Meal [29].

In these systems, the production decisions are taken, one after the other, using a sequential (decomposition) approach. For instance, in the classical MRP-II systems, the production plans of the end-items are decided first at the MPS level, without considering the capacity or inventory available for intermediate products and raw materials. Then a tentative production plan for intermediate items is decided assuming that the production capacity is infinite (or at least does not impose any constraint on the production plans). Finally, the available capacity is taken into account, and the production plans are modified (smoothed) by finite loading heuristics.

This sequential approach is suboptimal because it is not capable to integrate some important constraints in a global model. The complicating factors and constraints that would need to be integrated are: multilevel product structures (raw materials, intermediate products and end products), capacity constraints (and in particular the presence of setup times), sequencing aspects that have to be taken into account because they affect the capacity utilization.

The purpose of this lecture is precisely to describe and study more global production planning models addressing these complicating factors. The final aim of such modelling approaches is to provide tools allowing one to better plan and control the flow of materials and information within the firms.

2.2 Production Planning Models

We describe in this Subsection the main modelling elements defining production planning models, and we provide examples of mixed integer programming formulations corresponding to the MRP-I, MPS and integrated MPS/MRP decision problems.

Modelling Elements. There are a number of modelling elements present in many or most production planning problems.

- Production planning problems deal with sizing and timing decisions for production lots,
- defining the availability of resources (machine hours, workforce, subcontracting, ...),
- allocating the resources to the production lots,
- meeting forecasted demand (in a make to stosk environment) and/or customer orders (in a make to order environment)
- and maximizing performance, expressed in terms of production costs, inventory costs, and customer service level.
- over a finite planning horizon.

There are also some complicating modelling elements that are not present in all models.

- Multiple items interacting through shared resources.
  In this case, there is no material flow between the items, but there are capacity restrictions linking the items and coming from the shared resources. A typical example of such a model is the MPS model described in Subsection 2.2.
- Multiple items interacting through multi-level product structures.
  In the case of multi-level product structures, there are additional material flow restrictions because a product can be an input of some production stage and also an output of some other production stage, or it may be delivered from an external supplier. This creates some precedence constraints between the supply and the consumption of that product. These restrictions are usually modelled through inventory balance constraints. A typical example of such a model is the MPS/MRP intergated model described in Subsection 2.2.
- Demand backlogging.
  In this case, it is possible – but penalized because it has a negative impact on customer satisfaction – to deliver a customer later than required. This occurs for example when a factory does not have enough capacity to deliver all customers on time. The single item model with demand backlogging will be studied in Section 3.5.

– Startup or switching capacity utilization.
  In many cases, it is necessary to model accurately the capacity utilization in order to obtain feasible production plans. This requires sometimes to model the capacity consumed when a machine starts a production batch, or when a machine switches from one product to another. In these cases, we obtain setup or startup times models, change over times models, or models with sequencing restrictions. These models will be described in Section 4.
  In some other cases, the model are too complex to be solved with such setup or startup times restrictions, and usually simpler models involving setup or startup costs are considered. Such models can be seen as obtained by relaxing the setup or startup time restrictions. Models with setup and startup costs will be studied respectively in Sections 3 and 3.5.

Models Classification. The classification of deterministic production planning models given in Table 1 is adapted from Kuik et al [35]. In each model class, we cite seminal or important work related to the optimization of some corresponding models. Some more references will be given in the text. We refer to the above paper for a more comprehensive classification of the related literature.

The models are classified along three criteria: capacitated or uncapacitated, constant or variable demand, single or multiple item.

Uncapacitated Lot-Sizing Model. The first example model is the uncapacitated single item, single level, uncapacitated lot-sizing problem that we will study in detail in Section 3. This model is the core subproblem in production planning because it is the problem solved repeatedly for each item (from end products to raw materials) in the material requirements (MRP-I) sequential planning system.

We define the index $t = 1, \cdots, n$ to represent the discrete time periods, and n is the final period at the end of the planning horizon. The purpose is to plan the production over the planning horizon (i.e. fix the lot size in each period) in order to satisfy demand, and to minimize the sum of production and inventory costs. Classically, the production costs exhibit some economies of scale that are are modelled through a fixed charge cost function. That is, the production cost of a lot in decomposed into a fixed cost independent of the lot size, and a unit cost incurred for each unit produced in the lot. The inventory costs are modelled by charging an inventory cost per unit held in inventory at the end of each period. Any demand in a period can be satisfied by production or inventory, and backlogging is not allowed. The production capacity in each period is not considered in the model, and therefore assumed to be infinite.

For each period $t = 1, \cdots, n$, the decision variables are x_t , y_t and s_t . They represent respectively the production lot size in period t, the binary variable indicating whether or not there is a positive production in period t ($y_t = 1$ if

Table 1. Production planning models

		Uncapacitated (fixed lead times)	Capacitated (variable lead times)
Stationary or Constant demand	Single Item	Inventory control EOQ (raw material) EPQ (production) [Harris [28], Wilson [80]]	
Stationary or Constant demand	Multi Item	Serial lot sizing Assembly lot sizing (multi-level) [Crowston, W., W. [15]]	Economic lot scheduling (single-level) (shared resources) [Elmaghraby [18]]
Dynamic demand	Single Item	Uncapacitated lot sizing (ULS) [Wagner, Whitin [78]]	Capacitated lot sizing (CLS) [Florian, Klein [23]]
Dynamic demand	Multi Item	Serial lot sizing Assembly lot sizing General lot sizing (multi-level) [Veinott [75]] [Zangwill [85]] [Love [41]] [Crowston, W., [14]] [Afentakis, G., K. [1]] [Afentakis, G., [2]]	Big buckets Small buckets Discrete LS (single-level) (shared resources) [Eppen, Martin, [19]] [Trigeiro, [68] [Karmarkar, Schrage, [31] [Lasdon, Terjung, [36]] [Fleischmann, [21], [22]] [Salomon, K., K., V.W., [63]]

$x_t > 0$), and the inventory at the end of period t. The data are p_t, f_t, h_t and d_t modelling respectively, and for each period t, the unit production cost, the fixed production cost, the unit inventory cost, and the demand to be satisfied. For simplicity we suppose that $d_t \geq 0$ for all periods t.

The natural formulation of this uncapacitated lot-sizing problem (ULS) can be written as follows.

$$\min \sum_{t=1}^{n} (p_t x_t + f_t y_t + h_t s_t) \tag{1}$$

$$s_{t-1} + x_t = d_t + s_t \quad \text{for all } t \tag{2}$$

$$s_0 = s_n = 0 \tag{3}$$

$$x_t \leq M y_t \quad \text{for all } t \tag{4}$$

$$x_t, s_t \geq 0,\ y_t \in \{0, 1\} \quad \text{for all } t \tag{5}$$

where M is a large positive number. Constraint (2) expresses the demand satisfaction in each period, and is called the flow balance or flow conservation constraint. This is because every feasible solution of ULS corresponds to a flow in the network shown in Figure 3, where $d_{14} = \sum_{i=1}^{4} d_i$ is the total demand. Constraint (3) says there is no initial and no final inventory. Constraint (4) forces the setup variable in period t to be 1 when there is positive production (i.e. $x_t > 0$) in period t. Constraint (5) imposes the nonnegativity and binary restrictions on the variables. The objective function defined by (1) is simply the sum of unit production, fixed production and unit inventory costs.

The set of feasible solutions to (2)-(5) is called X^{ULS}.

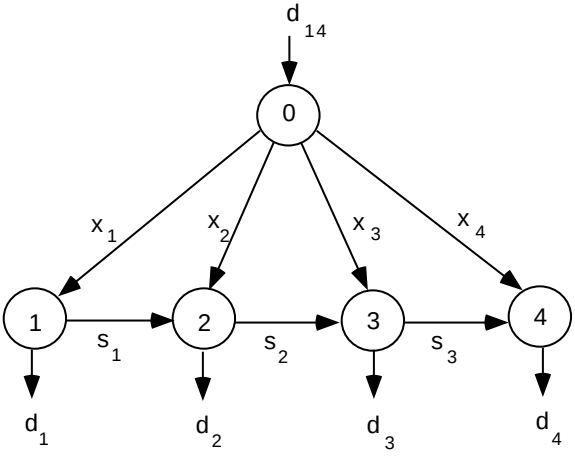


Fig. 3. Uncapacitated lot-sizing network $(n = 4)$

Master Production Scheduling Model. The next example is known as the multi item (single level) capacitated lot-sizing model. This model corresponds typically to the problem solved at the master production scheduling level in an MPC system. The purpose is to plan the production of a set of items, usually finished products, over a short term horizon corresponding to the total production cycle of these items. For each item, the model is the same as the ULS model in terms of costs and demand satisfaction. In addition, the production plans of the different items are linked through capacity restrictions coming from the common resources used to produce the items.

We define the indices $i = 1, \cdots, I$ to represent the set of items whose production has to be planned, $k = 1, \cdots, K$ to represent the set of shared resources with limited capacity, and $t = 1, \cdots, n$ to represent the time periods. The variables

x, y, s, and the data p, f, h, d, have the same meaning for each item i as in the model ULS. A superscript i has been added to represent the item i for which they are each defined.

The data L_t^k represents the available capacity of resource k during period t. The data α^{ik} and β^{ik} represent the amount of capacity of resource k consumed respectively per unit of item i produced, and for a setup of item i. The coefficient β^{ik} is often called the setup time of item i on resource k, and represents the time spent to prepare the resource k just before the production of a lot of item i. Together with α^{ik}, it may also be used to represent some economies of scale in the productivity factor of item i on resource k.

The natural formulation of this multi item capacitated lot-sizing model, or basic MPS model, can be written as follows.

$$\min \quad \sum_i \sum_t \{p_t^i x_t^i + f_t^i y_t^i + h_t^i s_t^i\} \tag{6}$$

$$s_{t-1}^i + x_t^i = d_t^i + s_t^i \quad \text{for all } i,t \tag{7}$$

$$x_t^i \leq M y_t^i \quad \text{for all } i,t \tag{8}$$

$$\sum_i \alpha^{ik} x_t^i + \sum_i \beta^{ik} y_t^i \leq L_t^k \quad \text{for all } t,k \tag{9}$$

$$x_t^i, s_t^i \geq 0,\ y_t^i \in \{0,1\} \quad \text{for all } i,t \tag{10}$$

where constraints (6)-(8) and (10) are the same as for the ULS model, and constraint (9) expresses the capacity restriction on each resource k in each period t.

Material Requirements Planning Model. As a last example model, we describe the multi-item multi-level capacitated lot-sizing model, that can be seen as the integration of the previous MPS model for finished products, and the ULS models for all intermediate products ans raw materials, into a single monolithic model. The purpose of this model is to optimize simultaneously the production and purchase of all items –from raw materials to finished products–, in order to satisfy for each item the external or independent demand coming from customers and the internal or dependent demand coming from the production of other items, over a short term horizon.

The dependency between items is modelled through the definition of the product structure, also called the bill of materials (BOM). The product structures are usually classified into Series, Assembly or General structures, see Figure 4.

The indices, variables and data are the same as before, except that, for simplicity, we also use the index $j = 1, \cdots, I$ to indentify items. For item i, we use the additional notation $S(i)$ to represent the set of successor items of i, i.e. the items consuming directly some amount of item i when they are produced. Note

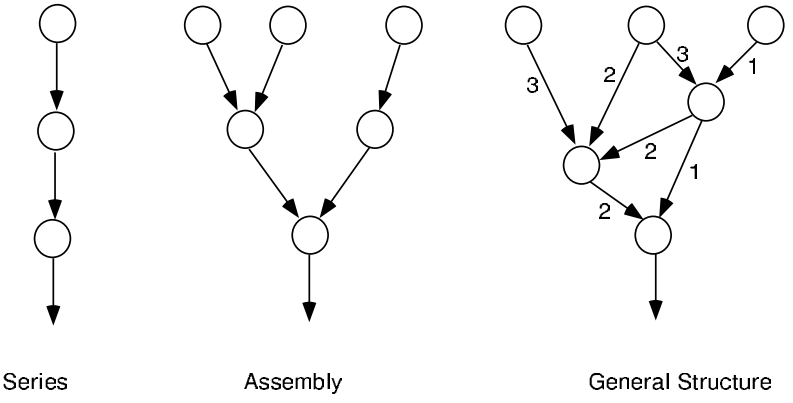


Fig. 4. Types of product structures in multi-level models

that for series and assembly structures, these sets $S(i)$ are singleton for all items i, and for a finished product i, we always have $S(i) = \emptyset$. And for $j \in S(i)$, we denote by r^{ij} the amount of item i required to make one unit of item j. These r^{ij} values are indicated along the edges (i, j) in Figure 4. This parameter r is used to identify the dependent demand, whereas d_t^i corresponds to the independent demand. For each item i, we denote by γ^i the lead time to produce or deliver any lot of i. More precisely, x_t^i represents the size of a production or purchase order of item i launched in period t, and delivered in period $t + \gamma^i$.

The natural formulation for the general product structure capacitated multi-level lot-sizing model, or the monolithic MRP model, is

$$\min \quad \sum_i \sum_t \{p_t^i x_t^i + f_t^i y_t^i + h_t^i s_t^i\} \tag{11}$$

$$s_{t-1}^i + x_{t-\gamma^i}^i = [d_t^i + \sum_{j \in S(i)} r^{ij} x_t^j] + s_t^i \quad \text{for all } i, t \tag{12}$$

$$x_t^i \leq M y_t^i \quad \text{for all } i, t \tag{13}$$

$$\sum_i \alpha^{ik} x_t^i + \sum_i \beta^{ik} y_t^i \leq L_t^k \quad \text{for all } t, k \tag{14}$$

$$x_t^i,\ s_t^i \geq 0,\ y_t^i \in \{0, 1\} \quad \text{for all } i, t \tag{15}$$

where the only difference with respect to the previous MPS model resides in the form of the flow conservation constraint (12). For each item i in each period t, the amount delivered from production or vendors is $x_{t-\gamma^i}^i$ ordered in period $t - \gamma^i$, and the demand to be satisfied is the sum the independent demand d_t^i and the dependent demand $\sum_{j \in S(i)} r^{ij} x_t^j$ implied by the production of immediate successors $j \in S(i)$.

Because of the multi-level structure, the presence of single item ULS models as submodels is less obvious, but we will show in Section 5.1 how to reformulate this model in the form of single item ULS models linked by capacity and product structure restrictions.

2.3 Optimization Methods

We are interested in this lecture in the optimization approaches used to solve production planning problems. This means that we want either to find provably optimal solutions, or to find near-optimal solutions with a performance guarantee, expressed usually in terms of a percentage of deviation of the objective value from the optimal value.

As we have seen in the examples from Section 2.2, many or most multi item production planning problems can be modelled as the following mixed integer programming (MIP) model

$$MIP = \min \sum_i \sum_t \{p_t^i x_t^i + h_t^i s_t^i + f_t^i y_t^i\} \tag{16}$$

$$(x^i, s^i, y^i) \in W^i \quad \text{for all } i \tag{17}$$

$$(x^i, s^i, y^i)_{i=1,\cdots,I} \in P \tag{18}$$

$$\tag{19}$$

where the index t is used to represent time periods, the index i is used to represent the different items, W^i represents the set of feasible solutions (i.e. lot sizes, setups and inventory levels) to the item i lot-sizing problem – like ULS, or some of its variants studied in Section 3.5–, and P represents the set of solutions satisfying a set of coupling linear constraints – like the capacity constraints (9) in Section 2.2, among others–.

Most optimization methods are based on easy to solve relaxations of the initial problem, either to prove optimality, or to provide a performance guarantee of some near-optimal solution. For example, the above problem can be solved by some standard MIP software using a branch and bound approach based on the following linear relaxation

$$LR = \min \sum_i \sum_t \{p_t^i x_t^i + h_t^i s_t^i + f_t^i y_t^i\} \tag{20}$$

$$(x^i, s^i, y^i) \in W_{LR}^i \quad \text{for all } i \tag{21}$$

$$(x^i, s^i, y^i)_{i=1,\cdots,I} \in P \tag{22}$$

$$\tag{23}$$

where W_{LR}^i represents the linear relaxation of the set W^i obtained by relaxing the constraints $y_t^i \in \{0, 1\}$ into $0 \leq y_t^i \leq 1$, for all i, t.

Unfortunately, this direct MIP approach can only be used for the solution of small size problems. In order to solve or to find good solutions for more realistic or real size problems, one has to work with better or tighter relaxations. Because of the structure of the intial problem, most efficient solution approaches are based on the following relaxation

$$LB = \min \sum_i \sum_t \{p_t^i x_t^i + h_t^i s_t^i + f_t^i y_t^i\} \tag{24}$$

$$(x^i, s^i, y^i) \in conv(W^i) \quad \text{for all } i \tag{25}$$

$$(x^i, s^i, y^i)_{i=1,\cdots,I} \in P \tag{26}$$

$$\tag{27}$$

where $conv(W^i)$ represents the convex hull of the solutions of W^i.

This bound LB is never worse, and typically much tighter than the bound LR, and is precisely the relaxation bound exploited in many different methods.

- Lagrangean relaxation methods and Dantzig-Wolfe or column generation methods give rise to schemes where the bound LB is iteratively computed by relaxing or eliminating the coupling constraints (26), and solving separately for each item i the resulting single item subproblems. These subproblems are optimization problems defined over the sets W^i. See, for instance, [1], [2], [10], [17], [66], [67], [68] for Lagrangean relaxation approaches, and [30], [69] for column generation approaches. See also Lemarechal [⟶ Lemaréchal] for a general presentation of Lagrangean relaxation.
- If a compact (small size) linear description of $conv(W^i)$ is available for each item i, then by adding the coupling constraints one gets a formulation whose linear relaxation gives the bound LB. Then a direct branch and bound approach can be used with this formulation. See, for instance, [19], [54] and [81].
- If a linear but non compact description of $conv(W^i)$ is available for each item i, using the corresponding formulation directly to compute LB takes too much time. Then the classical approach is to add (some of) the constraints defining $conv(W^i)$ in the course of optimization, instead of a priori, by solving a so-called separation problem over the set $conv(W^i)$. This gives branch and cut type methods. See, for instance, [7], [12] and [55].

Therefore, in order to design optimization methods for solving complex multi item production planning problems, one has to design algorithms to solve the single item subproblem, or one has to find a compact complete linear description for the subproblem, or one has to find a complete linear description with an efficient separation algorithm for the subproblem.

3 The CORE Subproblem: Single Item Uncapacitated Lot-Sizing

First, we study in detail in this Section the uncapacitated lot-sizing problem (ULS) already introduced in Section 2.2. Next, we will give the main optimization and reformulation results known for the main variants of this single item problem. A number of formulations and the general structure of this Section are imported from the more technical paper by Pochet and Wolsey [58], and from Chapter 13 of Wolsey [83].

3.1 Basic Formulation and Motivation

For the ease of reading, we have tried to separate as much as possible the previous modelling Section from this more technical Section. Therefore, we recall here the basic formulation of the ULS problem.

$$\min \sum_{t=1}^{n} (p_t x_t + f_t y_t + h_t s_t) \tag{28}$$

$$s_{t-1} + x_t = d_t + s_t \quad \text{for all } t \tag{29}$$

$$s_0 = s_n = 0 \tag{30}$$

$$x_t \leq d_{tn} y_t \quad \text{for all } t \tag{31}$$

$$x_t, s_t \geq 0,\ y_t \in \{0,1\} \quad \text{for all } t \tag{32}$$

where the large upper bound on x_t in constraint (31) has been replaced by its true upper bound d_{tn} equal to the sum of the demands from period t up to end of the horizon. In general, we define d_{kl} by $d_{kl} = \sum_{t=k}^{l} d_t$ for all $k \leq l$.

The set of feasible solutions to (29)-(32) is called X^{ULS}.

The first motivation to study this specific ULS problem is that ULS is solved repeatedly in MRP systems for each component, backward in the product structure from end products to raw materials. This approach, to define the production plans using a sequential item by item and uncapacitated model, has important shortcomings. By solving the capacity problems only after the solution of ULS for all items, and by neglecting the multi-level product structure, the production plans obtained are suboptimal in terms of cost and flexibility. The extension to capacitated models and multi-level models are discussed in Sections 4 and 5, respectively.

The other motivation to study ULS comes from the fact that it is the most common subproblem arising in complex multi-item production planning problems, and we have already seen in Section 2.3 the central role of single item subproblems in designing decomposition or branch and cut solution approaches for multi item problems.

3.2 Dynamic Programming Algorithm

The first natural question to ask in studying a subproblem is the following.

Question 1: *Is optimization over X^{ULS} polynomially solvable ?*

The answer is yes, and comes from a well known decomposition property of optimal solutions (see, for example, Zangwill [84]). This property means that there always exists an optimal solution where production only occurs in a period t when the entering stock s_{t-1} is zero.

Theorem 1 *There exists an optimal solution to problem ULS (28)-(32) where $s_{t-1}x_t = 0$ for all t*

Proof. Production may only occur in periods t with $y_t = 1$. For any selection of such production periods, the problem reduces to a minimum cost flow problem through the network shown in Figure 3. In any extreme solution, the arcs with positive flow form an acyclic subgraph and any acyclic subgraph satisfies $s_{t-1}x_t = 0$ for all t. ∎

There are a number of algorithms based on this decomposition property. The seminal paper of Wagner and Whitin [78] describes an $O(n^2)$ dynamic programming algorithm solving ULS. More efficient implementations of this algorithm running in $O(n\ log\ n)$ have been proposed by Federgrun, Tsur [20], Wagelmans, van Hoesel, Kolen [77], and Aggarwal, Park [3].

We describe here a simple forward dynamic program with a trivial implementation in $O(n^2)$. For simplicity of notation, we eliminate first the stock cost coefficients from the objective function using the flow balance constraints (29). That is, we replace $h_t s_t$ by $h_t[\sum_{k=1}^{t} x_k \ - d_{1,t}]$ in the objective (28), for all t (Recall that $d_{1,t} = \sum_{k=1}^{t} d_k$). Observe also that if $f_t < 0$ for some t in the objective (28), then $y_t = 1$ in all optimal solutions to (28)-(32). Therefore, problem ULS can be solved using the following objective function.

$$\min \sum_{t=1}^{n} (c_t x_t + f_t^+ y_t) \tag{33}$$

with $c_t = p_t + \sum_{i=t}^{n} h_i$ and $f^+ = \max\{f, 0\}$. In other words, to obtain the optimal solution to ULS ((28)-(32)) is suffices to add $\sum_{t=1}^{n} f_t^-$, with $f^- = \min\{f, 0\}$, to the optimal solution of the transformed problem (33), (29)-(32). From now on, and unless otherwise mentioned, we will work with this transformed objective function.

Now, let $H(k)$ be the cost of the minimum cost solution for problem ULS restricted to periods 1 up to k. From Theorem 1, there exists a minimum cost solution of value $H(k)$ such that, if period $t \leq k$ is the last production period

before period k, then $x_t = d_{tk}$, $x_l = 0$ for $l = t+1, \cdots, k$, and the cost of the solution for periods 1 to $t-1$ must be $H(t-1)$. This holds by the optimality principle, and the fact that $s_{t-1} = 0$ when $x_t > 0$. This allows us to define the following recursion to compute $H(k)$.

$$H(k) = \min_{1 \le t \le k} \{H(t-1) + f_t^+ + c_t \; d_{tk}\} \tag{34}$$

Starting from $H(0) = 0$, and computing $H(k)$ using (34) for $k = 1, \cdots, n$ leads to the value

$$H(n) \; + \; \sum_{t=1}^{n} f_t^- \tag{35}$$

of an optimal solution to ULS. It is easy to check that a direct implementation of this recursion leads to an $O(n^2)$ algorithm for ULS.

3.3 Extended Reformulations

Given the equivalence between optimization and separation (see Grotschel, Lovasz and Schrijver [25]), and given the fact that ULS is polynomially solvable and appears as a subproblem in many complex multi-item lotsizing problems, the next question to ask in order to solve these problems by branch and bound is now.

Question 2: *Is there a compact linear description for ULS ?*

The answer is again yes for such a simple lot-sizing problem, and we describe three of the popular extended reformulations for ULS.

Facility Location. The first reformulation takes the form of a simple facility location model, defined for facilities and customers located on a line, and introduced by Krarup, Bilde [34]. It consists in locating facilities ($y_t = 1$ if a facility is located at location t) to serve customer demands on a one way road (d_t is the demand to meet at location t) and minimize installation costs (f_t at location t), unit production costs (p_t at location t) and unit transportation costs (h_t from location t to location $t+1$).

If we define the variable w_{st} as the fraction of the demand d_t served from a production in period s (i.e. a facility located in s), for all $1 \le s \le t \le n$, and y_s as the usual 0/1 setup variable for period $s = 1, \cdots, n$, then, by analogy to the location model, ULS can be reformulated as

$$\text{(FL)} \qquad \min \sum_{t=1}^{n} f_t y_t + \sum_{s=1}^{n} \sum_{t=s}^{n} c_s d_t w_{st} \tag{36}$$

$$\sum_{s=1}^{t} w_{st} = 1 \quad \text{for all } t \tag{37}$$

$$w_{st} \leq y_s \quad \text{for all } 1 \leq s \leq t \leq n \tag{38}$$

$$w_{st}, y_t \geq 0 \quad \text{for all } 1 \leq s \leq t \leq n \tag{39}$$

$$y_t \leq 1 \quad \text{for all } t \tag{40}$$

$$y_t \text{ integer} \quad \text{for all } t \tag{41}$$

where constraint (37) expresses the demand satisfaction in period t from productions in periods $s \leq t$, constraint (38) imposes a setup in period s when there is production in s to satisfy some demand in period $t \geq s$, constraints (39)-(41) impose the nonnegativity and binary restrictions on the variables, the objective function (36) expresses the cost of the production plan in the form (33).

Formulation (36)-(41) is a valid reformulation for ULS, but due to the particular structure of the objective function, we have the following stronger result from Krarup, Bilde [34] (see also Barany, Van Roy and Wolsey [6] for a primal-dual proof).

Theorem 2 *The linear programming relaxation (36)-(40)of FL always has an optimal solution with y integer, and solves ULS.*

Note that the polyhedron (37)-(40) has fractional vertices. However, as there is always an optimal integral solution to (36)-(40), this linear formulation suffices to solve ULS. In order to obtain an integral polyhedron corresponding to the convex hull of solutions to ULS (i.e. a linear formulation with all integral vertices), one needs to add the constraints $w_{st} \geq w_{s,t+1}$ for $1 \leq s \leq t < n$ to (37)-(40).

Multicommodity. A classical way to tighten the formulation of fixed charge network flow problems is to decompose the flow along each arc of the network as a function of its destination. This defines a so-called multicommodity formulation by assigning a different commodity to each destination node, see Rardin and Choe [60].

We define the variable x_{it} (resp. s_{it}) as the production of commodity t (resp. inventory of commodity t) in period i. Commodity t corresponds to the demand delivered in period t. By decomposing the flow by commodity, ULS can be reformulated as

$$\text{(MC)} \qquad \min \sum_{i=1}^{n} \sum_{t=i}^{n} c_i x_{it} + \sum_{i=1}^{n} f_i y_i \tag{42}$$

$$s_{i-1,t} + x_{it} = \delta_{it} d_t + s_{it} \quad \text{for } 1 \leq i \leq t \leq n \tag{43}$$

$$s_{0t} = s_{tt} = 0 \quad \text{for } 1 \leq t \leq n \tag{44}$$

$$x_{it} \leq d_t y_i \quad \text{for } 1 \leq i \leq t \leq n \tag{45}$$

$$s_{it}, x_{it}, y_{it} \geq 0,\ y_{it} \leq 1 \quad \text{for } 1 \leq i \leq t \leq n \tag{46}$$

$$y_{it} \text{ integer} \quad \text{for } 1 \leq i \leq t \leq n \tag{47}$$

where the notation δ_{it} denotes 1 if $i = t$, and 0 otherwise. Constraint (43) is the flow conservation constraint of commodity t in period $i \leq t$. Constraints (44) impose there is no intial and no final inventory (end of period t) of commodity t. Constraint (45) forces the setup variable y_i to be 1 when there is production in period i, and using the decomposition of the flow the upper bound on x_{it} is d_t. Constraints (46)-(47) impose the nonnegativity and binary restrictions on the variables and constraint (42) expresses the cost of the production plan.

Formulation (42)-(47) is a valid reformulation for ULS. Using the equations (43) to eliminate the inventory variables from formulation (MC), and using the substitution $w_{s,t} = x_{s,t}/d_t$ between formulations MC and FL, it is easy to prove the following result.

Theorem 3 *The linear relaxations of MC ((42)-(46)) and FL ((36)-(40)) are equivalent. Therefore,the linear programming relaxation of MC always has an optimal solution with y integer, and solves ULS.*

Shortest Path. The last reformulation technique that we illustrate here is due to Martin [44], and first applied to lot-sizing problems by Eppen and Martin [19]. It can be used to transform the dynamic programming algorithm for ULS into a pair of primal-dual linear formulations.

The dynamic programming algorithm (34) constructs a least cost succession of so-called (disjoint) regeneration intervals $[t, k]$, with $t \leq k$, covering the planning horizon from period 1 to period n. Each interval $[t, k]$ represents the satisfaction of the demands in periods t up to k by the production of d_{tk} in period t. This is illustrated in Figure 5 with $n = 7$, where the entire production plan consists in the succession of 3 regeneration intervals: $[1, 2], [3, 6], [7, 7]$.

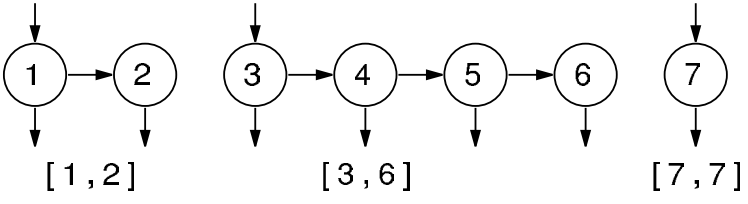


Fig. 5. An extreme solution as a succession of regeneration intervals

Using as variables $H(k)$, for $k = 0, \cdots, n$, the following linear program computes the optimal value $H(n) + \sum_{t=1}^{n} f_t^-$ of the dynamic program (34) solving ULS.

$$\max \quad H(n) - H(0) \quad + \quad \sum_{t=1}^{n} f_t^- \tag{48}$$

$$H(k) \leq H(t-1) + f_t^+ + c_t \; d_{tk} \quad \text{for all } 1 \leq t \leq k \leq n \tag{49}$$

$$H(0) = 0 \tag{50}$$

This claim is easy to prove by observing that the constraints (49) are equivalent to $H(k) \leq \min_{1 \leq t \leq k} \{H(t-1) + f_t^+ + c_t \; d_{tk}\}$, for all k. The maximization direction in the objective function (48) suffices to guarantee that the values $H(k)$, for all k, computed in the dynamic program (34) define an optimal solution to (48)-(50).

Associating the dual variable $\phi_{t,k}$ to each constraint in (49), the dual of (48)-(50) can be written as the minimum cost network flow problem

$$\min \sum_{t=1}^{n} \sum_{k=t}^{n} [c_t d_{tk} + f_t^+] \phi_{tk} \quad + \quad \sum_{t=1}^{n} f_t^- \tag{51}$$

$$\sum_{t=1}^{n} \phi_{tn} = 1 \tag{52}$$

$$\sum_{i=1}^{t} \phi_{it} - \sum_{l=t+1}^{n} \phi_{t+1,l} = 0 \quad \text{for } 1 \leq t \leq n-1 \tag{53}$$

$$-\sum_{l=1}^{n} \phi_{1l} = -1 \tag{54}$$

$$\phi_{tk} \geq 0 \quad \text{for } 1 \leq t \leq k \leq n \tag{55}$$

where the constraints (52)-(54) correspond to the flow conservation constraints of the network flow problem, with one flow constraint for each time period, and one unit of flow sent from the single source node (52) to the single sink node (54). Therefore, the linear program (51)-(55) defines a shortest path problem from source to sink, and it has 0-1 extreme points. It is easy to see that this formulation models the possible sequences of regeneration intervals, with $\phi_{t,k} = 1$ if the regeneration interval $[t, k]$ is part of the solution at cost $[c_t d_{tk} + f_t^+]$. This is illustrated in Figure 6 for the case $n = 3$.

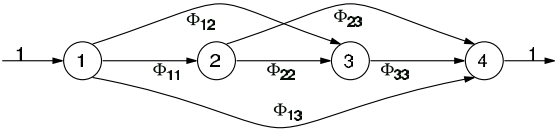


Fig. 6. Shortest Path Formulation of ULS ($n = 3$)

A final reformulation step can be used to obtain an equivalent formulation where the setup decisions are made explicit.

$$\text{(SP)} \qquad \min \sum_{t=1}^{n} \sum_{k=t}^{n} c_t d_{tk} \phi_{tk} + \sum_{t=1}^{n} f_t y_t \tag{56}$$

$$\sum_{t=1}^{n} \phi_{tn} = 1 \tag{57}$$

$$\sum_{i=1}^{t} \phi_{it} - \sum_{l=t+1}^{n} \phi_{t+1,l} = 0 \quad \text{for } 1 \le t \le n-1 \tag{58}$$

$$-\sum_{l=1}^{n} \phi_{1l} = -1 \tag{59}$$

$$\sum_{k=t}^{n} \phi_{tk} \le y_t \quad \text{for all } t \tag{60}$$

$$\phi_{tk}, y_t \ge 0, \; y_t \le 1 \quad \text{for } 1 \le t \le k \le n \tag{61}$$

In order to obtain this formulation, we have first replaced the objective term $\sum_{k=t}^{n} [f_t^+] \phi_{tk}$ by $[f_t^+] y_t$ and added the setup defining equalities $\sum_{k=t}^{n} \phi_{tk} = y_t$, for all t. Next we have replaced $\sum_{t=1}^{n} f_t^+ y_t + \sum_{t=1}^{n} f_t^-$ by $\sum_{t=1}^{n} [f_t^+ + f_t^-] y_t = \sum_{t=1}^{n} f_t y_t$, and relaxed $\sum_{k=t}^{n} \phi_{tk} = y_t$ into $\sum_{k=t}^{n} \phi_{tk} \le y_t$ to be able to force $y_t = 1$ when $f_t < 0$ without perturbing the flow constraints.

The above discussion has shown the following.

Theorem 4 *The linear program SP ((56)-(61)) always has an optimal solution with y integer, and solves ULS.*

Note that here the polyhedron (57)-(61) has no fractional vertices. To relate this shortest path formulation to the other formulations, one can show that formulation (57)-(61) is equivalent to the facility location formulation (37)-(40), augmented with $w_{st} \ge w_{s,t+1}$ for $1 \le s \le t < n$, using the substitution $\phi_{tk} = w_{tk} - w_{t,k+1}$ for $1 \le t \le k \le n$.

3.4 Complete Linear Description in the Initial Space

We have described in Section 3.3 several linear formulations for the ULS problem with n periods, but they involve $O(n^2)$ variables and $O(n^2)$ or $O(n)$ constraints.

Although these formulations are as tight as possible, to be able to solve large size multi item production planning problems – with many items and many periods –, we need to reduce the size of the formulation used for the single item subproblems. One way to achieve this is to look for a complete linear description of X^{ULS} in the initial variable space with only $O(n)$ variables. In that case, even if this complete linear description needs an exponential number of constraints, we do not need to add them all (a priori) to solve a particular instance. If we add the constraints using a separation algorithm, then at most $O(n)$ variables and constraints are needed to describe any particular extreme point of X^{ULS}.

This is typically what happens when branch and cut is used instead of branch and bound.

In order to identify a complete linear description of $conv(X^{ULS})$ in the initial space of variables, we first need to answer to the following question.

Question 3: *How to find valid inequalities for the set* X^{ULS}*?*

To answer to that question, we describe the typical way a researcher could think of in order to identify classes of valid inequalities.

One approach is to find numerically the complete description of small instances, and to generalize the results to classes of valid inequalities for arbitrary data and problem dimension.

Another approach is to solve the LP relaxations of some instances, and to find valid inequalities cutting off the non-integral optimal solutions so obtained, and then to generalize the results to classes of valid inequalities for arbitrary data and problem dimension. This approach gives also some intuition for the separation problem.

The example and the procedure used to find valid inequalities are taken from Wolsey [83]. Consider the fractional solution pictured in Figure 7 and obtained by solving the linear programming relaxation of the natural formulation (29)-(32) of some specific instance of ULS.

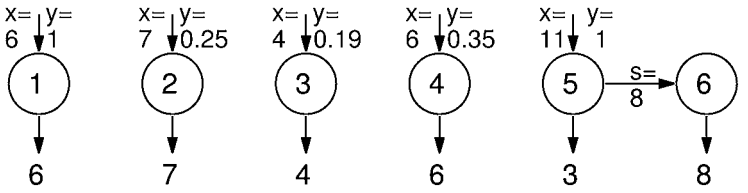


Fig. 7. A fractional solution($n = 6$)

Observation: As in the dynamic program, the solution decomposes into intervals of periods between successive points with $s_t = 0$. Intervals in which all the y variables are integral, like interval $[5, 6]$ in Figure 7, cannot be cut off because they correspond to integer (partial) solutions in $conv(X^{ULS})$. Therefore, we look at intervals in which the corresponding y variables are fractional, like interval $[2, 2]$ in Figure 7, and try to cut-off the corresponding point.

This can be done either by generating valid inequalities using known cutting planes, or by generating new and specialized valid inequalities for the problem at hand. We illustrate these two approach types in turn on this very simple problem and on the fractional interval $[2, 2]$ from Figure 7.

Valid Inequalities Using Known Cutting Planes. The fractional interval $[2, 2]$ from Figure 7 corresponds to a fractional solution of the single node or single period flow model defined by

$$\{(x_2, y_2, s_1, s_2) \in R^1_+ \times B^1 \times R^2_+ : s_1 + x_2 = d_2 + s_2,\ x_2 \leq d_{2,n} y_2\} \tag{62}$$

where $d_{2,n} = 28$ and $d_2 = 7$ in our toy example.

The well known flow cover inequalities have been defined for such single node flow models, see Padberg, Van Roy and Wolsey [50]. Let the cover be $C = \{x_2\}$, with excess capacity $\lambda = d_{2,n} - d_2$. The resulting flow cover inequality is

$$x_2 \leq d_2 \ - (d_{2,n} - \lambda)(1 - y_2) \ + s_2 \quad \text{, or equivalently} \tag{63}$$

$$x_2 \leq d_2\ y_2 \ + s_2. \tag{64}$$

which gives $x_2 \leq 7\ y_2 \ + s_2$ in our example.

This inequality is violated by the current fractional point from Figure 7, and it can be easily generalized to

$$x_t \leq d_t y_t \ + s_t \ \text{ forall } t \tag{65}$$

for arbitrary demand data and time period.

New Valid Inequalities. Consider again the fractional interval $[2, 2]$ from Figure 7. In any feasible solution, if there is no production in period 2 (which surely occurs if $y_2 = 0$), then the demand d_2 for period 2 must be produced earlier and must be contained in the entering stock s_1.

Therefore, a logical implication is $y_2 = 0$ implies $s_1 \geq d_2$. This implication can be converted into the valid linear inequality $s_1 \geq d_2(1 - y_2)$, which can be interpreted as if $y_2 = 0$ then $s_1 \geq d_2$, and if $y_2 = 1$ then $s_1 \geq 0$.

This inequality is violated by the current fractional point from Figure 7, and can be easily generalized to

$$s_{t-1} \geq d_t(1 - y_t) \ \text{ forall } t \tag{66}$$

for arbitrary demand data and time period.

Reoptimization and Generalization. After some violated valid inequalities have been identified, they can be added to the initial formulation, and the resulting tighter linear formulation solved.

Using the flow balance equality, $s_{t-1} + x_t = d_t + s_t$, one observes that the 2 classes of valid inequalities found (65) and (66) are equivalent. We add these inequalities for $t = 2,\ 3,\ 4$ to the formulation, reoptimize, and get the new non-integral solution pictured in Figure 8. Again, to find new violated valid inequalities, it suffices to look at the solution for periods 1 up to 3 because the partial solution for periods 4 to 6 is integral.

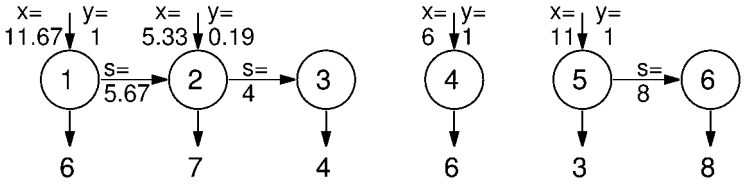


Fig. 8. A fractional solution after some cuts ($n = 6$)

By the same type of logical arguments, one can derive the valid inequality $s_1 \geq d_2(1-y_2) + d_3(1-y_2-y_3)$, which says that s_1 must carry over d_2 if $y_2 = 0$ and must also carry over d_3 if both $y_2 = 0$ and $y_3 = 0$.

This inequality is violated by the current fractional point from Figure 8, and it can be generalized to

$$s_{t-1} \geq \sum_{k=t}^{l} d_k(1 - y_t - \cdots - y_k) \quad \text{for } 2 \leq t \leq l \leq n \tag{67}$$

Complete Linear Description. Once classes of valid inequalities have been identified, the next natural questions to ask are, in sequence.

Question 4: *Are the valid inequalities (67) facet defining for conv*(X^{ULS})*?*
Question 5: *Do we have a complete linear description of conv*(X^{ULS})*?*

We will not describe here the various techniques that can be used to prove that some valid inequalites are facet defining, and that some valid inequalities suffice to describe the convex hull of a problem. For a general presentation of these topics, we refer the reader to Nemhauser and Wolsey [47]. We will rather state the main results for the ULS problem, with adequate references for their proofs.

Using the flow balance constraints to eliminate the stock variables (i.e. $s_t = [\sum_{k=1}^{t} x_k - d_{1,t}]$), the class of valid inequalities (67) is rewritten as

$$\sum_{k=1}^{t-1} x_k \; + \; \sum_{k=t}^{l} d_{k,l} \; y_k \geq d_{1,l} \quad \text{for } 2 \leq t \leq l \leq n \tag{68}$$

These inequalities (68) do not suffice to describe $conv(X^{ULS})$, but they suffice to solve most practical instances without any branching, i.e. the solution of the linear relaxation of (28)-(32), augmented with the inequalities (68), is almost always integral. We will see later why, and in which sense, this is the case (see Section 3.5). To obtain a linear description of $conv(X^{ULS})$, the inequalities (68) have to be further generalized, as shown in the following Theorem.

Theorem 5 *Assuming $d_t > 0$ for all t, $conv(X^{ULS})$ is described by*

$$\sum_{i \in L \setminus S} x_i + \sum_{i \in S} d_{il} y_i \geq d_{1l} \quad \text{for } 1 \leq l \leq n, S \subseteq L = \{1, \cdots, l\} \tag{69}$$

$$y_1 = 1 \tag{70}$$

$$\sum_{i=1}^{n} x_i = d_{1n} \tag{71}$$

$$x_i \geq 0, \; 0 \leq y_i \leq 1 \quad \text{for all } i \tag{72}$$

Barany, Van Roy, Wolsey [6] give a primal-dual proof of Theorem 5, Pochet and Wolsey [58] give a direct proof using a technique due to Lovasz [40].

Separation. We have obtained a complete linear programming formulation in the original variables (x, y) that solves ULS. Unfortunately this formulation contains an exponential number of so-called (l, S) inequalities (69). In a branch and cut or cutting plane approach, instead of adding all these inequalities a priori to the formulation, they are added in the course of optimization when they are needed. We then need to solve the following separation problem.

We define X_{LR}^{ULS} as the linear relaxation of the natural formulation (29)-(32) of ULS.

Separation Given $(x^*, y^*) \in X_{LR}^{ULS}$:

- Either we find an (l, S) inequality violated by (x^*, y^*),
- or we prove that all (l, S) inequalities are satisfied by (x^*, y^*).

To find the most violated (l, S) inequality for fixed $l \in \{1, \cdots, n\}$, it suffices to test whether or not $\sum_{i=1}^{l} \min(x_i^*, d_{il} y_i^*) < d_{1l}$.

- If this holds, then the (l, S^*) inequality with $S^* = \{i \in L : d_{il} y_i^* \leq x_i^*\}$ is the most violated inequality for the given value of l.
- Otherwise, there is no violated (l, S) inequality for the given value of l.

By enumerating all possible values of l, we obtain an $O(n^2)$ separation algorithm for the (l, S) inequalities.

3.5 Single Item Variants of Uncapacitated Lot-Sizing

Using the same approach as for ULS, we describe here very briefly several ULS variants that have been studied in the literature, and for which complexity and reformulation results are available.

Start-up Costs. We consider first the single item lot-sizing problem with start-up costs. This model is identical to ULS, with an additional startup cost incurred in the first period of a production sequence or batch. More precisely, the startup cost is incurred in period t, if there is a setup in period t, but not in period $t-1$. To introduce this startup cost model into the formulation, we define additional binary variables z_t taking the value 1 when there is a startup in period t. This leads to the formulation

$$\min \quad \sum_t \{p_t x_t + f_t y_t + h_t s_t + g_t z_t\} \tag{73}$$

$$s_{t-1} + x_t = d_t + s_t \quad \text{for all } t \tag{74}$$

$$s_0 = s_n = 0 \tag{75}$$

$$x_t \leq M y_t \quad \text{for all } t \tag{76}$$

$$y_t \geq z_t \geq y_t - y_{t-1} \quad \text{for all } t \tag{77}$$

$$x_t, s_t \geq 0,\ y_t, z_t \in \{0,1\} \quad \text{for all } t \tag{78}$$

where constraints (77) impose a startup in period t if there is a setup in t, but not in $t-1$, and also force a setup in period t if there is a startup in period t. We define ULSS as the optimization problem (73)-(78).

Magnanti and Vachani [42] show that the decomposition property of optimal solutions given in Theorem 1 still holds. This allows one to define a dynamic programming algorithm for solving ULSS, as in van Hoesel [71], with a direct implementation running in $O(n^2)$. Again, more effcient implementations run in $O(n\ log\ n)$, see van Hoesel [71] and Aggarwal, Park [3].

The techniques used to generate compact linear formulations for ULS in Section 3.3 can be extended to ULSS. In particular, Wolsey [81] provides a facility location type formulation and van Hoesel, Wagelmans and Wolsey [72] provide a proof that its linear relaxation solves ULSS, Rardin and Wolsey [61] prove that the muticommodity linear formulation solves ULSS, and the technique of Martin [44] can be used to generate a shortest path linear formulation from the dynamic program solving ULSS. All these compact linear formulations involve $O(n^2)$ variables and constraints.

A class of valid inequalities can easily be obtained starting from the (l, S) inequalities (69) of ULS. For example, starting with the (l, S) inequality $s_1 \geq d_2(1 - y_2) + d_3(1 - y_2 - y_3) + d_4(1 - y_2 - y_3 - y_4)$, which is also valid in the presence of startup costs, one can derive the stronger valid inequality $s_1 \geq$

$d_2(1-y_2) + d_3(1-y_2-z_3) + d_4(1-y_2-z_3-z_4)$. This inequality says that s_1 must carry over d_2 if $y_2 = 0$, must also carry over d_3 if $y_2 = z_3 = 0$ (which implies that $y_3 = 0$, and thus there is no production in periods 2 and 3), and must carry over d_4 when $y_2 = z_3 = z_4 = 0$ (which implies $y_3 = y_4 = 0$).

This inequality can be generalized to

$$s_{t-1} \geq d_t(1-y_t) + \sum_{k=t+1}^{l} d_k(1-y_t-z_{t+1}-\cdots-z_k) \quad \text{for } 2 \leq t < l \leq n. \tag{79}$$

However, as for ULS, the inequalities (79) do not suffice to obtain a linear description of the convex hull of solutions, but they suffice to solve most practical instances by linear programming, without any branching (see Section 3.5). A further generalization of inequalities (79) proposed by van Hoesel, Wagelmans and Wolsey [72] is proved to define the convex hull of feasible solutions to ULSS in the initial variable space. This formulation contains an exponential number of constraints that can be separated in time of $O(n^3)$.

Constant Capacity. We consider now the single item constant capacity lot-sizing problem (CCLS) defined by (80)-(84). This constant capacity model is quite realistic and occurs often in practice when the production, inventory and demand are expressed in equivalent-time units.

$$\min \quad \sum_t \{p_t x_t + f_t y_t + h_t s_t\} \tag{80}$$

$$s_{t-1} + x_t = d_t + s_t \quad \text{for all } t \tag{81}$$

$$s_0 = s_n = 0 \tag{82}$$

$$x_t \leq C y_t \quad \text{for all } t \tag{83}$$

$$x_t, s_t \geq 0,\ y_t \in \{0,1\} \quad \text{for all } t \tag{84}$$

The formulation of CCLS is identical to the natural formulation of ULS, except that constraint (83) expresses now a limited capacity when $C < d_{1n}$. The capacity limit C is constant over time.

Florian and Klein [23] propose a dynamic programming programming algorithm for CCLS running in $O(n^4)$ time, and van Hoesel, Wagelmans [73] reduce this time to $O(n^3)$. The technique of Martin [44] can be used to generate a shortest path linear formulation from the dynamic program solving CCLS with $O(n^4)$ variables and constraints. Pochet and Wolsey [56] give a linear programming formulation with $O(n^3)$ variables and constraints solving CCLS that corresponds to a shortest path sequence of regeneration intervals, but cannot be derived from the dynamic program solving CCLS.

Pochet and Wolsey [56] describe also a new class of valid inequalities for CCLS. This class was generalized to other problems than CCLS, and a new procedure to obtain these inequalities by mixing the mixed-integer rounding inequalities of Nemhauser and Wolsey [48] is described in Gunlunk and Pochet [26]. We illustrate here this class of inequalities with an example. The reader is referred to the above papers for a formal definition of these inequalities.

Consider the instance of CCLS with $C = 10$, $n = 5$ and demand $d = (1, 7, 4, 2, 3)$ represented in Figure 9. First, by summing the flow conservation constraints (81) from period 2 to some period $l \geq 2$, by replacing the production variables x_t by their variable upper bounds Cy_t, one gets the following starting or base (valid but redundant) inequalities

$$s_1 + 10y_2 \geq 7, \tag{85}$$
$$s_1 + 10y_2 + 10y_3 \geq 11, \tag{86}$$
$$s_1 + 10y_2 + 10y_3 + 10y_4 + 10y_5 \geq 16. \tag{87}$$

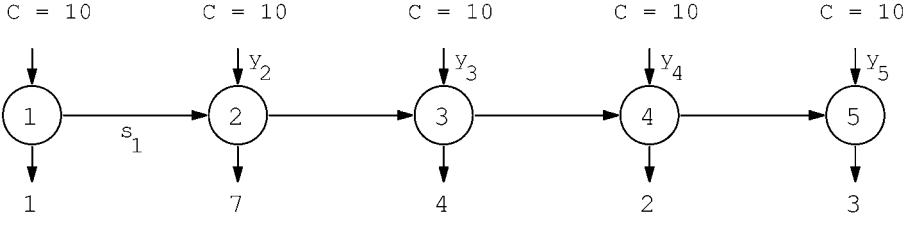


Fig. 9. A CCLS example($n = 5$)

Then, standard mixed-integer rounding (MIR) inequalities [⟵ Martin] for a general description of MIR inequalities) can be constructed from these base inequalities. This gives

$$s_1 \geq 7(1 - y_2), \tag{88}$$
$$s_1 \geq 1(2 - y_2 - y_3), \tag{89}$$
$$s_1 \geq 6(2 - y_2 - y_3 - y_4 - y_5). \tag{90}$$

Finally, the MIR inequalities can be mixed by taking differences of terms at the right hand side. This gives, for example, the new inequalities

$$s_1 \geq 1(2 - y_2 - y_3) + (6 - 1)(2 - y_2 - y_3 - y_4 - y_5) + (7 - 6)(1 - y_2), \tag{91}$$
$$s_1 \geq 1(2 - y_2 - y_3) + (7 - 1)(1 - y_2), \tag{92}$$
$$s_1 \geq 6(2 - y_2 - y_3 - y_4 - y_5) + (7 - 6)(1 - y_2). \tag{93}$$

For example, the second of these inequalities can be interpreted as follows. As $d_{23} = 11$, if there is only one production period in periods 2 and 3 (i.e. $y_2+y_3 = 1$, so a capacity of 10 is available in periods 2 and 3), then s_1 must carry over at least one unit in order to satisfy demand in periods 2 and 3. But if, in addition, $y_2 = 0$, then s_1 must carry over the whole demand $d_2 = 7$ (instead of only 1 unit when $y_2 = 1$ and $y_3 = 0$).

Such mixed MIR inequalities do not suffice to describe the convex hull of solutions for CCLS by linear inequalites only, but they suffice to solve most practical instances without branching (see Section 3.5). Examples of facet defining inequalities that are not of the type (91)-(93) are given in Pochet and Wolsey [56]. However, in this case, no complete linear description of the convex hull of solutions is known for CCLS.

Leung, Magnanti and Vachani [37], Pochet [53], Loparic, Marchand and Wolsey [39] define and test valid inequalities for the case where the capacities may vary over time.

Backlogging. Another variant of ULS is to allow for backlogging. To model this situation, we introduce a new variable r_t, for each period t, representing the backlog of demand at the end of period t. So, r_t is the demand from periods $\{1, \cdots, t\}$ that will be delivered late (in some period in $\{t+1, \cdots, n\}$). A classical formulation for the uncapacitated lot-sizing problem with backlogging (BLS) is (94)-(98)

$$\min \quad \sum_t \{p_t x_t + f_t y_t + h_t s_t + g_t r_t\} \tag{94}$$

$$(s_{t-1} - r_{t-1}) + x_t = d_t + (s_t - r_t) \quad \text{for all } t \tag{95}$$

$$s_0 = r_0 = s_n = r_n = 0 \tag{96}$$

$$x_t \leq M y_t \quad \text{for all } t \tag{97}$$

$$x_t, s_t, r_t \geq 0, \ y_t \in \{0, 1\} \quad \text{for all } t \tag{98}$$

where the flow conservation constraints (95) are adapted to take into account these backlogging variables. In each period t, r_t represents a flow coming from period $t + 1$ to period t (to satisfy artificially some demand in periods at or before t). Constraints (96) impose that there is no initial and final inventory and backlog. It is assumed in 94 that each unit of product backlogged at the end of period t costs g_t.

Zangwill [85] describes a dynamic programming programming algorithm for BLS running in $O(n^2)$ time, and Wagelmans, van Hoesel, Kolen [77] obtain an implementation in $O(n \ log \ n)$ time. Pochet and Wolsey [54] provide facility location, multicommodity and shortest path linear formulations involving $O(n^2)$ variables and constraints.

Pochet and Wolsey [54] describe a new class of valid inequalities for BLS. This class, that can also be applied to other problems than BLS, gives a way to generalize the class of valid inequalities defined in Van Roy and Wolsey [74] for uncapacitated fixed charge network flow problems. We illustrate this class of inequalities with a simple example. The reader is referred to the above papers for a formal definition of these inequalities.

Consider an instance of BLS with $n \geq 5$, and represented in Figure 10. First we define a subset of the inventory and backlog variables, for example $\{s_1, s_3, r_2, r_4\}$, and then we find logical conditions under which some demand has to flow through the selected variables. For example, if $y_2 = 0$ then $s_1 + r_2 \geq d_2$, giving the base valid inequality $s_1 + r_2 \geq d_2(1 - y_2)$. Similarly, one can obtain the base valid inequalities $s_1 + r_4 \geq d_3(1 - y_2 - y_3 - y_4)$ and $s_3 + r_4 \geq d_4(1 - y_4)$. Finally we put all these inequalities together to obtain the valid inequality

$$s_1 + s_3 + r_2 + r_4 \geq d_2(1 - y_2) \tag{99}$$
$$+ d_3(1 - y_2 - y_3 - y_4) \tag{100}$$
$$+ d_4(1 - y_4) \tag{101}$$

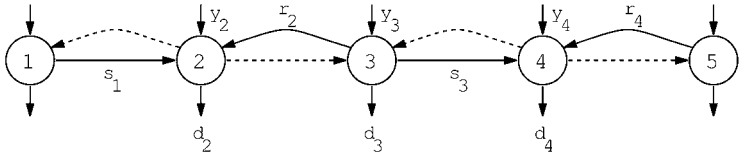


Fig. 10. A BLS example ($n \geq 5$)

Again, the inequalities illustrated above do not suffice to obtain a linear description of the convex hull of solutions to BLS (such a description is unknown for general objective functions), but they suffice to solve most practical instances of BLS without branching (see Section 3.5).

Wagner-Whitin Costs. We analyze in this Section the special case of the previously studied lot-sizing problems where the production, inventory and backlogging costs satisfy the Wagner-Whitin condition. An instance of ULS, ULSS, CCLS or BLS is said to have Wagner-Whitin costs if $p_t + h_t \geq p_{t+1}$ and $p_{t+1} + g_t \geq p_t$, for all t. This means that the unit production, unit inventory and unit backlogging costs are non speculative in the sense that,

– if a demand d_t is satisfied from production in periods t or before, then this production occurs as late before t as possible (with respect to setup decisions and capacity available) because producing and stocking in earlier periods costs more than producing later ($p_t + h_t \geq p_{t+1}$),

– if a demand d_t is satisfied from production in periods t or after, then this production occurs as early after t as possible (with respect to setup decisions and capacity available) because producing and backlogging from later periods costs more than producing earlier ($p_{t+1} + g_t \geq p_t$).

The implication of this non speculative cost condition is that

– the complexity bound for the optimization of the subproblems ULS, ULSS, CCLS and BLS are reduced,
– new and smaller (in number of variables and constraints) linear reformulations can be derived based on the fact that there always exists stock minimal and backlog minimal solutions (i.e. each stock and backlog variable is set to its minimal minimal possible value with respect to setup decisions and capacity constraints) to these problems,
– fewer constraints are needed in the complete linear description of the convex hull of feasible solutions in the initial space, and the complexity bounds of the separation algorithms are reduced.

Federgrun, Tsur [20], Wagelmans, van Hoesel, Kolen [77], and Aggarwal, Park [3]. Furthermore have shown that the dynamic programs solving ULS and ULSS can be implemented in $O(n)$ when the cost function satisfies the Wagner-Whitin assumption.

For the linear reformulation, in the case of ULS, we have seen that inequalities (68) do not suffice to obtain a linear description of $conv(X^{ULS})$, but they suffice to solve ULS by linear programming with a Wagner-Whitin objective function. This means that in this case there always exists an optimal solution with y integer to the linear relaxation of (28)-(32) augmented by (68). Moreover, this Wagner-Whitin condition is a very natural assumption that is almost always satisfied by the objective functions encountered in practice. This is the reason why it generally suffices to add the inequalities (68) to solve most ULS problems by LP.

Similarly, there always exists an optimal solution with y integer to the linear relaxation of problem (73)-(78), plus (79). Therefore, adding inequalities (79) suffices to solve ULSS by LP. Similar results hold for problems CCLS and BLS.

We summarize in Table 2 the results about the linear reformulations of problems ULS, ULSS, CCLS and BLS. See Pochet and Wolsey [57] for a detailed description of the formulations and separation algorithms under the Wagner-Whitin cost assumption. We indicate in the table the sizes and complexity bounds for general cost functions, and the size/complexity reduction obtained, if any, with Wagner-Whitin costs (after the $\rightarrow_{WW}$ sign). In the table, ?? indicates that no result is known, to the best of our knowledge. These results are extended to the uncapacitated lot-sizing problem with backlogging and startup costs in Agra and Constantino [4].

Table 2. Size/complexity reductions with Wagner-Whitin costs

	Extended linear formulation	Initial space linear formulation
Uncapacitated (ULS)		
number of variables	$O(n^2)$	$O(n)$
number of constraints	$O(n^2)$	$O(2^n) \rightarrow_{WW} O(n^2)$
separation complexity		$O(n^2)$
Startup costs (ULSS)		
number of variables	$O(n^2)$	$O(n)$
number of constraints	$O(n^2)$	$O(2^n) \rightarrow_{WW} O(n^2)$
separation complexity		$O(n^3) \rightarrow_{WW} O(n^2)$
Constant capacity (CCLS)		
number of variables	$O(n^3) \rightarrow_{WW} O(n^2)$	$O(n)$
number of constraints	$O(n^3) \rightarrow_{WW} O(n^2)$	?? $\rightarrow_{WW} O(2^n)$
separation complexity		?? $\rightarrow_{WW} O(n^2\ log\ n)$
Backlogging (BLS)		
number of variables	$O(n^2) \rightarrow_{WW} O(n)$	$O(n)$
number of constraints	$O(n^2)$	?? $\rightarrow_{WW} O(2^n)$
separation complexity		?? $\rightarrow_{WW} O(n^3)$

Profit Maximization. As a last variant of the ULS problem, we consider the profit maximization problem PLS formulated as

$$\max \quad \sum_t \pi_t v_t \; - \; \sum_t \{p_t x_t + f_t y_t + h_t s_t\} \tag{102}$$

$$s_{t-1} + x_t = v_t + s_t \quad \text{for all } t \tag{103}$$

$$s_0 = s_n = 0 \tag{104}$$

$$x_t \leq M y_t \quad \text{for all } t \tag{105}$$

$$0 \leq v_t \leq U_t \quad \text{for all } t \tag{106}$$

$$x_t, s_t \geq 0, \; y_t \in \{0,1\} \quad \text{for all } t \tag{107}$$

where the variable v_t, representing the amount sold in period t, replaces the fixed demand in the flow balance constraint (103), and is bounded from above by the maximum forecasted market potential U_t in constraint (106). The objective function (102) represents the profit contribution to be maximized and defined as the product of sales at a price π_t per unit sold in period t, minus production and inventory costs.

Such a problem arises typically as a subproblem in profit maximization multi item production planning problems, where there are capacity linking constraints between the items. For each single item subproblem, the production costs in each period contain implicitely, or even explicitely when the problem is solved by decomposition or Lagrangean relaxation, the opportunity cost of the capacity

used for production. The problem consists in allocating the capacity to the items in order to maximize overall profit contribution.

Loparic, Pochet and Wolsey [38] analyze the optimization algorithms and linear formulations for problem PLS with additional lower bounds on stocks to model required safety stock levels. They show that the problem can be solved in $O(n^4)$ by dynamic programming, they propose linear formulations with $O(n^2)$ variables and constraints, and they provide a complete linear description with an exponential $O(2^n)$ number of constraints of the convex hull of solutions in the initial variable space. Again the problem can be solved by LP with only a subset of these constraints – but still exponentially many – when the cost function satisfies the Wagner-Whitin assumption.

We only illustrate here the type of stock minimal inequalities needed to solve PLS by LP in the presence of such Wagner-Whitin costs. The inequality

$$s_1 \geq (v_2 - U_2 y_2) \; + \; (v_4 - U_4(y_2 + y_3 + y_4)) \tag{108}$$

is valid for the convex hull of solutions to (103)-(107) and it is a slight generalisation of the inequality

$$s_1 \geq (d_2 - d_2 y_2) \; + \; (d_4 - d_4(y_2 + y_3 + y_4)) \tag{109}$$

needed to describe $conv(X^{ULS})$. The inequality (108) expresses that if $y_2 = 0$, then $x_2 = 0$ and s_1 must cover the sales level v_2, and if $y_2 = 1$ then there is no restriction on s_1 because $v_2 \leq U_2$ can be covered by x_2. Therefore, $s_1 \geq (v_2 - U_2 y_2)$ is valid. Similarly, $s_1 \geq v_4$ when there is no production in periods 2, 3 and 4, that is when $y_2 + y_3 + y_4 = 0$.

4 Formulations of Capacitated Models

We analyze in this section the main classes of capacitated production planning models. We first classify the models, and then describe in more details the basic mathematical programming models and formulations for the two main model classes, the big buckets and small buckets models.

4.1 Classes of Capacitated Models

The first and important shortcoming of the classical MRP approach in discrete manufacturing is to decompose the planning procedure in two phases. First the uncapacitated planning problem is solved, to obtain production or purchase plans for all items in the product structure. Then the capacity problems (overloads) created by the production plans are resolved by smoothing the load profiles, i.e. by shifting some of the production lots forward or backward in time. This lack of integration between the uncapacitated planning phase and the capacitated post-optimization phase explains that there is no guarantee to obtain a feasible

production plan if one exists, and explains why the production plans obtained are suboptimal in terms of cost. There is thus a need to develop capacitated models allowing one to obtain feasible and optimized production plans, and providing the needed flexibility and tools to adapt the production plans to demand and to market requirements.

There are two main classes of capacitated models that can be distinguished based on the size, or time length, of the production periods. These model classes are known as the big buckets and small buckets models, and correspond to quite different planning environments and problems that we describe now.

Discrete time planning models are models where the planning horizon is divided into discrete time periods, and where all events – production, sales, ... – occuring during a period are aggregated in time as if they occur all at the same time. All the models that we have studied so far are of this type.

Big buckets planning models are discrete time models with large time periods where the typical aim is to take global decisions about the resources to be used and to acquire, and about the assignment of products to facilities or departments, rather than to plan the production to meet some precise and firm short term customer orders, or to control the short term output from the shop floor. Therefore, there is usually no need for a detailed representation of time. Moreover, and very often, a more detailed time representation would not make sense because the required detailed information (e.g. detailed sales forecasts) would not be available on time.

Such models are used for solving tactical medium term resource planning problems. The size of the time period is usually taken as one week, or one month, depending on the particular problem, and the planning horizon ranges from a few weeks to months, up to one year.

Small buckets planning models are discrete time models with small time periods where the aim is to take detailed decisions about the materials flow in the shop floor, the sequence of production lots on machines to control short term output and capacity usage, to verify that technical precedence constraints between lots can be satisfied. A detailed time representation is needed to represent accurately these events.

Such models are used for solving operational short term materials planning problems. The size of the time period is usually taken as one week, or one day, down to one hour, depending on the particular problem, and the planning horizon ranges from a few weeks down to one day. In some cases, similar models have to be used for longer term problems with larger time periods when, for instance, the detailed sequence of products has a significant impact on the medium term capacity utilization. This is for example often the case in the process or chemical industry where change-over times may represent days of production lost.

For simplicity, we will consider that big buckets models are discrete time models where there is no need to represent the sequence of events inside a period,

and between the different periods. Small buckets models are models where it is required to represent the sequence of events from one period to the other.

4.2 Big Buckets Models: Capacitated Lot-Sizing

Basic Big Buckets Multi-item Model. The master production scheduling model (6)-(10) formulated in Section 2.2, also known as the capacitated multi-item capacitated lot-sizing problem, is the basic example of big bucket model. We formulate here the simplified version without setup times, called MICLS, that has been widely studied in the literature.

$$\min \quad \sum_i \sum_t \{p_t^i x_t^i + f_t^i y_t^i + h_t^i s_t^i\} \tag{110}$$

$$s_{t-1}^i + x_t^i = d_t^i + s_t^i \quad \text{for all } i,t \tag{111}$$

$$x_t^i \leq C_t^i y_t^i \quad \text{for all } i,t \tag{112}$$

$$\sum_i \alpha^i x_t^i \leq L_t \quad \text{for all } t \tag{113}$$

$$x_t^i, s_t^i \geq 0, \ y_t^i \in \{0,1\} \quad \text{for all } i,t \tag{114}$$

where constraint (112) represents a single item capacity constraint and constraint (113) is the multi-item capacity linking constraint. Note that if $C_t^i = L_t/\alpha^i$, then constraint (112) is just used to define the setup variable y_t^i without imposing another capacity constraint than the linking constraint (113).

Several optimization approaches have been proposed and tested for problem MICLS. They are based on the single item uncapacitated ULS subproblems, and use the optimization and reformulation results surveyed in Section 3. Thizy and Van Wassenhove [67], Chen and Thizy [10] test and compare several Lagrangean relaxation schemes, Eppen and Martin [19] solve the problem by branch and bound using extendend linear formulations for the ULS subproblems, Barany, Van Roy and Wolsey [7] and Pochet, Wolsey [55] use a cut and branch approach (cuts are added only before the branch and bound enumeration), Constantino [13] solve MICLS by a branch and cut approach.

Valid Inequalities Using Known Cutting Planes. As we did in Section 3.4, we illustrate how to identify classes of valid inequalities for problem MICLS using known cutting planes. This approach can be used to tighten the formulation and solve such problems using branch and cut approaches.

The first idea is to use the mixed integer rounding (MIR) inequalities from Nemhauser and Wolsey [48] [⟶ Martin]. Marchand and Wolsey [43] show how to generate MIR inequalities from 0-1 knapsack problems with continuous variables. To use this idea, it suffices to construct relaxations of MICLS in the form of 0-1 knapsacks with continuous variables.

We illustrate this approach with an example. First we construct a single item knapsack relaxation for item i by summing the flow conservation constraints (111) over periods $t = k, \cdots, l$, replacing s_l^i by its lower bound 0, and replacing x_t^i by its upper bound $\min[C_t^i, d_{tl}^i]y_t^i$, for $t = k, \cdots, l$. This gives the following knapsack relaxation with one continuous variable.

$$s_{k-1}^i \; + \; \sum_{t=k}^{l} \min[C_t^i, d_{tl}^i] y_t^i \; \geq \; d_{kl}^i \tag{115}$$

$$s_{k-1}^i \; \geq 0, \; y_t^i \in \{0,1\} \quad \text{for} \quad t = k, \cdots, l \tag{116}$$

where as before we use the notation $d_{tl}^i = \sum_{j=t}^{l} d_j^i$.

Suppose now that we have solved the linear relaxation of MICLS and we want to cut off the fractional solution obtained and illustrated in Figure 11 for one item i and for periods 2 up to 5.

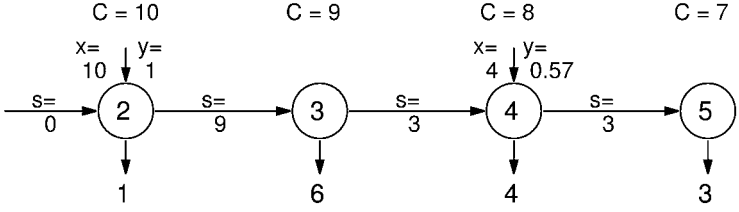


Fig. 11. A fractional solution for one item in MICLS

The corresponding knapsack relaxation is (where we have dropped the item i superscript for simplicity)

$$s_1 \; + 10y_2 + 9y_3 + 7y_4 + 3y_5 \geq \; 14, \tag{117}$$

which can be relaxed into (where we consider variable y_5 as continuous)

$$(s_1 + 3y_5) \; + 10(y_2 + y_3 + y_4) \geq \; 14. \tag{118}$$

We obtain the classical MIR inequality

$$\frac{s_1}{4} + \frac{3}{4}y_5 \; + y_2 + y_3 + y_4 \; \geq \; 2 \tag{119}$$

by dividing (118) by 10 and rounding. The inequality (119) is violated by the fractional point in Figure 11.

Another idea is to use the well known flow cover inequalities that have been defined for single node flow models in Padberg, Van Roy and Wolsey [50]

[⟶ Martin] for a general description of flow cover inequalities). Again, it suffices to construct single node flow relaxations, and to generate the corresponding flow cover inequalities.

Consider first the single node flow relaxation defined for a single item i over several consecutive periods, and obtained by summing the flow conservation constraints (111) over periods $t = k, \cdots, l$, replacing s^i_{k-1} by its lower bound 0. This gives the following single node flow model.

$$\sum_{t=k}^{l} x^i_t \;\leq\; d^i_{kl} \;+\; s^i_l \tag{120}$$

$$x^i_t \leq \min[C^i_t, d^i_{tl}]y^i_t + s^i_l \quad \text{for} \;\; t = k, \cdots, l \tag{121}$$

$$s^i_l \;\geq 0,\; x^i_t \geq 0,\; y^i_t \in \{0,1\} \quad \text{for} \;\; t = k, \cdots, l \tag{122}$$

where we have changed the variable upper bound constraint in (121) by adding the flow s^i_l at the right hand side, allowing us to reduce the coefficient of y^i_t.

Again, suppose that we are given the fractional solution illustrated in Figure 11. The corresponding single node flow relaxation is

$$x_2 + x_3 + x_4 + x_5 \leq 14 \tag{123}$$

$$x_2 \leq 10y_2 + s_5,\; x_3 \leq 9y_3 + s_5,\; x_4 \leq 7y_4 + s_5,\; x_5 \leq 3y_5 + s_5 \tag{124}$$

By projecting s_5 to zero, we obtain the classical flow cover inequality

$$x_2 + x_4 \leq 14 \;\; -(10-3)(1-y_2) - (7-3)(1-y_4) \tag{125}$$

using the cover $C = \{2, 4\}$ and the excess capacity of the cover $\lambda = 3$. We obtain finally the valid inequality

$$x_2 + x_4 \leq 14 \;\; -(10-3)(1-y_2) - (7-3)(1-y_4) \;\; + s_5 \tag{126}$$

by adding the flow s_5 at the right hand side of the inequality (125) to lift back the variable s_5, i.e. to make it valid even when $s_5 > 0$. The inequality (125) is violated by the fractional point in Figure 11.

Consider next another single node flow relaxation obtained by taking the linking capacity constraint (113) for a given period t, and the single item capacity constraints (113) for all items in the same period t. This gives the following single node flow relaxation

$$\sum_i (\alpha^i x^i_t) \leq L_t \tag{127}$$

$$(\alpha^i x^i_t) \leq (\alpha^i C^i_t) y^i_t \quad \text{for all } i \tag{128}$$

$$(\alpha^i x^i_t) \geq 0,\; y^i_t \in \{0,1\} \quad \text{for all } i \tag{129}$$

where the continuous flow variables are the variables $(\alpha^i x_t^i)$ with an indivudual upper bound of $(\alpha^i C_t^i)$.

Suppose now that we have solved the linear relaxation of MICLS and we want to cut off the fractional solution obtained and illustrated in Figure 12 for all items and for a given single period.

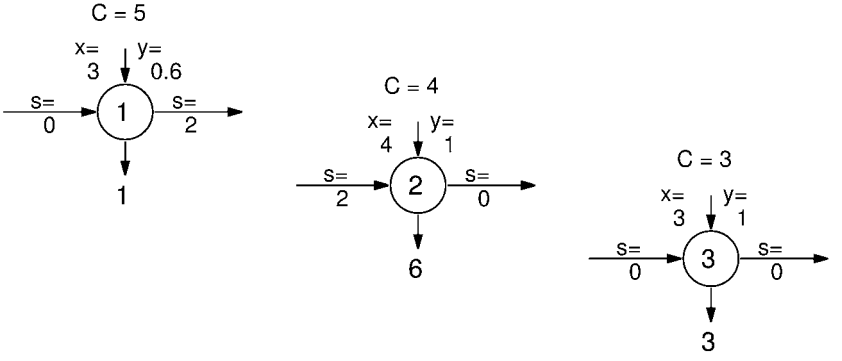


Fig. 12. A fractional solution for one period, 3 items in MICLS

The corresponding single node flow model is (where we have dropped the time index t for simplicity)

$$x^1 + 2x^2 + 3x^3 \leq 20 \tag{130}$$

$$x^1 \leq 5y^1,\ 2x^2 \leq 2(4y^2),\ 3x^3 \leq 3(3y^3). \tag{131}$$

We obtain the classical flow cover inequality

$$\begin{aligned} x^1 + 2x^2 + 3x^3 \leq 20 \ & -(5-2)(1-y^1) \\ & -(8-2)(1-y^2) - (9-2)(1-y^3) \end{aligned} \tag{132}$$

using the cover $C = \{1, 2, 3\}$ and the excess capacity of the cover $\lambda = 2$. This inequality is violated by the fractional point in Figure 12.

Big Buckets Multi-item Model with Setup Times. The classical and most studied MICLS problem as formulated in (110)-(114) does not contain any setup time, but setups are only taken into account through a penalty cost in the objective (110).

Therefore, this setup cost f_t^i must contain the opportunity cost of the setup activity for item i in period t. The opportunity cost is defined as the additional or marginal cost incurred because of the setup activity. It is very difficult to estimate a priori because it really depends on problem data, on capacity utilization and capacity available and productivity in period t. Hence, because of the difficulty

to estimate the setup cost , the true optimality of the solution to MICLS in not guaranteed.

Moreover, the setup activity consumes part of the capacity available, and influences the productivity. Therefore, it is not even guaranteed that the solution obtained from MICLS is really feasible because the capacity consumed by the setup activity is neglected.

In order to improve the reliability of the solution from MICLS, both in terms of feasibility and optimality, it is important to model more accurately the productivity, i.e. the capacity used as a function of the production lot sizes. In the presence of economies of scale in production, a better model is to introduce directly in the model the capacity used by the setup activity – and not only implicitely through the setup cost –. The linking capacity constraint in period t should then be modelled as

$$\sum_i (\alpha^i x_t^i) \; + \; \sum_i (\beta^i y_t^i) \leq L_t \tag{133}$$

where β^i represents the capacity used by the setup of item i, and is often called the setup time of item i. We call MICLSS the multi item model MICLS where the capacity constraint with setup times (133) replaces the original capacity constraint (113).

Very few optimzation methods have been proposed and tested for MICLSS. Trigeiro, Thomas, McClain [68] and Diaby, Bahl, Karwan, Zionts [17] test Lagrangean relaxation heuristic approaches – with performance guarantee – based on the solution of the ULS subproblems and the relaxation of the capacity linking constraints (133).

To tighten the basic formulation of MICLSS beyond the complete description of the ULS subproblems, other relaxations or subproblems have to be studied. Goemans [24] proposes generalized flow cover valid inequalities to tighten the formulation of the single period t multi item relaxation defined by

$$\sum_i (\alpha^i x_t^i) \; + \; \sum_i (\beta^i y_t^i) \leq L_t \tag{134}$$

$$0 \leq x_t^i \leq M y_t^i \quad \text{for all } i \tag{135}$$

$$y_t^i \in \{0,1\} \quad \text{for all } i \tag{136}$$

Constantino [13] provides a complete linear description of this single period relaxation when the setup times are constant for all items, i.e. $\beta^i = \beta$ for all i. These tightened formulations have been tested on multi-item single-period subproblems, but no tests of these improved formulations on multi-item problems with setup times (MICLSS) have been reported on in the literature.

Another single period relaxation of MICLSS that has been studied tries to combine the linking capacity constraints and the demand satisfaction for each item. It is formulated as

$$\sum_i (\alpha^i x_t^i) \; + \; \sum_i (\beta^i y_t^i) \leq L_t \tag{137}$$

$$s_{t-1}^i + x_t^i \geq d_t^i \quad \text{for all } i \tag{138}$$

$$0 \leq x_t^i \leq M y_t^i \quad \text{for all } i \tag{139}$$

$$s_{t-1}^i \geq 0, \; y_t^i \in \{0,1\} \quad \text{for all } i \tag{140}$$

Miller, Nemhauser and Savelsbergh [46] define cover and reverse cover valid inequalities for the one period subproblem (137)-(140), and tests their effectiveness in solving MICLSS instances by branch and cut.

4.3 Small Buckets Models

Small buckets models are used when changing the setup of the machines has a significant impact on the capacity used and/or on the production costs. Typically in these models, on each machine, we keep on producing the same item for a number of periods, and we only incur a setup cost/time when the setup of the machine is changed from one item to another.

When the capacity used or the cost incurred during the setup modification do not depend on the sequence of items, but on the fact that the machine setup is changed, we speak about startup time or startup cost models. When the cost or time lost during the setup depend on the sequence of products, i.e. on the products produced before and after the setup modification, we speak about change over costs and times.

We describe here the basic models, formulations and applications for such small buckets planning problems with startup costs and times, or change over costs and times.

Basic Small Buckets Multi-item Model, One Setup per Period. The natural and classical single machine formulation to represent the setup, startup and change over decisions in a small buckets planning model allows one to setup only one item per period. If the setup of the machine is changed from period $t-1$ to period t, this change is supposed to occur at the beginning of period t. This means that only one item can be produced in any period, the one for which the machine is setup in the period.

The variable y_t^i is used to represent the setup decision, it takes the value 1 when the machine is setup to produce item i during the whole period t, and 0 otherwise.

The variable z_t^i is the startup variable and takes the value 1 when the machine starts producing item i in period t, and was not setup for item i in period $t-1$, and 0 otherwise.

The variable w_t^{ij} is the change over variable and takes the value 1 when the machine strats producing item j in period t and was setup for item i in period $t-1$, and 0 otherwise.

Independently on the production levels and the demand satisfaction part of the problem, the natural formulation used to represent these setup, startup and change over decisions is the following

$$\sum_i y_t^i = 1 \quad \text{for all } t \tag{141}$$

$$z_t^i \geq y_t^i - y_{t-1}^i \quad \text{for all } i,t \tag{142}$$

$$w_t^{ij} \geq y_{t-1}^i + y_t^j - 1 \quad \text{for all } i,j,t \tag{143}$$

$$z_t^i,\ w_t^{ij},\ y_t^i \in \{0,1\} \quad \text{for all } i,j,t \tag{144}$$

where constraint (141) imposes to setup the machine for exactly one item per period, constraint (142) forces the startup of item i in period t when $y_t^i = 1$ and $y_{t-1}^i = 0$, constraint (143) forces the change over from item i in period $t-1$ to item j in period t when $y_{t-1}^i = y_t^j = 1$.

To obtain a complete formulation of the small buckets planning problem, classical flow conservation constraints are used to model the satisfaction of demand, as well as capacity constraints. In a model with startup time γ^i to start producing item i, the production capacity constraints are formulated as

$$x_t^i \leq U_t^i y_t^i \ - \gamma^i z_t^i \quad \text{for all } i,t. \tag{145}$$

In a model with change over time γ^{ji} for switching from j to i, the production capacity constraint are formulated as

$$x_t^i \leq U_t^i y_t^i \ - \sum_{j:j\neq i} \gamma^{ji} w_t^{ji} \quad \text{for all } i,t. \tag{146}$$

Several optimization approaches have been proposed in the literature for solving small buckets constant capacity over time (but different for each item) multi item lot sizing problems. Karmarkar and Scharge [31] propose, and test on small instances, a Lagrangean relaxation and branch and bound approach for the model with startup costs. Wolsey [81] solve problems with change over costs by branch and bound using the extended unit flow reformulation (see below) to model the setup decisions. Constantino [12] solve larger instances, with up to 5 items and 36 periods, of problems with startup costs using a branch and cut approach. Vanderbeck [69] solve problems with startup times using a column generation approach combined with cut generation.

Unit Flow Reformulation. The formulation (141)-(144) used to model the steup decisions can be tightened and reformulated as a network flow problem consisting of sending one unit of flow from period 1 to period n. The flow models the status of the machine through the time priods. This formulation is the tightest possible to represent the machine status decisions in the sense that the constraint matrix is totally unimodular. This formulation proposed in Karmarkar and Schrage [31] has been tested in Wolsey [81]. It is defined by

$$\sum_i y_0^i = 1 \tag{147}$$

$$\sum_j w_t^{ij} = y_{t-1}^i \quad \text{for all } i,t \tag{148}$$

$$z_t^j + w_t^{jj} = y_t^j \quad \text{for all } j,t \tag{149}$$

$$\sum_{i:i\neq j} w_t^{ij} - z_t^j = 0 \quad \text{for all } j,t \tag{150}$$

$$z_t^j,\ w_t^{jj},\ y_t^j \in \{0,1\} \quad \text{for all } i,j,t \tag{151}$$

where constraint (147) defines the initial (period 0) status of the machine – usually these variables have a fixed value corresponding to the current status of the machine–. Constraint (148) stipulates that if the machine was setup for i in period $t-1$, then it must switch from i to some j (possibly equal to i) in period t. Constraint (149) expresses that if the machine is setup for j in period t, then either it was already setup for j in period $t-1$, or it starts producing item j in period t. Constraint (150) says that if the machine starts producing item j in period t, it must switch from a different product in period $t-1$.

Alternative formulations for modelling changeovers in production planning and scheduling problems are surveyed in Wolsey [82].

Models with Two Setups per Period. Haase [27] has proposed a more flexible interpretation of the setup decisions. Suppose that $y_t^i = 1$ if the machine is setup for item i at the end of period t, and 0 otherwise. The natural formulation (141)-(144), or the unit flow formulation (147)-(151), can still be used to model the sequence of admissible setup decisions. But with this interpretation of the setup variables, it is now possible to produce two items in each period t. These are the items for which the machine is setup at the end of period $t-1$ and t. If any, the setup modification occurs now during period t between the production lots of the two products. So we only need to reformulate the capacity constraints. For instance, in the case of startup times, this gives

$$x_t^i \leq U_t^i(y_{t-1}^i + y_t^i - w_t^{ii}) \ - \gamma^i z_t^i \quad \text{for all } i,t \tag{152}$$

$$\sum_i (\alpha^i x_t^i) \ + \ \sum_i (\gamma^i z_t^i) \leq L_t \tag{153}$$

Kimms [32] solves multi item small buckets problems with two setups per period using a Lagrangean relaxtion approach.

Sequencing Models or m-Setups per Period. As a final extension of the small buckets models, we can mention the pure batch sequencing problem, or problems with many setups per period. In this case, we still need to know the sequence of products in order to compute the change over times, but we may produce in each period as many items or batches as feasible given the available capacity.

The natural formulation (141)-(144), or the unit flow formulation (147)-(151), can still be used to model the sequence of admissible setup decisions inside each period, but with a different interpretation of the index t as the batch number rather than the period number. For instance, $y_t^i = 1$ means that item i is the item produced during the t^{th} batch.

An alternative formulation in this case is to use the asymmetric traveling salesman problem (ATSP) to formulate the sequence of setup decisions inside each period.

Several papers in the literature report on such applications and propose optimization approaches for similar problems. Kang, Malik and Thomas [30] propose and test a column generation approach to solve a multi period small buckets sequencing problem with change over costs, and compare with a direct branch and bound approach using the ATSP formulation for the setup decisions. Batta and Teghem [8] solve a multi item single period problem with change over times by branch and bound, again using an ATSP formulation.

5 Formulations of Multi-level Models

We study in this section a general class of multi level production planning models. First we recall the description of the multi level lot sizing problem from Section 2.2, and analyze the classical level by level decompostion approach to show that it is suboptimal. Then we survey the optimzation approaches from the literature. Finally we present the echelon stock reformulation that plays an important role in all optimization approaches published so far.

5.1 Multi-level Models and the Decomposition Approach

We recall first the formulation of the general product structure capacitated multi-level lot-sizing model from Section 2.2 (see for instance Billington, McClain and Thomas [9]). The product structures are classified into Series, Assembly or General structures, as represented in Figure 4.

$$\min \sum_i \sum_t \{p_t^i x_t^i + f_t^i y_t^i + h_t^i s_t^i\} \tag{154}$$

$$s_{t-1}^i + x_{t-\gamma^i}^i = [d_t^i + \sum_{j \in S(i)} r^{ij} x_t^j] + s_t^i \quad \text{for all } i,t \tag{155}$$

$$x_t^i \leq M y_t^i \quad \text{for all } i,t \tag{156}$$

$$\sum_i \alpha^{ik} x_t^i + \sum_i \beta^{ik} y_t^i \leq L_t^k \quad \text{for all } t,k \tag{157}$$

$$x_t^i,\ s_t^i \geq 0,\ y_t^i \in \{0,1\} \quad \text{for all } i,t \tag{158}$$

A classical approach used for solving multi level planning problems is to decompose the problem level by level into single level subproblems, solved sequentially from end-products to raw materials. The production plans at each level define the demand at subsequent levels. This is the approach used in MRP-type systems. This separate optimization at each level yields suboptimal production plans. This sequential planning procedure, and its suboptimality, are illustrated in the following simple example.

Suppose that we have a 3 periods lot-sizing problem with 2 levels, one item at each level, and a serial product structure where one unit of the raw material ($i = 2$) is required to produce one unit of the finished product($i = 1$). For simplicity, we assume that the lead time γ^i in zero for each item. The external or independent demand for the finished product is $d^1 = (10, 15, 20)$. There is no external demand for the raw material. There is a fixed ordering cost of $f_t^2 = 200$ for the raw material, and fixed production cost of $f_t^1 = 100$ for the finished product, for all t. There is no unit production cost. The inventory cots is $h_t^i = 5$, for all i, t. This planning problems corresponds to the fixed charge minimum cost network flow problem represented in Figure 13.

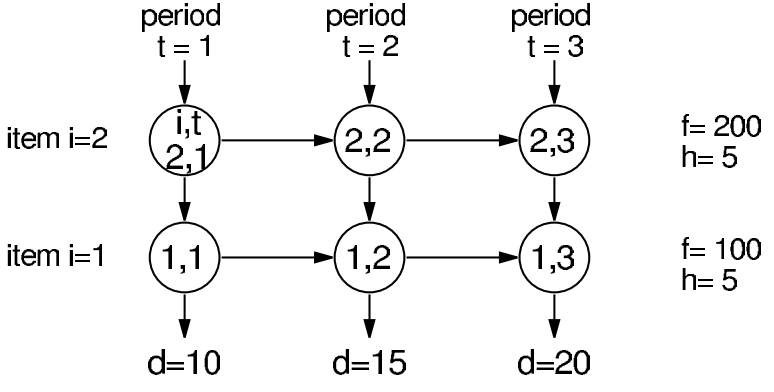


Fig. 13. A 2-level serial production planning example

In the classical decomposition approach, we plan first the production of the finished product in order to satisfy its external demand, by solving the corresponding single item ULS problem. The optimal solution with a cost of 275 is to produce 25 units in period 1, stock 15 from period 1 to period 2, and produce 20 units in period 3.

This production plan defines the internal or dependent demand for the raw material, 25 units of raw material have to be available in period 1, and 20 units have to be available in period 3. We then solve the ULS subproblem for the raw material, and find that it is optimal at a cost of 400 to order 45 units in period 1, and stock 20 units from period 1 to period 3. Note that an alternate optimal solution for the raw material is to order twice (25 units in period 1 and 20 units in period 3), and avoid the inventory costs.

So, we have obtained a global (finished product and raw material) production plan with a cost of 675. The optimal solution with a cost of 575 of this network flow problem is to order 45 units of raw material in period 1, transform these units directly into finished product in period 1, and stock the finished product to satisfy demand in periods 2 and 3. Furthermore, at the first step of the sequential procedure, if you enumerate all extreme solutions (those satisfying satisfying the decomposition property in Theorem 1) of the ULS subproblem for the finished product, then the worst extreme solution (which is to produce the 45 units in period 1, and to stock to periods 2 and 3) of this subproblem should be selected in order to find the globally optimal solution.

We have illustrated with this naive example, that it seems very difficult to optimize the production plans by solving the single level subproblems sequentially. Therefore, we need to solve integrated models, as (154)-(158), taking the planning decisions at all levels simultaneously, in order to optimize the quality of the production plans.

5.2 Optimization of Multi-level Models

We review briefly the complexity results known, and the optimization approaches used to solve various multi level planning problems.

The uncapacitated problem (without constraint (157)) with a serial product structure (see Figure 4) is the simplest problem in the class. It can be solved in polynomial time by dynamic programming (Zangwill [85] and Love [41]) using the same decomposition property as for ULS, and the technique of Martin [44] can be used to transform this dynamic program into a compact (i.e. polynomial number of variables and constraints, in the number of items and time periods) linear extended reformulation of the problem (see Pochet [52]). The solution of this problem using a branch and cut approach, and a partial description of the convex hull of solutions, is presented in Pochet [52].

The next problem studied in the literature is the uncapacitated problem with an assembly product structure (see Figure 4). The computational complexity of

this problem is still an open question. Afentakis, Gavish and Karmarkar [1] have solved problems with up to 50 items using a Lagrangean relaxation approach based on the echelon stock reformulation and combined with a specialized branch and bound algorithm.

The uncapacitated problem with general product structure has been tackled by Afentakis and Gavish [2] using a Lagrangean relaxation approach, and by Pochet and Wolsey [55] using a cut and branch approach (i.e. cuts are added only at the first node of the branch and bound tree, before any branching).

The full problem (154)-(158) with general product structure, capacities and setup times, has been addressed by Tempelmeier and Derstroff [66] to obtain heuristic solutions with performance guarantee using a Lagrangean relaxation approach, and by Stadtler [65] using a branch and bound approach and the extended linear reformulations for the ULS subproblems.

All the optimization applications mentioned for assembly and general product structures, using Lagrangean relaxation, branch and bound or branch and cut approaches, are based on the same reformulation of the problem. First, problem (154)-(158) is reformulated using the echelon stock concept. This transforms the initial formulation into a series of ULS subproblems linked by capacity constraints and by product structure constraints (see Section 5.3 below). Then, the ULS part of the problem is reformulated in the branch and bound or branch and cut approaches, or the linking constraints are relaxed in the Lagrangean relaxation approach.

Better relaxations are available, but not yet used for solving complex capacitated multi-level problems. For instance, the reformulations and valid inequalities found for big buckets capacitated models with setup times could be used to tighten the formulations further. Valid inequalities for fixed charge network flow problems could be generated on the multi-level network flow problem (see Van Roy and Wolsey [74]).

5.3 The Echelon Stock Reformulation

The formulation (154)-(158) of the general multi-level planning problem does not contain explicitely the single item ULS subproblems (except for finished products) because of the presence of both dependent and independent demand for the items. By using the concept of echelon stock, introduced by Clark and Scarf [11], one can obtain a reformulation that contains explicitely the ULS subproblems. This allows one then to design decomposition optimization algorithms based on our knowledge on how to solve and reformulate ULS.

For simplicity, we assume that the lead times γ^i are equal to zero in (155). The echelon stock E_t^i of item i in period t is the total stock of component i in the production system, as item i or in items appearing further on in the production process. It can be defined as

$$E_t^i = s_t^i + \sum_{j \in S(i)} r^{ij} E_t^j \tag{159}$$

because, in addition to the stock s_t^i of items i as items i, there are r^{ij} units of items i present in each unit of stock of item j.

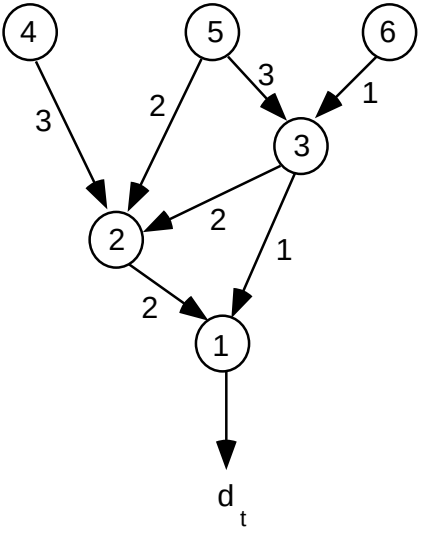


Fig. 14. A general product structure

Figure 14 gives an example of general product structure. For this example, the echelon stocks of the different items are

$$E_t^1 = s_t^1 \tag{160}$$
$$E_t^2 = s_t^2 + 2E_t^1 = s_t^2 + 2s_t^1 \tag{161}$$
$$E_t^3 = s_t^3 + 2E_t^2 + E_t^1 = s_t^3 + 2s_t^2 + 5s_t^1 \tag{162}$$
$$E_t^4 = s_t^4 + 3E_t^2 = s_t^4 + 3s_t^2 + 6s_t^1 \tag{163}$$
$$E_t^5 = s_t^5 + 2E_t^2 + 3E_t^3 = s_t^5 + 3s_t^3 + 8s_t^2 + 19s_t^1 \tag{164}$$
$$E_t^6 = s_t^6 + E_t^3 = s_t^6 + s_t^3 + 2s_t^2 + 5s_t^1 \tag{165}$$

We can use equation (159) to define k^{ij} as the number of items i present in each unit of item j, for all i, j (i.e. $E_t^i = \sum_j k^{ij} s_t^j$), and to replace the natural stock variables s_t^i by the echelon stock variables E_t^i in formulation (154)-(158). This gives the formulation

$$E_{t-1}^i + x_t^i = \sum_j k^{ij} d_t^j + E_t^i \text{ for all } i, t \tag{166}$$

$$x_t^i \leq M y_t^i \quad \text{for all } i,t \tag{167}$$

$$E_t^i - \sum_{j \in S(i)} r^{ij} E_t^j \geq 0 \quad \text{for all } i,t \tag{168}$$

$$\sum_i \alpha^{ik} x_t^i + \sum_i \beta^{ik} y_t^i \leq L_t^k \quad \text{for all } t,k \tag{169}$$

$$x_t^i,\ E_t^i \geq 0,\ y_t^i \in \{0,1\} \quad \text{for all } i,t \tag{170}$$

where (168) is the product structure linking constraint coming from the non negativity of the natural stock variables, (169) is the capacity linking constraint, (166)-(167) define independent ULS subproblems, one for each item.

6 New Directions

We conclude in this Section by giving a personal view on some new directions to be investigated in modeling production planning problems. These include better productivity models, and in particular better models for capacity utilization and setup times, new models to represent the product structure – or recipes – in process industries, and the study of continuous time planning and scheduling models as opposed to the discrete time models studied in this review. As pointed out in Kuik et al [35], such extensions are crucial for the vitality of the production planning and lot-sizing research.

We also define some challenges for the future of this research field.

6.1 Setup Times, Capacity Utilization, and Productivity Models

We have already mentioned in Section 4.2 that the optimal solutions of multi-item capacitated lot-sizing models (MICLS) with setup cost may well correspond to infeasible and/or suboptimal in reality. The main difficulty there was to estimate a priori the setup cost as the opportunity cost of the capacity used during setup. This was the main reason to study similar capacitated models, but with setup times instead of setup costs, using constraint 133 to model capacity utilization.

For such capacitated models with setup times, a first direction of research is to investigate the polyhedral structure of multi-item subproblems involving both capacity utilization and demand satisfaction, such as Miller [46]. This will allow one to obtain new classes of valid inequalities and to solve problems using better relaxations than the one obtained from the reformulation of the simple uncapacitated subproblems.

Another extension is to develop new capacity utilization or productivity models, and to study the corresponding formulations.

– In classical MRP databases, the routing of an item is defined as a fixed sequence of processing or production steps, each performed in a specific workcenter (i.e. on a specific resource), requiring a fixed setup time and a variable processing time per unit produced. This capacity utilization model can be introduced in the capacitated multi-level lot-sizing formulation (154)-(158) using $\alpha^{ik\Delta}$ and $\beta^{ik\Delta}$ to represent respectively the unit processing time and fixed setup time for item i, on resource k, in the Δ^{th} processing period, for $\Delta = 1, \cdots, \gamma^i$. The capacity constraint (157) becomes then

$$\sum_i \sum_{l=t-\gamma^i+1}^{t} \alpha^{ik,t-l+1} x_l^i + \sum_i \sum_{l=t-\gamma^i+1}^{t} \beta^{ik,t-l+1} y_l^i \leq L_t^k \quad \text{for all } t,k \quad (171)$$

– Product similarities or product families also have some influence on the setup and change over times. On a specific resource or machine, switching from one product family to another requires more time than switching from one product to another product of the same family. Van de Velde [70] suggested new capacity utilization models to better reflect such situations, see Simpson, Erenguc [64].
– In activity based costing and management systems (ABC/ABM), hierarchical models for activities and costs are used to better approximate the (production) cost structure. These can be modelled using joint or hierarchical setup cost models, as in Degraeve and Roodhooft [16].

6.2 Process Industry: Products and Operations Structure

The multi-level MRP-II model, and its typical formulation (154)-(158), should be generalized in order to be applicable to other processes than discrete parts manufacturing. In particular, for chemical or process industries, production planning and sequencing problems require some generalization of BOM structures to allow the modelling of **OPERATIONS** consuming items as **INPUTS** producing items as **OUTPUTS** in fixed or variable proportions and requiring some amount of **RESOURCES**. The main difference with respect to discrete BOMs is that several output items or co-products are generally produced simultaneously in a single operation, see Westenberger, and Kallrath [79], Kondili, Pantelides and Sargent [33].

The basic example –and formulation– of an operations model is

$$s_{t-1}^i + \sum_o \rho_{OUT}^{oi}\, x_{t-\gamma^{oi}}^o = [d_t^i + \sum_o \rho_{IN}^{oi}\, x_t^o] + s_t^i \quad \text{for all } i,t \quad (172)$$

$$LB_t^o y_t^o \leq x_t^o \leq UB_t^o y_t^o \quad \text{for all } o,t \quad (173)$$

$$x_t^o,\ s_t^i \geq 0,\ y_t^o \in \{0,1\} \quad \text{for all } i,o,t \quad (174)$$

where the index o is used to model operations, x_t^o represents the amount of time spent on performing operation o in period t, and $y_t^o = 1$ indicates that operation

o is carried out in period t. The extended BOM structure is defined by ρ^{oi}_{OUT} and ρ^{oi}_{IN} representing respectively the number of units of item i produced and consumed per time unit of operation o, for all operations o and items i. The data γ^{oi} represents the lead time needed for operation o to produce the output item i.

Many of the reformulation techniques presented in this paper can be used to tighten the formulation of this operations planning model.

6.3 Continuous Time Scheduling Models

In production planning, small buckets models are used when there is a need (cost, capacity, ...) to represent the detailed sequence and timing of events (such as production start, production end, empty or full storage tank, ...) through time. Very often in such models, the period time lengths are very small in order to represent these events at precise moments in time, but the number of events to occur in the planning horizon is small. To avoid to have too many time periods, and too large models, continuous time models are used. They use variable period lengths in order to represent the events in time, see Zhang and Sargent [86], Pinto and Grossman [51], Pochet, Tahmassebi, Wolsey [59].

Consider an example planning problem where a number of items – indexed by i or j – have to be produced on a set of independent parallel production lines – indexed by k. There is a demand of d^i units of item i to meet, and the production rate of item i on machine k is ρ^{ik} units of i per unit production time. When there is a change over on machine k from item i to item j, c^{ijk} units of time are lost, and L^k units of time are available on line k during the planning horizon.

A continuous time formulation for this model is

$$\sum_k \rho^{ik}\ (b^{ik} - a^{ik}) \ = \ d^i \quad \text{for all } i \tag{175}$$

$$\sum_i (b^{ik} - a^{ik}) \ + \ \sum_{i,j} \ c^{ijk}\ w^{ijk} \leq L^k \quad \text{for all } k \tag{176}$$

$$a^{jk} \geq b^{ik} + c^{ijk}\ w^{ijk} \ - \ M(y^{ik} - w^{ijk}) \quad \text{for all } i,j,k \tag{177}$$

$$0 \leq (b^{ik} - a^{ik}) \leq [d^i/\rho^{ik}]\ y^{ik} \quad \text{for all } i,k \tag{178}$$

$$\sum_j w^{0jk} = 1 \quad \text{for all } k \tag{179}$$

$$\sum_i w^{i0k} = 1 \quad \text{for all } k \tag{180}$$

$$w^{0jk} + \sum_i w^{ijk} = y^{jk} \quad \text{for all } j,k \tag{181}$$

$$w^{i0k} + \sum_j w^{ijk} = y^{ik} \quad \text{for all } i,k \tag{182}$$

$$a^{ik}, b^{ik} \geq 0, \quad y^{ik}, w^{ijk} \in \{0,1\} \quad \text{for all } i,j,k \tag{183}$$

where variable a^{ik} is the start time of item i on line k, variable b^{ik} is the end time of item i on line k, variable y^{ik} takes the value 1 if item i is produced on line k, and variable w^{ijk} equals 1 if there is a change over from item i to item j on line k. Item 0 is a fictive item used to start and finish the sequence on each machine. Constraint (175) imposes demand satisfaction, constraint (176) is the line capacity constraint, constraint (177) imposes the starting time of item j when it follows directly item i on line k, constraint (178) fixes the value of the setup variables, (179)–(182) model the sequencing constraints (i.e. each line must have a starting item, a finishing item, each item must have a predecessor, and a successor).

6.4 Challenges

As a conclusion, we summarize some key challenges for the future of this research field

- The multi-item production planning model (154)-(158) with general product structure, capacities and setup times must be solved in practice. This means that specific optimization codes have to be developed, that heuristic algorithms based on linear programming relaxations must be designed in order to produce good solutions for large instances, and fast and specialized algorithms for solving the large size LP relaxations have to be developed.
- Other more general models have to be constructed in order to come closer to interesting and real planning problems. This includes new capacity utilization models, joint setup times and costs models, operations planning and scheduling models.
- The models have to be changed when the limits of such deterministic discrete time planning problems have been reached. This includes the development and resolution of continuous time scheduling models, as well as of stochastic models.

References

1. P. Afentakis, B. Gavish and U. Karmarkar, "Computationally efficient optimal solutions to the lot-sizing problem in multistage assembly systems", Management Science 30(2), 222–239, 1984.
2. P. Afentakis and B. Gavish, "Optimal lot-sizing algorithms for complex product structures", Operations Research 34, 237–249, 1986.
3. A. Aggarwal and J. Park, "Improved algorithms for economic lot-size problems", Operations Research 41, 549–571, 1993.
4. A. Agra and M. Constantino, "Lotsizing with backlogging and start-ups: the case of Wagner-Whitin costs", Operations Research Letters 25 (2), 81–88, 1999.
5. R.N. Anthony, "Planning and control systems: a framework for analysis", Harvard University Press, Cambridge, Mass., 1965.
6. I. Barany, T.J. Van Roy and L.A. Wolsey, "Uncapacitated lot sizing: the convex hull of solutions", Mathematical Programming Study 22, 32–43, 1984.

7. I. Barany, T.J. Van Roy and L.A. Wolsey, "Strong formulations for multi-item capacitated lot-sizing", Management Science 30, 1255–1261, 1984.
8. C. Batta and J. Teghem, "Optimization of production scheduling in plastics processing industry", Jorbel (Belgian Journal of operations research, statistics and computer science) 34 (2), 55–78, 1994.
9. P.J. Billington, J.O. McClain, L.J. Thomas, "áMathematical Programming approaches to capacity constrained MRP systemsá: review, formulation and problem reductioná", Management Science 29 (10), 1126–1141, 1983.
10. W.-H. Chen and J.-M. Thizy, "Analysis of relaxations for the multi-item capacitated lot-sizing problem", Annals of Operations Research 26, 29–72, 1990.
11. A.J. Clark and H. Scarf, "Optimal policies for multi echelon inventory problems, Management Science 6, 475–490, 1960.
12. M. Constantino, "A cutting plane approach to capacitated lot-sizing with start-up costs", Mathematical Programming 75 (3), 353–376, 1996.
13. M. Constantino, "Lower bounds in lot-sizing models: A polyhedral study", Mathematics of Operations Research 23 (1), 101–118, 1998.
14. W.B. Crowston, M.H. Wagner, "Dynamic lot size models for multi stage assembly systems", Management Science 20(1), 14–21, 1973.
15. W.B. Crowston, M.H. Wagner, J.F. Williams, "Economic lot size determination in multi stage assembly systems", Management Science 19(5), 517–527, 1973.
16. Z. Degraeve, F. Roodhooft, "Improving the efficiency of the purchasing process using total cost of ownership informationá: The case of heating electrodes at Cockerill-Sambre S.A.á", European Journal of Operational Research 112, 42–53, 1999.
17. M. Diaby, H.C. Bahl, M.H. Karwan and S. Zionts, "A Lagrangean relaxation approach to very large scale capacitated lot-sizing", Management Science 38 (9), 1329–1340, 1992.
18. S.E. Elmaghraby, "The economic lot-scheduling problem (ELSP): reviews and extensions", Management Science 24, 587–598, 1978.
19. G.D. Eppen and R.K. Martin, Solving multi-item lot-sizing problems using variable definition, Operations Research 35, 832–848, 1987.
20. A. Federgrun and M. Tsur, "A simple forward algorithm to solve general dynamic lot-size models with n periods in $O(nlogn)$ or $O(n)$ time", Management Science 37, 909–925, 1991.
21. B. Fleischmann, "The discrete lotsizing and scheduling problem", European Journal of Operational Research 44(3), 337–348, 1990.
22. B. Fleischmann, "The discrete lotsizing and scheduling problem with sequence-dependent setup costs", European Journal of Operational Research,1994.
23. M. Florian and M. Klein, "Deterministic production planning with concave costs and capacity constraints", Management Science 18, 12–20, 1971.
24. M.X. Goemans, "Valid inequalities and separation for mixed 0-1 constraints with variable upper bounds", Operations Research Letters 8, 315–322, 1989.
25. M. Grotschel, L. Lovasz and A. Schrijver, "The ellipsoid method and its consequences in combinatorial optimization", Combinatorica 1, 169–197, 1981.
26. O. Gunluk and Y. Pochet, "Mixing mixed integer rounding inequalities", CORE discussion paper 9811, Université catholique de Louvain, Belgium, 1998. (to appear in Mathematical Programming)
27. Haase, "Lotsizing and scheduling for production planning", Lecture notes in economics and mathematical systems 408, Springer, Berlin, 1994.
28. F.W. Harris, "How many parts to make at once", Factory, the Magazine of Management 10(2), 1913.

29. A.C. Hax and H.C. Meal, "Hierarchical integration of production planning and scheduling", in M. Geisler editor, TIMS studies in Management Science, chapter 1, North Holland/American Elsevier, New York, 1975.
30. S. Kang, K. Malik, L.J. Thomas, "Lotsizing and scheduling in parallel machines with sequence dependent setup costs", Working paper 97-07, Johnson Graduate school of Management, Cornell university, 1997.
31. U.S. Karmarkar and L. Schrage, "The deterministic dynamic product cycling problem", Operations Research 33, 326–345, 1985.
32. A. Kimms, "Multi-level lot sizing and scheduling: methods for capacitated, dynamic and deterministic models", Physica-Verlag (production and logistics series), Heidelberg, 1997.
33. E. Kondili, C.C. Pantelides, R.W.H. Sargent, "A general algorithm for short-term scheduling of batch operations - 1. MILP formulation", Computers Chemical Engineering 17, 211–227, 1993.
34. J. Krarup and O. Bilde, "Plant location, set covering and economic lot sizes: an $O(mn)$ algorithm for structured problems", in "Optimierung bei Graphentheoretischen und Ganzzahligen Probleme", L. Collatz et al. eds, Birkhauser Verlag, Basel, 155–180, 1977.
35. R. Kuik, M. Salomon and L.N. van Wassenhove, "Batching decisions: structure and models", European Journal of Operational Research 75, 243–263, 1994.
36. L.S. Lasdon and R.C. Terjung, "An efficient algorithm for multi-item scheduling", Operations Research 19, 946–969, 1971.
37. J. Leung, T.M. Magnanti and R. Vachani, "Facets and algorithms for capacitated lot-sizing", Mathematical Programming 45, 331–359, 1989.
38. M. Loparic, Y.Pochet and L.A. Wolsey, "Uncapacitated lot-sizing with sales and safety stocks", Mathematical Programming 89 (3), 487–504, 2001.
39. M. Loparic, H. Marchand and L.A. Wolsey, "Dynamic knapsack sets and capacitated lot-sizing", CORE discussion paper 2000/47, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2000.
40. L. Lovasz, "Graph theory and integer programming", Annals of Discrete Mathematics 4, 141–158, 1979.
41. S.F. Love, "A facilities in series inventory model with nested schedules", Management Science 18(5), 327–338, 1972.
42. T.M. Magnanti and R. Vachani, "A strong cutting plane algorithm for production scheduling with changeover costs", Operations Research 38, 456–473, 1990.
43. H. Marchand and L.A. Wolsey, "The 0-1 knapsack problem with a single continuous variable", Mathematical Programming 85, 15–33, 1999.
44. R.K. Martin, "Generating alternative mixed-integer programming models using variable redefinition", Operations Research 35, 331–359, 1987.
45. R.K. Martin, "Using separation algorithms to generate mixed integer model reformulations", Operations Research Letters 10, 119–128, 1991.
46. A.J. Miller, G.L. Nemhauser and M.W.P. Savelsbergh, "On the polyhedral structure of a multi-item production planning model with setup times", CORE discussion paper 2000/52, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2000.
47. G.L. Nemhauser and L.A. Wolsey, "Integer and combinatorial optimization", Wiley, New York, 1988.
48. G.L. Nemhauser and L.A. Wolsey, "A recursive procedure for generating all cuts for 0-1 mixed integer programs", Mathematical Programming 46, 379–390, 1990.

49. J. Orlicky, "Material Requirements planning", McGraw-Hill, New York, 1975.
50. M.W. Padberg, T.J. Van Roy and L.A. Wolsey, "Valid inequalities for fixed charge problems", Operations Research 33, 842–861 , 1985.
51. Pinto and Grossmann, "A logic-based approach to scheduling problems with resource constraints", Computers and Chemical Engineering 21 (8), 801–818, 1997.
52. Y. Pochet, "Lot-sizing problems : reformulations and cutting plane algorithms", PhD Thesis, Université Catholique de Louvain, Belgium, 1987.
53. Y.Pochet, "Valid inequalities and separation for capacitated economic lot-sizing", Operations Research Letters 7, 109–116, 1988.
54. Y.Pochet and L.A. Wolsey, "Lot-size models with backlogging: Strong formulations and cutting planes", Mathematical Programming 40, 317–335, 1988.
55. Y.Pochet and L.A. Wolsey, "Solving multi-item lot sizing problems using strong cutting planes", Management Science 37, 53–67, 1991.
56. Y.Pochet and L.A. Wolsey, "Lot-sizing with constant batches: Formulation and valid inequalities", Mathematics of Operations Research 18, 767–785, 1993.
57. Y.Pochet and L.A. Wolsey, "Polyhedra for lot-sizing with Wagner-Whitin costs", Mathematical Programming 67, 297–323, 1994.
58. Y.Pochet and L.A. Wolsey, "Algorithms and reformulations for lot-sizing problems", DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 20, 245-293, 1995.
59. Y. Pochet, T. Tahmassebi and L.A. Wolsey, "Reformulation of a single stage packing model", Report, Memips: Esprit project 20118, June 1996.
60. R.L. Rardin and U. Choe, "Tighter relaxations of fixed charge network flow problems", report J-79-18, Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, Georgia, 1979.
61. R. Rardin and L.A. Wolsey, "Valid inequalities and projecting the multicommodity extended formulation for uncapacitated fixed charge network flow problems", European Journal of Operations Research, Nov 1993.
62. M. Salomon, "Deterministic lotsizing models for production planning", PhD. Thesis, Erasmus Universiteit Rotterdam, The Netherlands, 1990.
63. M. Salomon, L.G. Kroon, R. Kuik, L.N. Van Wassenhove, "Some extensions of the discrete lotsizing and scheduling problem", Management Science 37(7), 801–812, 1991.
64. N.C. Simpson, S.S. Erenguc, "Production planning in multiple stage manufacturing environments with joint costs, limited resources nad set-up times", Technical report, Department of Management Science and Systems, University of Buffalo, 1998.
65. H. Stadtler, "Mixed integer programming model formulations for dynamic multi-item multi-level capacitated lotsizing", European Journal of Operational Research 94 (3), 561–581, 1996.
66. H. Tempelmeier and M. Derstroff, "A Lagrangean-based heuristic for dynamic multilevel multiitem constrained lotsizing with setup times", Management Science 42 (5), 738–757, 1996.
67. J.M. Thizy and L.N. Van Wassenhove "Lagrangean relaxation for the multi-item capacitated lot-sizing problem: a heuristic implementation", IIE Transactions 17 (4), 308–313, 1985.
68. W. Trigeiro, L.J. Thomas and J.O. McClain, "Capacitated lot sizing with setup times", Management Science 35(3), 353–366, 1989.
69. F. Vanderbeck, "Lot-sizing with start up times", Management Science 44 (10), 1409–1425, 1998.

70. W. Van de Velde, Private communication, 1997.
71. C.P.M. van Hoesel, "Models and algorithms for single-item lot sizing problems", Ph.D. Thesis, Erasmus Universiteit, Rotterdam, 1991.
72. S. van Hoesel, A. Wagelmans and L.A. Wolsey, "Economic lot-sizing with start-up costs: the convex hull", SIAM Journal of Discrete Mathematics, 1994.
73. S. van Hoesel and A. Wagelmans , "An O(T-3) algorithm for the economic lot-sizing problem with constant capacities", Management Science 42 (1), 142–150, 1996.
74. T.J. Van Roy and L.A. Wolsey, "Valid inequalities and separation for uncapacitated fixed charge networks", Operations Research Letters 4, 105–112,1985.
75. A.F. Veinott, "Minimum concave cost solution of Leontief substitution models of multi-facility inventory systems", Operations Research 17(2), 262–291, 1969.
76. T.E. Vollman, W.L. Berry, and D.C. Whybark, "Manufacturing Planning and Control Systems", Third Edition, Richard D. Irwin., 1997.
77. A.P.M. Wagelmans, C.P.M. van Hoesel and A.W.J. Kolen, "Economic lot-sizing: an $O(nlogn)$ algorithm that runs in linear time in the Wagner-Whitin case", Operations Research 40, Supplement 1, 145–156, 1992.
78. H.M. Wagner and T.M. Whitin, "Dynamic version of the economic lot size model", Management Science 5, 89–96, 1958.
79. H. Westenberger and J. Kallrath, "Formulation of a jobshop problem in process industry", Preprint, Bayer, Leverkusen, January 1995.
80. R.H. Wilson, "A scientific routine for stock control", Harvard Business Review 13, 1934.
81. L.A. Wolsey, "Uncapacitated lot-sizing problems with start-up costs", Operations Research 37, 741–747, 1989.
82. L.A. Wolsey, "MIP modelling of changeovers in production planning and scheduling problems", European Journal of Operational Research 99, 154–165, 1997.
83. L.A. Wolsey, "Integer programming", Wiley, New York, 1999.
84. W.I. Zangwill, "Minimum concave cost flows in certain networks", Management Science 14, 429–450, 1968.
85. W.I. Zangwill, "A backlogging model and a multi-echelon model of a dynamic economic lot size production system – a network approach", Management Science 15(9), 506–527, 1969.
86. X. Zhang and R.W.H. Sargent, "A new unified formulation for process scheduling", AIChE annual meeting, Paper 144c, St Louis, Missouri, 1993.

Lagrangian Relaxation

Claude Lemaréchal

Inria, 655 avenue de l'Europe, Montbonnot, 38334 Saint Ismier, France
Claude.Lemarechal@inrialpes.fr

Abstract. Lagrangian relaxation is a tool to find upper bounds on a given (arbitrary) maximization problem. Sometimes, the bound is exact and an optimal solution is found. Our aim in this paper is to review this technique, the theory behind it, its numerical aspects, its relation with other techniques such as column generation.

1 Introduction, Motivation

1.1 Scope of the Paper

One of the basic techniques in combinatorial optimization is *bounding*: given a (finite but) nasty set $S \subset \mathbb{R}^n$ and an "easy" function f (say linear or quadratic), find an upper bound for the optimal value of the problem

$$\max f(x)\,, \quad x \in S\,.$$

Lagrangian relaxation is a universal technique for that. To make it applicable, the feasible set S must be written as $S = \mathcal{X} \cap \{x : c(x) = 0\}$, where $\mathcal{X}$ is "easy" in terms of maximizing f over it, while the constraints $c = (c_1, \ldots, c_m)$ are "complicating". *In fine*, the technique produces a bunch of optimal solutions to the problem $\max_{x \in \bar{S}} f(x)$, where $\bar{S}$ is a certain convex enlargement of S. For a general introduction and motivation of this technique, see for example [25, Chap. XII]; or also [19], [63, Chap. 6] for its application to combinatorial problems.

We will first describe in §1.2 what Lagrangian relaxation is. Then a list of possible applications (§2) will illustrate its versatility. Section 3 will be devoted to theory; in particular we will study the relations between the original problem and its relaxed version, i.e. between S and $\bar{S}$. The end of the paper will be devoted to numerical algorithms.

Our development may be deemed too technical at places. In particular, we will not necessarily assume S to be finite. As a result, we will have to care about continuity and boundedness, while combinatorics belong to the polyhedral world, with everything contained in the unit cube. Indeed, there are several reasons for our attitude:

(i) Perhaps paradoxically, it helps to think in abstract terms, forgetting as much as possible about linear algebra. The latter may make life easier, in particular skipping analysis and calculus; however this has its price: linear algebra is often "the tree hiding the forest", which prevents a sound understanding of what one is doing.

M. Jünger and D. Naddef (Eds.): Computat. Comb. Optimization, LNCS 2241, pp. 112–156, 2001.
© Springer-Verlag Berlin Heidelberg 2001

(ii) SDP optimization, which has entered recently the combinatorial world (see [1] for example), is a form of Lagrangian relaxation; and of course, SDP optimization is definitely beyond the polyhedral world.
(iii) Anyway, Lagrangian relaxation has a much larger field of applicability than combinatorial problems, and it does not harm to care about mathematical rigor.

1.2 The Basic Idea of Lagrangian Relaxation

The present section introduces the mechanism of Lagrangian relaxation as a *simple* yet *general* tool. Consider first an optimization problem put in the abstract form

$$\max f(x)\,, \quad x \in \mathcal{X}\,, \quad c_j(x) = 0,\ j = 1, \dots, m\,, \tag{1}$$

hereafter called the primal problem. We introduce the *Lagrangian*, a function of the primal variable x and of the *dual variable* $u \in \mathbb{R}^m$:

$$\mathcal{X} \times \mathbb{R}^m \ni (x, u) \mapsto L(x, u) := f(x) - \sum_{j=1}^{m} u_j c_j(x) = f(x) - u^\top c(x)\,, \tag{2}$$

where the last equality introduces the notation c for the m-vector of constraint-values. In plain words, the Lagrangian replaces each constraint by a linear "price" to be paid or received, according to the sign of u_j. Maximizing $L(\cdot, u)$ over the whole of $\mathcal{X}$ is therefore closely related to solving (1). In a sense, $\mathcal{X}$ can be considered as the "universe" in which x lives, while c represents the constraints that are "relaxed".

Definition 1 *The* dual function *associated with* (1), (2) *is the function of* u *defined by*

$$\mathbb{R}^m \ni u \mapsto \theta(u) := \max_{x \in \mathcal{X}} L(x, u)\,. \tag{3}$$

The dual problem *is then*[1]

$$\min \theta(u)\,, \quad u \in \mathbb{R}^m\,. \tag{4}$$

□

Lagrangian relaxation is a very versatile technique, which accepts essentially *any* data $\mathcal{X}, f, c$. The only really crucial assumption is the following, pragmatic but fundamental; it will be in force throughout:

[1] Notationally, we should write sup and inf throughout instead of min and max, since nothing guarantees that the various suprema are attained. Despite our care for mathematical rigor, we will neglect this subtlety and use min and max uniformly.

Note also that minimizing θ excludes the value $\theta(u) = +\infty$. As a result, the dual problem entails possible *dual constraints*, preventing the value $+\infty$ in (3).

Assumption 2 *Solving* (1) *is "difficult", while solving* (3) *is "easy". An* oracle *is available which solves* (3) *for given* $u \in \mathbb{R}^m$. □

Lagrangian relaxation then tries to take advantage of this, and aims at finding an appropriate u. To explain why finding an appropriate u amounts to solving the dual problem (4), observe the following triviality: by definition of θ,

$$\theta(u) \geqslant f(x) \quad \text{for all } x \text{ feasible in (1) and all } u \in \mathbb{R}^m \ ,$$

simply because $u^\top c(x) = 0$. This relation is known as *weak duality*, which gives two motivations for the dual problem:

– Because each $\theta(u)$ bounds from above the optimal cost in (1), minimizing θ amounts to finding the best possible such bound – a very relevant idea when bounding is at stake.
– The other motivation is less known, perhaps more important, and certainly more subtle. Suppose we do want to solve (1) with the help of (3). In other words, suppose we want a u such that (3) produces some argmax x_u of $L(\cdot, u)$ which also solves (1). In particular, this x_u must be feasible in (1) and therefore satisfy $\theta(u) = f(x_u)$. In view of weak duality, we see that the only chance for our u is to minimize θ.

Remark 3 *Suppose a problem with inequality constraints:*

$$\max f(x)\,, \quad x \in \mathcal{X}\,, \quad c_j(x) \leqslant 0,\ j = 1, \ldots, m\,.$$

To get the form (1)*, introduce slack variables* s *and use the flexibility of Lagrangian relaxation: set* $\mathcal{X}' := \mathcal{X} \times \mathbb{R}^m_+$*, so that we have to dualize*

$$\max f(x)\,, \quad x \in \mathcal{X}, s \geqslant 0 \in \mathbb{R}^m\,, \quad c(x) + s = 0 \in \mathbb{R}^m\,.$$

The Lagrangian is

$$L'(x, s, u) = f(x) - u^\top(c(x) + s) = L(x, u) - u^\top s$$

where $L(x, u)$ *is the "ordinary" Lagrangian (as though we had equalities). The resulting dual function* θ' *is clearly*

$$\theta'(u) = \begin{cases} +\infty & \text{if some } u_j < 0\,, \\ \theta(u) & \text{otherwise}\,, \end{cases}$$

where θ *is the "ordinary" dual function. In a word, the dual problem becomes* $\min_{u \geqslant 0} \theta(u)$*: the nonpositive part of the dual space is chopped off, we have an immediate illustration of note 1, p. 113.* □

To bypass the above reasoning, it suffices to memorize the following rule:

– consider a *maximization* problem
– and dualize a constraint c by *subtracting* the term $uc(x)$;
– if the constraint was of the *lower than* type,

– then the corresponding multiplier u must be *nonnegative*
– (one may prefer a minimization problem, with a + sign in the Lagrangian, and $\leqslant$-type constraints; the multipliers must again be nonnegative; memorizing the appropriate rule is a matter of taste).

Along the lines of Remark 3, suppose that one insists on having f linear in (1): adopt the formulation in the new universe $\mathcal{X} \times \mathbb{R}$

$$\max s_0\,, \quad s_0 \leqslant f(x)\,, \quad x \in \mathcal{X}\,, \quad c(x) = 0 \in \mathbb{R}^m\,.$$

A first possibility is to insert the constraint $s_0 \leqslant f(x)$ in the universe: we form the Lagrangian $s_0 - u^\top c(x)$, whose maximum over $x \in \mathcal{X}$ and $s_0 \leqslant f(x)$ is clearly $\theta(u)$; nothing is changed.

Alternatively, we can dualize that constraint, forming the Lagrangian

$$\mathcal{X} \times \mathbb{R} \times \mathbb{R}^{m+1} \ni (x, s_0, u, u_0) \mapsto L'(x, s_0, u, u_0) := s_0 - u_0[s_0 - f(x)] - u^\top c(x)\,.$$

Maximizing it with respect to s_0 (unconstrained!) gives $+\infty$ if $u_0 \neq 1$; and for $u_0 = 1$, its maximum with respect to $x \in \mathcal{X}$ gives $\theta(u)$; again nothing is changed.

Remark 4 *As a result, there is no loss of generality in assuming f linear in* (1)*. In case the given objective is really nonlinear, just use the formulation deemed the most convenient (with or without s_0); they all give the same dual anyway.* □

2 Examples

The examples of this section are intended to familiarize the reader with the duality mechanism, but also to show its versatility. Indeed, the simple principles of §1.2 above cover a wide range of techniques; see also [65, Chap. 4] for a number of applications issued from operations research.

2.1 Linear Programming

Take a linear program in standard form:

$$\max b^\top x\,, \quad x \geqslant 0 \in \mathbb{R}^n\,, \quad Ax = a \in \mathbb{R}^m\,.$$

Set first $\mathcal{X} := \mathbb{R}^n_+$, $c(x) := Ax - a$: with $u \in \mathbb{R}^m$, we obtain

$$L(x, u) = (b - A^\top u)^\top x + a^\top u\,, \quad \theta(u) = \begin{cases} a^\top u & \text{if } b - A^\top u \leqslant 0 \in \mathbb{R}^n, \\ +\infty & \text{otherwise}\,. \end{cases}$$

The dual problem is therefore to minimize $a^\top u$, subject to $A^\top u \geqslant b$.

Now set $\mathcal{X} := \mathbb{R}^n$ and dualize all constraints: $Ax - a = 0$ (dual variable $u \in \mathbb{R}^m$) and $-x \leqslant 0$ (dual variable $v \in \mathbb{R}^n$). We obtain the Lagrangian $L'(x, u, v) =$

$(b-A^\top u+v)^\top x+a^\top u$, to be minimized over x unconstrained. The dual function is therefore

$$\theta'(u,v)=\begin{cases} a^\top u & \text{if } b-A^\top u+v=0\,,\\ +\infty & \text{otherwise}\,. \end{cases}$$

Remembering that we also have $v\geqslant 0$, either from Remark 3 or from the rule following it, the dual problem is to minimize $a^\top u$ subject to $b-A^\top u=-v\leqslant 0$, which is exactly as before.

We leave it as an exercise to dualize in various ways various forms of LP, and to realize in each case that the selected duality scheme has little influence on the resulting dual problem. For example, any conceivable duality scheme for

$$\max b^\top x\,,\quad x\geqslant 0\,,\quad -\delta e\leqslant Ax-a\leqslant \delta e \tag{5}$$

($e\in\mathbb{R}^m$ is the vector of all ones, $\delta\geqslant 0$) eventually boils down to

$$\min a^\top u+\delta\sum_{j=1}^m |u_j|\,,\quad A^\top u\leqslant a\,;$$

the simplest scheme is to set $Ax-a=s$ and to dualize just that constraint.

2.2 Quadratic Programming

An example hardly more complicated is

$$\max b^\top x-\tfrac{1}{2}x^\top Qx\,,\quad Ax\leqslant a\,,$$

where Q is a symmetric matrix. Take $u\geqslant 0$ and form the Lagrangian

$$L(u,x)=(b-A^\top u)^\top x-\tfrac{1}{2}x^\top Qx+a^\top u\,.$$

To maximize it with respect to x unconstrained, one must consider three cases:

- If Q is positive definite, there is a unique maximum $x_u=Q^{-1}(b-A^\top u)$; it produces the quadratic dual function

$$\theta(u)=\tfrac{1}{2}u^\top AQ^{-1}A^\top u+(a-Ab)^\top u+\tfrac{1}{2}b^\top Q^{-1}b\,,$$

  which is nicely convex and must be minimized over $u\geqslant 0$.
- If Q is positive semidefinite, the Lagrangian has a finite maximum only if $b-A^\top u$ lies in the range of Q (so that the equation $Qx=b-A^\top u$ has a solution). This results in extra linear equality constraints in the dual problem.
- If Q is indefinite, the "maximum" is always at infinity: $\theta(u)=+\infty$ for all u; the dual problem produces no better upper bound than $+\infty$.

This example reveals a situation (the case of Q indefinite, or even $Q\prec 0$) where Lagrangian relaxation is of little help for finding upper bounds. Of course, the trouble here comes from a highly nonconvex primal problem; this will be seen in more detail in §3.

2.3 Max-cut, Max-stable, and Quadratic Constraints

Here we apply Lagrangian relaxation to two special combinatorial optimization problems. Both are particular instances of the general class (9) below, which also lends itself to the approach.

The max-cut problem can be written

$$\min \sum_{i,j=1}^{n} Q_{ij} x_i x_j = x^\top Q x\,, \quad x_i^2 = 1,\ i = 1, \ldots, n\,. \tag{6}$$

Needless to say, the constraints $x_i^2 = 1$ just express that $x_i = \pm 1$. The symmetric matrix Q has nonnegative coefficients, but this is irrelevant for our purpose.

The above problem has the form (1), with $\mathcal{X} := \mathbb{R}^n$ and $c_j(x) := x_j^2 - 1$. Form the Lagrangian

$$L(x,u) = x^\top Q x - \sum_{i=1}^{n} u_i (x_i^2 - 1) = x^\top (Q - D(u)) x + e^\top u\,; \tag{7}$$

here $D(u)$ denotes the diagonal matrix constructed from the vector u, e is the vector of all ones. Minimizing L with respect to x (on the whole of $\mathbb{R}^n$!) is a trivial operation: if $Q - D(u)$ is not positive semidefinite, we obtain $-\infty$; otherwise, the best to do is to take $x = 0$. Now (6) is a minimization problem, for which the dual function provides lower bounds; this dual function must be maximized. In a word, the dual problem associated with (6), (7) is

$$\max e^\top u\,, \quad \text{subject to } Q - D(u) \succcurlyeq 0\,. \tag{8}$$

An interesting point, to be seen in (14) below, is that the optimal value of this SDP (semi-definite programming) problem is just the SDP bound of [20]. This gives an incentive to dualize more general problems with quadratic constraints, such as

$$\begin{cases} \max x^\top Q_0 x + 2{b_0}^\top x + c_0\,, \\ x^\top Q_j x + 2{b_j}^\top x + c_j = 0\,,\ j = 1, \ldots, m\,. \end{cases} \tag{9}$$

The Lagrangian is the quadratic function $L(x,u) = x^\top Q(u) x + 2b(u)^\top x + c(u)$, where $Q(u) := Q_0 - \sum_{j=1}^{m} u_j Q_j$ and likewise for $b(u)$, $c(u)$. Maximizing $L(\cdot, u)$ is slightly more complicated than for maxcut, due to the linear term $2b(u)^\top x$; but the idea is just the same and gives rise again to SDP programming. In fact it can be shown that the dual of (9) is the problem with variables $(u, r) \in \mathbb{R}^m \times \mathbb{R}$

$$\min r\,, \quad \begin{pmatrix} c(u) - r & b(u)^\top \\ b(u) & Q(u) \end{pmatrix} \preccurlyeq 0\,. \tag{10}$$

Such quadratic duality goes far back, at least to [18], continuing with [66], and more recently [38].

This duality tool, available for general quadratically constrained problems, can be appplied to other combinatorial problems fitting in the same framework. One is the maximum stable set problem, which can be formulated as:

$$\max w^\top x\,, \quad x_i x_j = 0,\ (i,j) \in A\,, \quad x_i^2 = x_i,\ i = 1,\dots,n$$

($A \subset \{1,\dots,n\}^2$ being the set of arcs in a graph). Just as for maxcut, the bound obtained by dualizing the quadratic constraints is a known one: Lovasz' ϑ number [46]. For a systematic use of relaxation with quadratic constraints in combinatorial optimization, see [60,38].

2.4 Conic Duality

Remark 3 is just a particular case of the following variant of (1):

$$\max f(x)\,, \quad x \in \mathcal{X}\,, \quad c(x) \in K\,, \tag{11}$$

where K is a given set in $\mathbb{R}^m$. Here we assume that K is a closed convex cone (but see Remark 5 below). To introduce a dual problem, use again slack variables to put (11) in the form (1):

$$\max f(x)\,, \quad (x,s) \in \mathcal{X} \times K\,, \quad c(x) - s = 0 \in \mathbb{R}^m\,.$$

Again with $u \in \mathbb{R}^m$, we form the Lagrangian $L'(x,s,u) = L(x,u) + u^\top s$, where $L(x,u) = f(x) - u^\top c(x)$ as in (2). Then the dual function is

$$\theta'(u) = \max_{(x,s)\in\mathcal{X}\times K} L'(x,s,u) = \theta(u) + \max_{s\in K} u^\top s\,, \tag{12}$$

with θ of (3). The term $\max u^\top s$ is clearly 0 for u in the so-called *polar cone* of K:

$$K^\circ := \{u \in \mathbb{R}^m : u^\top s \leqslant 0 \text{ for all } s \in K\}\,, \tag{13}$$

and $+\infty$ elsewhere: our dual problem is to minimize $\theta(u)$ over $u \in K^\circ$. This is the whole business of conic duality, as developed for example in [47, Chap. 8] or [56, Chap. 6]; see also [25, §XII.5.3(a)]. It contains among other things SDP duality, as developed for example in [1,73].

Remark 5 *One may ask why K should be a closed convex cone. Actually,* (12) *gives the fully general dual function $\theta' = \theta + \sigma_K$, where $\sigma_K(u) := \sup_{s\in K} u^\top s$ is the so-called* support function *of K (a fundamental object of convex analysis). Arbitrary K's are therefore allowed. However, a support function does not distinguish between a set and its closed convex hull, a consequence of which is:*

- *there is no loss of generality in assuming K closed convex;*
- *said otherwise: the dual problem will not change if K is replaced by its closed convex hull.*

Needless to say, just set $K = \{0\}$ *in* (11) *to obtain* (1)*; the set* $\{0\}^\circ$ *of feasible* u*'s is then the whole space (alternatively, observe that* $\sigma_{\{0\}} \equiv 0$*). In the case of Remark 3, the polar of the nonnegative orthant is the nonpositive orthant. In fact, the support function of a cone* K *is easily computed: it is 0 on* K° *and* $+\infty$ *elsewhere – hence the dual constraint* $u \in K^\circ$ *in this case.*

As an illustration of this remark, we leave it as an exercise to dualize the following variant of (1) *– or of* (11) *with* $K = [-\delta, \delta]^m$*, see also* (5)*:*

$$\max f(x)\,, \quad x \in \mathcal{X}\,, \quad \|c(x)\|_\infty \leqslant \delta\,. \qquad \square$$

As a conclusion, we reproduce the rule following Remark 3:
– consider a *maximization* problem
– and dualize a constraint by *subtracting* the linear term $uc(x)$;
– if the constraint was $c(x) \in K$, a cone,
– then the corresponding multiplier u is constrained to vary in K° of (13)
– (or, for a general K, add σ_K to the dual function).

Take (8) as a particular case: the constraint-value $c'(u) := Q - D(u)$ lies in the space of symmetric matrices (we use primes to recall that (8) is already a dual problem; its dual will be a "bidual" of (6)). Symmetric matrices form a finite-dimensional vector space, which can be made Euclidean with the natural scalar product $\langle M, N\rangle := \sum_{i,j} M_{ij}N_{ij}$. Introducing a (bi)dual variable – a symmetric matrix, call it X – we form the (bi)Lagrangian

$$L'(u, X) = e^\top u - \langle X, Q - D(u)\rangle = e^\top u + \sum_{i=1}^n X_{ii}u_i - \langle Q, X\rangle\,,$$

to be maximized over $u \in \mathbb{R}^n$. This gives the (bi)dual function

$$\theta'(X) = \begin{cases} -\langle Q, X\rangle & \text{if } X_{ii} = 1 \text{ for } i = 1, \ldots, n\,, \\ +\infty & \text{otherwise}\,. \end{cases}$$

Now K is here the closed convex cone of positive semidefinite matrices. It can be shown ([27, Cor. 7.5.4] for example) that its polar is the (closed convex) cone of negative semidefinite matrices, which imposes the constaint $X \preccurlyeq 0$. Finally, change X to $-X$ to make the (bi)dual nicer: we obtain

$$\min \langle Q, X\rangle\,, \quad X \succcurlyeq 0\,, \quad X_{ii} = 1,\ i = 1, \ldots, n\,. \tag{14}$$

This is the SDP relaxation of [20]. Note that a positive semidefinite matrix can be put in the form $X = \sum_{k=1}^r x_k x_k^\top$, where r is the rank of X. Observing that $\langle Q, xx^\top\rangle = x^\top Q x$ and that $(xx^\top)_{ii} = x_i^2$, we see that (6) is just (14), but with X constrained to have the form $X = xx^\top$, i.e. to be of rank 1.

2.5 A Large-Scale Problem: Unit-Commitment

Perhaps the most obvious usage of Lagrangian relaxation is when (1) has many variables, which would become independent if the constraints c were not present.

For an illustration consider the so-called unit-commitment problem, which consists in optimizing the production planning of a set I of power plants, over some time horizon T. A possible formulation is as follows (see [42] and the references therein):

– the control variables are x_i^t for $i \in I$ and $t = 1, \dots, T$, they denote the production level of unit i at time t; x_i [resp. x^t] will stand for the vector $(x_i^t)_{t=1}^T$ [resp. $(x_i^t)_{i\in I}$];
– each unit i can operate within a certain feasible set $\mathcal{X}_i$, representing the operating dynamic constraints (for a nuclear plant, $\mathcal{X}_i$ is a finite set of possible plannings: x_i^t may take a few possible values, say $x_i^t \in \{500, 1000, 1500, 2000\}$ MW; when the production has reached a certain level, it must stay there for a certain number of time periods, etc.);
– at each time t, the demand must be satisfied; these are the so-called static constraints, denoted by $c^t(x^t) \leqslant 0$; usually, the functions c^t are additive, say $c^t(x^t) = \sum_{i\in I} c_i^t(x_i^t)$, or even affine;
– finally, the cost to produce x_i by unit i over the whole time horizon is $C_i(x_i)$.

Then the problem is

$$\begin{cases} \min \sum_{i\in I} C_i(x_i)\,, \\ x_i \in \mathcal{X}_i\,, & i \in I\,, \\ c^t(x^t) \leqslant 0\,, & t = 1, 2, \dots, T\,. \end{cases} \tag{15}$$

Insofar as the static constraints are additive, dualizing them produces an additive Lagrangian: with $u = (u^t)_{t=1}^T$,

$$L(x,u) = \sum_{i\in I} C_i(x_i) + \sum_{t=1}^T u^t c^t(x^t) = \sum_{i\in I} C_i(x_i) + \sum_{t=1}^T u^t \sum_{i\in I} c_i^t(x_i^t)\,.$$

After some reordering, we see that L is a sum of "local" Lagrangians, depending only on x_i: minimizing $L(\cdot, u)$ over $\mathcal{X} := \prod \mathcal{X}_i$ gives one problem per plant:

$$\min_{x_i\in\mathcal{X}_i} C_i(x_i) + \sum_{t=1}^T u^t c^t(x_i^t)\,,$$

which is a lot easier than (15) (to fix ideas, the French production set has some 150 plants working each day; optimizing each of them separately is a reasonable task, while optimizing them altogether is unconceivable).

Remark 6 *The above example illustrates perfectly the usual situation in Lagrangian relaxation: the only available information concerning* (1) *– here* (15) *– is the oracle which, given u, maximizes the Lagrangian $L(\cdot, u)$; and this oracle may be fairly complex. Here it is itself composed of some 150 "local oracles", all probably very different (nuclear, gas, hydraulic, ...) and probably very sophisticated, at least for laymen in electrical engineering.* □

2.6 Column Generation in Linear Programming

Consider the following type of problem. We have a very long list of points $\varkappa_1, \dots, \varkappa_\propto$ in $\mathbb{R}^n$, where $\propto$ is some enormous number, possibly $+\infty$. We want to find a $\varkappa_k$ maximizing a linear function, say $b^\top \varkappa_k$, and satisfying affine constraints, say $A\varkappa_k = a \in \mathbb{R}^m$: we write the problem as

$$\max \{b^\top \varkappa_k : A\varkappa_k = a, k = 1, \dots, \propto\} . \tag{16}$$

Assumption 7 *Column generation requires the following conditions:*

(i) The number m of constraints is reasonable (as opposed to the cardinality $\propto$ of the list).

(ii) Rather than solving the initial problem, one accepts to relax *it, replacing the set $\{\varkappa_1, \dots, \varkappa_\propto\}$ by its convex hull.*

(iii) It is "easy" to maximize linear functions over the $\varkappa_k$'s, i.e. to solve the unconstrained version of (16)*: $\max \{c^\top \varkappa_k : k = 1, \dots, \propto\}$, for given $c \in \mathbb{R}^n$ (possibly different from b).* □

In view of (ii), one is interested by *upper bounds*, as in Lagrangian relaxation. As for (iii), it says that one can compute the support function (see Remark 5) of the $\varkappa_k$'s – or of their convex hull, which is the same function. In other words, the constraints $A\varkappa_k = a$ are *complicating*. Clearly the situation resembles (1).

We refer to [75, Chap. 11] and the references therein for the many applications of column generation. Let us just mention: cutting stock, where each $\varkappa_k$ represents a cut pattern; network optimization, a path from a source to a destination; facility location, a set of clients assigned to a depot; etc.

Remark 8 *Actually, Assumption 7 is just what combinatorial optimization is all about, each $\varkappa_k$ representing one among many possible candidates to solving a certain problem. The most naive instance is the standard 0-1 linear programming problem*

$$\max b^\top x , \quad x \in \{0,1\}^n , \quad Ax = a . \tag{17}$$

After all, it can be formulated with notation of the present section: we want to find a 0-1 vector $\varkappa \in \mathbb{R}^n$ (there are $\propto = 2^n$ of them) maximizing a linear function and satisfying affine constraints. Barring these last constraints, the problem would become trivial.

Admittedly, column generation may not be deemed most appropriate for the present model, though. It is more constructive to distinguish two groups of constraints in (17)*: we write*

$$\max b^\top x , \quad x \in \{0,1\}^n , \; Dx = d , \quad Ax = a ,$$

the $\varkappa_k$'s being now the 0-1 points satisfying $Dx = d$. The approach is still feasible if, among other things, D is structured enough, so that it is easy to maximize a linear function $c^\top x$ on 0-1 vectors satisfying $Dx = d$. Dispatching a given set of constraints among the above A and D belongs to the art of 0-1 programming. □

Let us formulate (16) more compactly as

$$\max b^\top x\,, \quad x \in \mathcal{X}\,, \quad Ax = a\,. \tag{18}$$

This has the advantage of accepting $\propto = +\infty$; but more importantly, the connection with (1) becomes more blatant: the universe $\mathcal{X}$ is made of the $\varkappa_k$'s or, in view of Assumption 7(ii), of their convex hull. The objective is linear and the constraints dualized in (2) are affine. In view of Assumption 7(iii), the Lagrangian problem (3), which is here $\max_{x\in\mathcal{X}} b^\top x - u^\top(Ax-a)$, is "easy": Assumption 7(iii) is nothing other than Assumption 2.

Now column generation is a special technique to solve (18), in which $\mathcal{X}$ is iteratively replaced by smaller sets $\mathcal{X}_K \subset \mathcal{X}$: (18) is replaced by the *master program*

$$\max b^\top x\,, \quad x \in \mathcal{X}_K\,, \quad Ax = a\,. \tag{19}$$

A bounded polyhedron is taken for $\mathcal{X}_K$, so that (19) is an ordinary linear program. Also, the extreme points of $\mathcal{X}_K$ are extracted from the $\varkappa_k$'s in (16); but this is of no importance for the moment anyway: what is interesting is how $\mathcal{X}_K$ is updated for the next iteration.

To obtain $\mathcal{X}_{K+1}$ in column generation, one gets from the resolution of (19) an m-vector u_K (the multipliers associated with $Ax = a$), and one solves the *satellite program*: one finds the most positive among the numbers $(b - A^\top u_K)^\top \varkappa_k$, $k = 1, \ldots, \propto$, which, in view of Assumption 2(iii), is an "easy" problem. Using notation as in (19) and neglecting the constant term $a^\top u_K$, this problem amounts to solving

$$\max_{x\in\mathcal{X}} (b - A^\top u_K)^\top x + a^\top u_K = b^\top x - u_K^\top(Ax - a) = L(x, u_K)\,.$$

We see that the satellite does nothing other than computing the dual function $\theta(u_K)$ as in (3). Let us summarize these observations:

Fact 9 *Insofar as* (1) *has linear data ($\mathcal{X}$ polyhedral, f linear, g affine), column generation and Lagrangian relaxation do the same thing with the same tool, starting from opposite premises:*

- *In Lagrangian relaxation, the constraints $c(x) = 0$ are relaxed and the universe $\mathcal{X}$ is kept as "hard"; the resulting problem is solved in the oracle* (3).
- *Column generation works the other way round: the constraints $c(x) = 0$ are kept as "hard", while the universe is restricted to a smaller set; the resulting problem is solved in the master* (19). □

Thus, any formulation of a problem in terms of column generation can be done in terms of Lagrangian relaxation (and conversely). We believe that this is important because Lagrangian relaxation – duality theory – mostly calls for fairly simple concepts, to be seen in §3. By contrast, column generation has to call for the often fussy language of linear programming. In particular, it cannot be accounted for directly on the compact formulation (19). Remember our point (i) in §1.1.

Anyway Fact 9 has several interesting consequences:

– Column generation can be applied to more general problems than (16). In fact, the resulting methodology is often called *generalized linear programming*, as is lucidly explained in [49].
– Because column generation actually convexifies $\mathcal{X}$ in (18), Lagrangian relaxation should do the same. This will be confirmed in §3.2 below; see more precisely Theorem 17, Fact 18, Example 22.
– In Lagrangian relaxation, the dual function θ of (3) must be minimized. Now we have seen that, in the column generation language, $\theta(u_K) = s + a^\top u_K$, where s is the output from the satellite. The question at stake in column generation *is* therefore to find multipliers u_K such that $s + a^\top u_K$ – i.e. θ – is minimal.
– In other words, column generation *is by necessity* a minimization mechanism. The master (19) *must* provide a possible algorithm to minimize θ; this will be confirmed in §4.2 below.
– Alternatively, any algorithm to minimize θ *must* provide an alternative to the master (19) for column generation; this will be confirmed in §4.3 and 5.2 below; see more precisely Remark 33.

2.7 Entropy Maximization

Even though the following example is definitely out of the combinatorial world, we mention it to emphasize once more the versatility of Lagrangian relaxation. It is indeed an optimization technique that should always be kept in mind.

Entropy maximization appears all the time when an unknown function $x(t)$, $t \in [0,1]$ must be identified with the help of some measurements. Then one seeks the "most probable" $x(\cdot)$ compatible with these measurements. Here is a typical example, where the measurements are some Fourier coefficients of x:

$$\min \int_0^1 x(t) \log x(t) \mathrm{d}t \,, \quad \int_0^1 \cos(2j\pi t) x(t) \mathrm{d}t = z_j \,,\ j = 1, \dots, m \,.$$

This problem has infinitely many variables but finitely many constraints. Changing signs (we have a minimization problem) and forgetting the term $-\sum_j u_j z_j$ (constant in x), the Lagrangian $L(x,u)$ is the integral over $[0,1]$ of the function

$$\ell(x(t),u) := x(t)\log x(t) + \sum_{j=1}^m u_j \cos(2j\pi t) x(t) = x(t)\log x(t) + g_u(t)x(t) \,.$$

Skipping infinite-dimensional technicalities, we can admit the (true) fact that minimizing $L(\cdot,u)$ is achieved by minimizing $\ell(\cdot,u)$ for each t. This last problem is straightforward indeed: differentiate ℓ with respect to its first argument and solve for $x(t)$. We obtain $1 + \log x(t) + g_u(t) = 0$, which has the unique solution

$$e^{-1-g_u(t)} = \exp\bigl(-1 - \textstyle\sum_{j=1}^m u_j \cos(2j\pi t)\bigr) =: x_u(t) \,.$$

Plugging this value (always positive!) in the expression of ℓ gives an explicit formula for the dual function. See [9] for more details and references.

3 Minimal Amount of Convex Analysis

As seen in §1.2, Lagrangian relaxation just replaces a primal problem (1) by its dual (2)-(4). We will first give the main properties of the dual problem. Then we will consider questions posed in the primal when the dual problem is solved: how good is the upper bound? how about recovering a primal optimal solution? Some familiarity with elementary convex analysis is required here. For this, [26] may be useful.

3.1 Minimizing the Dual Function

In this section, we look at (1)-(4) with dual glasses, concentrating on the space $\mathbb{R}^m$ of dual variables, and somehow forgetting the primal space $\mathbb{R}^n$. Here is the fundamental result, whose proof is immediate:

Theorem 10 *Whatever the data $\mathcal{X}, f, c$ in (1) can be, the function θ is always convex and lower semicontinuous.*

Besides, if u is such that (3) has an optimal solution x_u (not necessarily unique), then $g_u := -c(x_u)$ is a subgradient of θ at u:

$$\theta(v) \geqslant \theta(u) + g_u^\top (v-u) \quad \text{for any } v \in \mathbb{R}^m \ , \tag{20}$$

which is written as $g_u \in \partial\theta(u)$. □

Some comments are useful for a good understanding of this result.

- An important object in convex analysis is the *epigraph* of a function θ, denoted by $\operatorname{epi}\theta$: this is the set of $(u,r) \in \mathbb{R}^m \times \mathbb{R}$ such that $\theta(u) \leqslant r$. It is not difficult to realize that convex functions are exactly those whose epigraph is a convex subset of $\mathbb{R}^m \times \mathbb{R}$.
- Lower semicontinuity means: for any $u \in \mathbb{R}^m$, the smallest cluster value of $\theta(v)$ when $v \to u$ is at least $\theta(u)$: $\liminf_{v\to u} \theta(v) \geqslant \theta(u)$. Alternatively, a function is lower semicontinuous when its epigraph is a closed subset of $\mathbb{R}^m \times \mathbb{R}$.
- Lower semi-continuity is an important property for minimization. It guarantees existence of a minimum point as soon as θ "increases at infinity". By contrast, a non-lower semicontinuous function would be for example

$$[0,+\infty[\ni u \mapsto \theta(u) := \begin{cases} u \text{ if } u > 0 \ , \\ 1 \text{ if } u = 0 \ . \end{cases}$$

  Minimizing such a function (over $u \geqslant 0$) is meaningless; said otherwise: this function has no minimum point.
- In our $L \to \theta$ context, it helps intuition to think of $\mathcal{X}$ as a sort of index-set: instead of $\theta(u) = \max_{x\in\mathcal{X}} f(x) - u^\top c(x)$, a writing more suggestive for our purpose would be: $\theta(u) = \max_{k\in\mathcal{K}} f_k - u^\top c_k$ (it goes with the column generation of §2.6, where $\mathcal{K} = \{1, 2, \ldots, \infty\}$). Thus the Lagrangian defines a family of functions $u \mapsto L$ which are *affine* in u.

– Now observe that taking the supremum of a family of functions corresponds to intersecting their epigraphs.
– Then convexity and lower semicontinuity of θ become easy to accept: in fact, the epigraph of each function $u \mapsto L$ is a closed convex set (a so-called half-space). Their intersection $\operatorname{epi}\theta$ is again closed and convex.
– As for the subgradient property (20), it is straightforward: by definition of θ and looking at (2), we have

$$\theta(v) \geqslant L(x,v) = L(x,u) - c(x)^\top (v-u)$$

for all $x \in \mathcal{X}$, including for $x = x_u$.
– Observe that $-c(u)$ is the vector of partial derivatives of L with respect to u. The subgradient relation gives a rigorous meaning to the following very informal statement: "the (total) derivatives of θ are the partial derivatives of L with respect to u".

Such a statement can be justified in very special situations: suppose that $\mathcal{X}$ is the space $\mathbb{R}^n$, that L is a smooth function of x and u, and that (3) has a unique solution x_u which varies smoothly with u. Then write

$$\frac{\mathrm{d}\theta}{\mathrm{d}u}(u) = \frac{\partial L}{\partial u}(x_u,u) + \frac{\partial L}{\partial x}(x_u,u)\frac{\partial x_u}{\partial u}\,;$$

but because x_u is a maximum point, the partials of L with respect to x vanish. This explains that, to differentiate θ, it suffices to differentiate L with respect to u.

An important consequence of Theorem 10 is that the dual problem is therefore always "easy", insofar as (3) is easy to solve. First, it can be qualified as well-posed, in the sense that minimizing a convex function is well-posed; and this independently of (1), be it NP-hard or anything else. Second, under the sole condition that we are able to maximize the Lagrangian, we obtain "for free" the value $\theta(u)$ of the function, and the value $-c(x_u)$ of a subgradient; both informations are important to solve the dual problem.

As a result, solving a dual problem is exactly equivalent to minimizing a convex function with the help of the oracle (3) providing function- and subgradient-values. Sections 4 and 5 will be devoted to this problem, from a numerical point of view.

3.2 Primal-Dual Relations

A natural question is now: the dual problem is easy, but what is it good for, in terms of the primal? We already know that it provides upper bounds but can we say more? and when can we obtain a primal optimal solution once the dual is solved?

Everett's Theorem. Let us start with a very elementary observation. Suppose we have maximized $L(\cdot,u)$ for a certain u, and thus obtained an optimal x_u,

together with its constraint-value $c(x_u) \in \mathbb{R}^m$; set $g_u := -c(x_u)$ and take an arbitrary $x \in \mathcal{X}$ such that $c(x) = -g_u$. By definition, $L(x,u) \leqslant L(x_u,u)$, which can be written

$$f(x) \leqslant f(x_u) - u^\top c(x_u) + u^\top c(x) = f(x_u)\,. \tag{21}$$

This is a useful result for approximate resolutions of (1):

Theorem 11 (Everett) *With the notation above, x_u solves the following perturbation of* (1)*:*

$$\max f(x)\,, \quad x \in \mathcal{X}\,, \quad c(x) = -g_u\,. \qquad \square$$

The above "a posteriori" perturbed problem consists in replacing the right-hand side of the constraints in (1) by the vector g_u. If this vector is small, x_u can be viewed as approximately optimal; if $g_u = 0$, we are done: x_u is optimal.

More can actually be said in Everett's Theorem.

- First, suppose x_u is only an ε-maximizer of the Lagrangian; in other words: $L(x_u,u) \geqslant L(x,u) - \varepsilon$ for all $x \in \mathcal{X}$. Then (21) becomes $f(x) \leqslant f(x_u) + \varepsilon$, and x_u becomes an ε-optimal solution of the perturbed problem.
- Second observe that, to obtain the desired property $f(x) \leqslant f(x_u)$ in (21), it suffices to have $u^\top(c(x) - c(x_u)) \leqslant 0$. As a result, x_u solves the problem

$$\max f(x)\,, \quad x \in \mathcal{X}\,, \quad u^\top c(x) \leqslant -u^\top g_u\,.$$

- Even more: x_u solves any other problem having more constraints than above, providing these constraints are satisfied by x_u. Thus, consider the particular case of Remark 3: it is clear that x_u solves the perturbed problem

$$\max f(x)\,, \quad x \in \mathcal{X}\,, \quad c_j(x) \leqslant \begin{cases} \max\{0, c_j(x_u)\} & \text{if } u_j = 0\,, \\ c_j(x_u) & \text{if } u_j > 0\,, \end{cases}$$

  in which optimality conditions appear explicitly: if x_u is such that $c_j(x_u) \leqslant 0$ (x_u feasible) with equality whenever $u_j > 0$ (complementarity slackness), then x_u solves (1).

Dual Subdifferentials. The above considerations go back to [12]. Here we focus our attention to a differential study of the dual function.

In view of Theorem 10, the partial gradient $\nabla_u L(x_u,u) = -c(x_u)$ of L with respect to u lies in $\partial\theta(u)$. A converse to this property turns out to be crucial for primal-dual relations. Introduce the notation

$$\begin{aligned} \mathcal{X}(u) &:= \{x \in \mathcal{X} : L(x,u) = \theta(u)\}\,, \\ G(u) &:= \{g = -c(x) : x \in \mathcal{X}(u)\} \end{aligned} \tag{22}$$

for the set of optimal solutions in (3) and its image through $\nabla_u L$. Because a subdifferential is always a closed convex set, we deduce that the closed convex hull of $G(u)$ is entirely contained in $\partial\theta(u)$. An important question is whether $\partial\theta(u)$ is thus entirely described; besides, getting rid of the (trouble-making) closure operation is desirable. This explains the following concept:

Definition 12 (Filling Property) *The filling property for* (1)-(3) *is said to hold at* $u \in \mathbb{R}^m$ *if* $\partial\theta(u)$ *is the convex hull of the set* $G(u)$ *defined in* (22). □

A first question is then: when does the filling property hold? This is actually fundamental in subdifferential calculus: when is it possible to characterize the subdifferential of a sup-function? The question is extremely technical and will be skipped in this paper, as it has little to do with our development. To get a positive answer requires additional assumptions, of a topological nature. Let us enumerate some favourable situations:

Theorem 13 *The filling property of Definition 12 holds at any* $u \in \mathbb{R}^m$
- *when* $\mathcal{X}$ *is a compact set on which* f *and each* c_j *are continuous;*
- *in particular, when* $\mathcal{X}$ *is a finite set (usual Lagrangian relaxation of combinatorial problems);*
- *in linear programming of §2.1 and in quadratic programming of §2.2;*
- *more generally, in problems where* f *and the* c_j *are quadratic (§2.3), or even* ℓ_p*-norms,* $1 \leqslant p \leqslant +\infty$*; see [71,37].* □

The interested reader may consult [68] for the most recent results along these lines. When the filling property holds at u, then any anti-subgradient of θ at u is a convex combination of constraint-values at sufficiently many x's in $\mathcal{X}(u)$; say

$$g \in \partial\theta(u) \quad \Longrightarrow \quad g = -\sum_k \alpha_k c(x_k)\,.$$

Now a u minimizing θ is characterized by the property $0 \in \partial\theta(u)$, which means: there exists sufficiently many x's in $\mathcal{X}(u)$ such that the $c(x)$'s have 0 in their convex hull:

Proposition 14 *Let* u^* *solve the dual problem* (4) *and suppose the filling property holds at* u^**. Then there are (at most* $m+1$*) points* x_k *and convex multipliers* α_k *such that*

$$L(x_k, u^*) = \theta(u^*) \text{ for each } k, \text{ and } \sum_k \alpha_k c(x_k) = 0\,. \qquad \square$$

With these x_k's and α_k's, it is tempting to make up the primal point

$$x^* := \sum_k \alpha_k x_k\,, \tag{23}$$

which definitely deserves attention[2]. In fact, the following reasoning reveals conditions under which this x^* does solve (1):

[2] Of course, the writing (23) implies that $\mathcal{X}$ enjoys some structure, namely that its elements can be *averaged*. Besides, observe the connotation of column generation: as a convex combination of primal points computed by the satellite, x^* could very well solve (19) with an appropriate $\mathcal{X}_K$.

– If $\mathcal{X}$ is a convex set (say in $\mathbb{R}^n$) then $x^* \in \mathcal{X}$. Appropriate use of Everett's Theorem 11 tells us to what extent x^* is approximately optimal in (1).
– If, in addition, c is an affine mapping (from $\mathbb{R}^n$ to $\mathbb{R}^m$) then

$$c(x^*) = \sum_k \alpha_k c(x_k) = 0\,.$$

Thus x^* is feasible in (1). Besides $L(x^*, u^*) = f(x^*) - (u^*)^\top c(x^*) = f(x^*)$, so the number $\theta(u^*) - f(x^*)$ (nonnegative because of weak duality) again tells us to what extent x^* is aproximately optimal. This number, the so-called *duality gap*, will be studied in more detail below.
– If, in addition, $L(\cdot, u^*)$ is concave (as a function defined on $\mathbb{R}^n$) then

$$f(x^*) = L(x^*, u^*) \geqslant \sum_k \alpha_k L(x_k, u^*) = \theta(u^*)\,.$$

In view of weak duality, x^* is optimal in (1).

Remark 15 *The above reasoning can be reproduced in the case of conic duality of §2.4, knowing that the dual optimality condition is no longer* $0 \in \partial\theta(u^*)$ *but rather: there is* $g \in \partial\theta(u^*)$ *lying in the normal cone*[3] *to* K° *at* u^**. Alternatively, one can use the "slackened Lagrangian"* $L'(x, y, u) = L(x, u) + u^\top y$*, to be maximized over* $\mathcal{X} \times K$ *and just apply the same reasoning. We leave it as an exercise to do the calculations in the inequality-constrained case of Remark 3.* □

It is important to realize that we have here a *constructive process* to make primal optimal solutions:

Fact 16 *Let a dual algorithm produce* u^* *solving the dual problem. Suppose that*
– *the filling property of Definition 12 holds at* u^**,*
– *appropriate convexity properties hold for* $\mathcal{X}$*,* c *and* $L(\cdot, u^*)$ *so that the above reasoning is valid.*

Under the mere condition that this algorithm is able to declare optimality of u^**, it provides through* (23) *a point* x^* *which solves the primal problem* (1). □

Let us emphasize this fact: unless the dual algorithm minimizing θ is a naive heuristic, without even a proof of convergence, the information necessary to compute x^* – i.e. the x_k's and α_k's in (23) – is certainly available somewhere in this algorithm, namely in its stopping criterion: to detect (approximate) optimality of u^*, the algorithm must know (approximate) maximizers x_k of $L(\cdot, u^*)$ and convex multipliers α_k such that $g_{u^*} := -\sum_k \alpha_k c(x_k)$ is (approximately) zero. More will be seen on this question in Section 5.2.

[3] This is the set of g such that $g^\top(v - u^*) \leqslant 0$ for all $v \in K^\circ$; or equivalently, the set of $g \in K$ such that $g^\top u = 0$; see [25, Ex. III.5.2.6(a)] for example.

Duality Gaps. We have already encountered the *duality gap*: this is the difference of the optimal values in (4) and in (1). It is always a nonnegative number because of weak duality. In view of Proposition 14 and the reasoning following it (and remembering Fact 9), we guess that the dual problem amounts to solving a certain *convex relaxation* of (1). The duality gap would then be the corresponding increase of the optimal value, and this is the object of the present section; see also [10, §16], [14,42].

The most general and best known result is as follows. Introduce the *perturbation function* associated with (1), which is the optimal value as a function of the righthand side of the constraints:

$$\mathbb{R}^m \ni g \mapsto v(g) := \max\{f(x) : x \in \mathcal{X},\, c(x) = g\}\,. \tag{24}$$

Lagrangian relaxation amounts to replacing v by its *concave upper semicontinuous hull*: this is the smallest function v^{**} which is concave, upper semicontinuous (u.s.c.) and larger than v: $v^{**}(g) \geqslant v(g)$ for all $g \in \mathbb{R}^m$. Alternatively, v^{**} is the function whose hypograph (everything that is under the graph) is the closed convex hull of the hypograph of v, as a subset of $\mathbb{R}^m \times \mathbb{R}$. The role of v and v^{**} comes from the following result:

Theorem 17 *The dual optimal value is the value at 0 of the concave u.s.c. hull of the perturbation function:* $\min\theta = v^{**}(0)$. □

Similarly to Proposition 14, this result is completely general, as it assumes no structure whatsoever on $\mathcal{X}$ (while Fact 16, for example, implies that the word "convex combination" has a meaning in $\mathcal{X}$, remember note 2, p.127). In a way, it is minimal: it only requires the image-space $\mathbb{R}^m$ of the constraints to be a Euclidean spacee and neglects completely the. On the other hand, it is rather abstract: visualizing the behaviour of the perturbation function is not easy, in terms of the data $\mathcal{X}, f, c$. This is why some more specific properties are worth investigating when some structure is present in the data.

When $\mathcal{X} \subset \mathbb{R}^n$ (so that taking convex combinations in $\mathcal{X}$ makes sense) it becomes possible to analyze the duality gap in a more primal language.

Assume first that the Lagrangian is affine in x. Then its maximum values are not changed if $\mathcal{X}$ is replaced by its closed convex hull[4]. In view of weak duality, this implies the following:

Fact 18 *For an instance of* (1) *with linear data:*

$$\max b^\top x\,, \quad x \in \mathcal{X} \subset \mathbb{R}^n\,, \quad Ax = a\,, \tag{25}$$

denote by $\bar{\mathcal{X}}$ *the closed convex hull of* $\mathcal{X}$. *The dual minimal value* $\min\theta$ *is not smaller than the maximal value in* (25) *with* $\mathcal{X}$ *replaced by* $\bar{\mathcal{X}}$. □

[4] The dual function is then essentially the support function of $\mathcal{X}$, already seen in Remark 5; more precisely, in the notation of (25), $\theta(u) = a^\top u + \sigma_{\mathcal{X}}(b - A^\top u)$.

Beware that "not smaller than" does not mean "equal to": this result does *not* mean that the duality gap amounts to replacing $\mathcal{X}$ by $\bar{\mathcal{X}}$. In terms of Theorem 17, convexifying $\mathcal{X}$ amounts to taking the concave hull of the perturbation function v; but the resulting function need not be upper semicontinuous at 0. Here lies a technical difficulty, which will be illustrated by the counter-example 22 below. Nevertheless, cases where this difficulty does occur can be considered as pathological. The duality gap exactly makes this convexification in most cases, for example those described in the following result:

Theorem 19 *Equality holds in Fact 18 in any of the following cases:*

(i) $\bar{\mathcal{X}}$ *is a bounded set in* $\mathbb{R}^n$*;*

(ii) for any $g \in \mathbb{R}^m$ *close enough to 0, there is* $x \in \bar{\mathcal{X}}$ *such that* $Ax = a + g$*;*

(iii) there exists u^* *minimizing the dual function, and the filling property 12 holds at* u^**.* □

– Property (i) above is of course that of "ordinary" Lagrangian relaxation in combinatorial optimization; the corresponding result goes back to [13], rediscovered on various occasions, see for example [19,49]. At this point, it is worth mentioning that the closed convex hull of a bounded set is bounded ([25, Thm. IV.1.4.3] for example): when $\mathcal{X}$ is a finite set, $\bar{\mathcal{X}}$ is just the convex hull of $\mathcal{X}$.

– Property (ii) is universally invoked, under the name of *Slater condition*. Less known is the fact that it is necessary and sufficient for the dual function to have a nonempty compact set of minimum points.

– As for Property (iii), the result is easy to establish from Proposition 14 and the discussion following it.

Remark 20 *Remember column generation of §2.6: it also replaces a set* $\mathcal{X}$ *by its closed convex hull. When* $\propto < +\infty$*, Theorem 19 applies: both column generation and Lagrangian relaxation find the common optimal value of* (4) *or* (18)*.*

When $\propto = +\infty$ *– or more generally if* $\mathcal{X}$ *in* (18) *is an arbitrary (unbounded) closed convex set – there may be a pathological duality gap. In that case, Lagrangian relaxation does not solve* (18)*, but rather its dual* (4)*. So does column generation as well, this will result from §4.2 below.* □

The previous development applies when the Lagrangian $L(\cdot, u)$ is affine. Such is no longer the case when (1) has non-affine data f and/or c; the situation is then substantially more complicated, we outline the kind of results that can be obtained. Similarly to §2.4 and Remark 4, use a slack-technique and introduce the set

$$\mathcal{G} := \{(x, s_0, s) : x \in \mathcal{X},\ s_0 \leqslant f(x),\ s = c(x)\} \subset \mathbb{R}^n \times \mathbb{R} \times \mathbb{R}^m\,, \qquad (26)$$

which combines the graph of c and the epigraph of f. Put the s_0-axis as vertical. Solving (1) amounts to intersecting $\mathcal{G}$ with the subspace $s = 0$, and then finding the highest point in the resulting set. Now write the dual function as

$$\theta(u) = \max_{(x,s_0,s)\in\mathcal{G}} s_0 - u^\top s$$

to see that it is not changed if $\mathcal{G}$ is replaced by its closed convex hull[5]. This shows how Fact 18 and Theorem 19 can be generalized to nonlinear data: Lagrangian relaxation amounts to close-convexifying the above $\mathcal{G}$. However, the closed convex hull of $\mathcal{G}$ – and its intersection with $s = 0$ – is a rather complicated object; so we stop our development at this point and refer the interested reader to [42].

Exact Relaxation. Finally, a relevant question is when the duality gap is 0. According to Theorem 17, this occurs exactly when the perturbation function v coincides with its concave u.s.c. hull at 0. Such is the case when v itself is concave and upper semicontinuous at 0.

- Concavity holds whenever "everything is convex" in (1) (cf. the discussion following Theorem 14): $\mathcal{X}$ is a convex set (say in $\mathbb{R}^n$), f is a concave function on $\mathcal{X}$, and c is affine: $c(x) = Ax - a$. As already observed in Remark 20, there is little hope to have a 0 duality gap if $\mathcal{X}$, or more generally $\mathcal{G}$ of (26), is not convex.
- Upper semicontinuity is a "normal" property, in the sense that convex problems with a duality gap can be considered as pathological. Two sorts of situations are known, where such a pathology cannot occur:
  - (i) One is similar to the situation where the filling property 12 holds, namely when the primal problem enjoys some compactness – see Theorem 13.
  - (ii) The other is based on the following important result: a convex function is continuous at a point whenever it is defined ($< +\infty$) in a neighborhood of this point. Particularizing this result to our context, assume that the perturbation function v of (24) is concave. It is continuous (hence u.s.c.) at 0
    - if it is finite ($> -\infty$) in a neighborhood of 0, or equivalently,
    - if the feasible domain remains feasible despite small perturbations of the righthand sides.

    This explains the Slater condition popping up in Theorem 19. Interestingly enough, this situation is somehow dual to (i), in that it corresponds to compactness in the dual space.

We collect these situations in one exactness result:

Theorem 21 *Let* (1) *be a convex optimization problem: $\mathcal{X}$ is a closed convex set in $\mathbb{R}^n$, $f : \mathcal{X} \to \mathbb{R}$ is a concave function, the constraint-functions c_j are affine. Assume that the dual function* (3) *is not identically $+\infty$. Then there is no duality gap* – (1) *and* (4) *have the same optimal value* – *if one at least of the following properties holds:*

(i) $\mathcal{X}$ is a bounded set in $\mathbb{R}^n$, f [resp. each inequality constraint] is upper [resp. lower] semicontinuous on $\mathcal{X}$;

(ii) for any $g \in \mathbb{R}^m$ close enough to 0, there is $x \in \mathcal{X}$ such that $c(x) = g$;

(iii) there exists u^* minimizing the dual function, and the filling property 12 holds at u^*;

[5] Comparing with note 4, p. 129, we have $\theta(u) = \sigma_{\mathcal{G}}(0, 1, -u)$.

(iv) $\mathcal{X}$ *is a polyhedral set,* f *and all constraints are affine functions (linear programming).* □

Adapting this result to (non-affine) inequality contraints (Remark 3) and conic programming (§2.4) is an interesting exercise. We conclude this section with a counter-example, essentially due to R.T. Rockafellar, to show that pathological duality gaps do exist.

Example 22 *Consider the following instance of* (1)*, set in terms of* $(x,y,z) \in \mathbb{R}^3$*;*

$$\begin{cases} \max x\,, \quad \mathcal{X} := \left\{(x,y,z) : \sqrt{x^2+y^2} \leqslant z\right\}, \\ c_1(x,y,z) := x - 1 \leqslant 0\,, \\ c_2(x,y,z) := z - y \leqslant 0\,, \\ c_3(x,y,z) := 1 - z \leqslant 0\,. \end{cases}$$

It is easy to see that the constraints imply $z = y$ *and* $x = 0$*. The optimal value is 0, attained on all of the feasible set, namely* $\{0\}\times\,]-\infty,1] \times [1,+\infty[$*.*

Now keep $\mathcal{X}$ *as the universe, introduce three (nonnegative) dual variables* u,v,w *and dualize* c_1*,* c_2*,* c_3*, forming the Lagrangian*

$$\begin{aligned} L &= x - u(x-1) - v(z-y) - w(1-z) \\ &= (1-u)x + vy + (w-v)z + u - w\,. \end{aligned}$$

Maximizing it with respect to z *implies* $w - v \leqslant 0$ *– otherwise we get* $+\infty$ *–, in which case the optimal* z *is* $\sqrt{x^2+y^2}$*. The dual function is therefore obtained by maximizing*

$$L' = (1-u)x + vy - (v-w)\sqrt{x^2+y^2}$$

with respect to x,y*. A key is to realize that the maximum of* L' *is either 0 or* $+\infty$*, depending whether* $\sqrt{(1-u)^2+v^2} \leqslant v - w$*; in fact the maximal value of* L' *and its argmax are given as follows:*

$\sqrt{(1-u)^2+v^2}$	$\max_{x,y} L'$	*maximal* (x,y)
$< v - w$	0	$(0,0)$
$= v - w$	0	$\mathbb{R}_+(1-u,v)$
$> v - w$	$+\infty$	$-$

To see it[6]*, fix first the norm of* $(x,y) \in \mathbb{R}^2$*; the maximum is attained for* (x,y) *collinear to* $(1-u,v)$*. Then adjust suitably the norm of* (x,y)*.*

Because $w \geqslant 0$*, the condition* $\sqrt{(1-u)^2+v^2} \leqslant v - w$ *implies* $u - 1$ *and* $w = 0$*; then this condition is satisfied for arbitrary* $v \geqslant 0$*. The corresponding value of* L *(the dual function) is* $u - w = 1$*. In summary, the dual optimal value is 1 (*> 0*), attained on all of the dual feasible set, namely* $\{1\} \times \mathbb{R} \times \{0\}$*.* □

[6] An expert in convex analysis will recognize in $\max L'$ the indicator function of the ball of radius $(v-w)$ at the point $(1-u,v) \in \mathbb{R}^2$ (up to the multiplicative term $v-w$).

Among other things, this example illustrates the distinction between Fact 18 and Theorem 19: the objective is linear, the constraints are affine, both primal and dual problems have optimal solutions. Nevertheless there is a duality gap; what is missing is the filling property 12. We mention that the counter-example relies on a crucial property: its primal and dual optimal sets are unbounded, as well as the argmax of the Lagrangian at a dual optimal solution.

4 Dual Algorithms I: Subgradient, Cutting Plane, ACCPM

We now come back to the dual space and study numerical methods solving the dual problem. Mainly for simplicity, we assume that there are no dual constraints, a solution exists in (3) for all $u \in \mathbb{R}^m$. Even though it is not essential for dual algorithms, this assumption eases substantially the exposition.

Thus we are in the following situation:

Assumption 23 *We are given a convex function θ, to be minimized.*
- *It is defined on the whole of $\mathbb{R}^m$.*
- *The only information available on θ is an oracle, which returns $\theta(u)$ and one subgradient $g(u)$, for any given $u \in \mathbb{R}^m$.*
- *No control is possible on $g(u)$ when $\partial\theta(u)$ is not the singleton $\nabla\theta(u)$.* □

The third assumption above just means that it is impossible to select a particular maximizer of the Lagrangian, in case there are several. As far as dual algorithms are concerned, the fact that $g(u)$ has the form $-c(x_u)$ is irrelevant: x_u is an unknown object to the dual algorithm, which deals only with vectors in $\mathbb{R}^m$. The present situation is rather classical in the world of nonlinear progamming, where one has to minimize some objective function, given function- and gradient-values only, say $\theta(u)$ and $g(u)$ respectively. The difference here is that $g(u)$ does not vary continuously with u: we are dealing with *nonsmooth optimization*.

Remark 24 *In some situations, a much richer oracle is available; most notably in the relaxation of quadratically constrained problems of §2.3. The corresponding dual problems are traditionally solved via interior-point methods, coming from* SDP *optimization ([1,73]). However, problems such as* (8)*, or more generally* (10)*, can be fruitfully considered as instances of nonsmooth convex problems, for which a rather complete subdifferential analysis can be performed. For such problems, algorithms of the type below can be used, as well as more specialized ones, taking advantage of the richer information available. This gives birth to useful alternatives to interior-point methods, particularly well-suited for large* SDP *problems; see [24,39,59].* □

There are essentially two types of nonsmooth optimization methods: subgradient (§4.1) and cutting-plane[7] (§4.2), somehow lying at the opposite sides of a continuum of methods; in between, are found analytic center and bundle (§5).

[7] This terminology is unfortunate in a combinatorial community: here the planes in question cut a portion of the space $\mathbb{R}^{m+1}$ where epi θ is lying.

In what follows, we will call g_k the $g(u_k)$ computed by the oracle of Assumption 23, when called at some iterate u_k.

4.1 Subgradients and Ellipsoid Methods

Let u_K be the current iterate. Consider the subgradient relation (20) for u strictly better than the current iterate u_K: we have

$$\theta(u_K) > \theta(u) \geqslant \theta(u_K) + g_K^\top (u - u_K)\,,$$

which shows that $g_K^\top(u - u_K) > 0$ for any such u, in particular for $u = u^*$, a point minimizing θ. Then it is easy to see that, for $t > 0$ small enough, the point $u(t) := u_K - tg_K$ is strictly closer than u_K to u, hence to any optimal u^*.

This is the starting idea for the subgradient method, going back to [72], formalized in [11,61], and often called "Lagrangian relaxation" in the combinatorial community[8]. It works as follows:

$$\text{At iteration } K\text{, select } t_K > 0 \text{ and set } u_{K+1} = u_K - t_K g_K\ .$$

It is hard to imagine anything simpler. In fact, the only responsibility of the method is to choose the stepsize t_K – since no control on g_K can be exerted from the dual space. Besides, the choice of t_K is also the simplest thing in the world, at least in theory:

Theorem 25 *Let the subgradient method be applied with the stepsizes given as follows:*

$$t_K = \frac{\lambda_K}{\|g_K\|}\ , \quad \textit{with}\ \ \lambda_K \downarrow 0\ \textit{and}\ \sum_{K=1}^{\infty} \lambda_K = +\infty\,.$$

Then the sequence of best function-values $\theta_K^* := \min\{\theta(u_k) : k = 1, \ldots, K\}$ *tends to the minimum value of* θ *over* $\mathbb{R}^m$. □

However, this extreme simplicity has its price in terms of speed of convergence: the method can only give a coarse approximation of the optimal value. Anyway observe that the above result is of little practical value: who can wait long enough to check that λ_K satisfies the required properties? This is why a big issue when implementing this method is a correct adjustment of t_K; a hard job because the real trouble lies in g_K, a bad direction[9].

Considering that better convergence implies better directions, an idea is to adapt the metric. Make a change of variable in $\mathbb{R}^m$, say $u = Bv$ (B being an invertible matrix). Then minimizing $\theta'(v) := \theta(Bv)$ is obviously equivalent to

[8] This terminology is unfortunate in *any* community: it mixes the general *methodology* of Lagrangian relaxation and the particular *technique* of subgradient optimization: the latter method is by no means the only possibility to implement the mechanism (3), (4) – or more generally, to minimize a convex function.

[9] In nonlinear programming, $-g_K$ corresponds to the gradient method, or steepest descent, only used by amateurs.

minimizing θ. However, suppose that θ is differentiable; then the gradient of θ' is $\nabla\theta'(v) = B^\top \nabla\theta(Bv)$. It turns out that the same formula holds for subgradients. As a result, the subgradient iteration to minimize θ' is $v_{K+1} = v_K - t_K B^\top g_K$. Multiplying by B to return in the u-space, we obtain $u_{K+1} = u_K - t_K BB^\top g_K$; the direction has changed, as requested.

N.Z. Shor thus devised a class of *variable metric* subgradient methods, with a metric $B = B_K$ varying at each iteration. To construct B_K, he used the elementary operator *dilating* the space along a given vector $\xi = \xi_K$, suitably chosen. He gave two choices for ξ:

The Ellipsoid Algorithm. His first idea was motivated by the following observation ([65, §3.1]): assume for simplicity that θ has a unique minimum point u^*; the (sub)gradient direction is so bad that it may be considered as orthogonal to the direction pointing to u^*. It is therefore advisable to favour the subspace orthogonal to $B_K g_K$. This led him to take $\xi_K = B_K g_K$; the resulting algorithm was given in [64]. Later on, A.S. Nemirovski defined specific values for the coefficient of dilatation and for the stepsize t_K [54], and this was the celebrated ellipsoid algorithm.

The *r* Algorithm. Shor was never too convinced by the behaviour of the above algorithm, and he defined another choice for ξ. Consider a point where $g(\cdot)$ is discontinuous: for two values of u rather close together, say u_K and u_{K-1}, the oracle returns two very different subgradient-values g_K and g_{K-1}. Then there are good reasons to think that the desired direction (pointing toward u^*) is orthogonal to $g_K - g_{K-1}$ (draw a picture in two dimensions, with g_K and g_{K-1} quasi-opposite). It is therefore advisable to dilate the space along the bad direction $\xi_K := g_K - g_{K-1}$. This gave birth to the r-algorithm [67].

Remark 26 *Another derivation of the subgradient method is worth mentioning. Take a nonsingular $m \times m$ matrix M and form the quadratic function $q(h) := \theta(u_K) + g^\top h + \frac{1}{2} h^\top M h$. Straightforward calculations show that it has a minimum point at $h = -M^{-1}g$. Thus, a subgradient algorithm can be derived as follows: consider $q(h)$ with $M = B^{-\top} B^{-1}$ as a "model" approximating $\theta(u_K + h)$; then minimize q; then move along the direction thus obtained with a stepsize t_K. This idea of an approximating model such as q is important in optimization; it will be most instrumental in the methods to come.* □

We conclude this section with a comment about primal recovery. Indeed the subgradient algorithm does provide the primal point x^* of Fact 16. This little-known property belonged to folklore ([65, p. 116], [2]), until it was formally given in [35]. In fact, suppose each primal point x_k, computed by the oracle at u_k, are stored along the iterations. Then compute

$$\hat{x}_K := \frac{\sum_{k=1}^K t_k x_k}{\sum_{k=1}^K t_k} .$$

Under reasonable assumptions, this averaged point converges to x^* of (23).

4.2 The Cutting-Plane Method, or of Kelley, Cheney-Goldstein

The previous methods were simplest, here is now a fairly sophisticated one, going back to [7,28]. Every call to the oracle, input with u_k say, defines an affine function approximating θ from below: $\theta(u) \geqslant \theta(u_k) + g_k^\top (u - u_k)$ for all $u \in \mathbb{R}^m$ (with equality for $u = u_k$). After K such calls, we have on hand a piecewise affine function $\hat{\theta}$ which underestimates θ: for all $u \in \mathbb{R}^m$,

$$\theta(u) \geqslant \hat{\theta}(u) := \max \{\theta(u_k) + g_k^\top (u - u_k) : k = 1, \dots, K\}, \tag{27}$$

which we will call the *cutting-plane model* of θ. It is entirely characterized by the K-uple of elements $(\theta(u_k), g_k) \in \mathbb{R}^{m+1}$; this is what we will call the *bundle* of information.

Admitting that the model approximates correctly the true θ, minimizing it makes sense; and this is a linear programming problem. Thus, the method commonly called "cutting planes" in nonlinear programming, or also Kelley-Cheney-Goldstein, consists in computing an optimal solution (u_{K+1}, r_{K+1}) of

$$\begin{cases} \min r\,, \quad (u, r) \in \mathbb{R}^{m+1}\,, \\ r \geqslant \theta(u_k) + g_k^\top (u - u_k) \quad \text{for } k = 1, \dots, K\ . \end{cases} \tag{28}$$

The next iterate is u_{K+1}, with its model-value $\hat{\theta}(u_{K+1}) = r_{K+1}$. A call to the oracle input with u_{K+1} gives a new piece $(\theta(u_{K+1}), g_{K+1})$ enriching the bundle (raising the cutting-plane model $\hat{\theta}$) and the process is repeated.

In view of (27), we have by construction $r_{K+1} = \hat{\theta}(u_{K+1}) \leqslant \hat{\theta}(u) \leqslant \theta(u)$ for all $u \in \mathbb{R}^m$. Then the number

$$\delta := \theta(u_K) - r_{K+1} = \hat{\theta}(u_K) - \hat{\theta}(u_{K+1}) \geqslant 0 \tag{29}$$

is of particular interest.

- It represents a *nominal decrease*, which would be dropped from the true θ if the model were accurate enough[10].
- It serves to bracket the dual optimal value: because $r_{K+1} = \theta(u_K) - \delta \leqslant \theta(u)$ for all u, we have

$$\theta(u_K) \geqslant \min \theta \geqslant \theta(u_K) - \delta = r_{K+1}\,.$$

- Hence it can serve as stopping test: the algorithm can be stopped as soon as δ is small[11], and this will eventually occur if (and only if) 0 is a cluster point of the sequence of δ-values.
- In view of the bracket, this last event exactly means

$$\liminf \theta(u_K) = \limsup r_{K+1} = \min \theta\,.$$

[10] It is even a maximal decrease, and even an upper bound on the total possible decrease $\theta(u_K) - \min \theta$.

[11] A tradition is rather to stop when $\theta(u_{K+1}) - r_{K+1}$ is small. We introduce δ because it will be a central object in bundle methods, §5 below. Anyway the most reasonable test is based on the best function-value θ_K^* of Theorem 25 and should be preferred.

– The above sequence $(\theta(u_K))$ can be – and usually is – cahotic. By contrast, the sequence (r_K) is obviously nondecreasing (since the function $\hat{\theta}$ increases at each iteration) and bounded from above (by any $\theta(u)$); it therefore has a limit.

Thus the key issue for convergence is whether $r_K \uparrow \min \theta$. Such is the case indeed, but with a technical restriction which turns out to be important: a mechanism is needed to guarantee *boundedness* of the sequence (u_K).

The Case of Column Generation. To close this section, we proceed to show that the present cutting-plane algorithm is *nothing other* than column generation of §2.6. In fact form the dual of (28): with (bi)dual variables $\alpha_k \geqslant 0$, the (bi)Lagrangian is

$$L'(u, r, \alpha) = r - r \sum_{k=1}^{K} \alpha_k + \sum_{k=1}^{K} \alpha_k \big(\theta(u_k) - g_k^\top u_k\big) + u^\top \sum_{k=1}^{K} \alpha_k g_k \,.$$

Minimizing $L'(u, \cdot, \alpha)$ forces $\sum_k \alpha_k = 1$: the α_k's form a set of convex multipliers. Minimizing $L'(\cdot, r, \alpha)$ forces $\sum_k \alpha_k g_k = 0$. The (bi)dual problem is therefore

$$\max \sum_{k=1}^{K} \alpha_k \big(\theta(u_k) - g_k^\top u_k\big) \,, \quad \alpha_k \geqslant 0 \,, \ \sum_{k=1}^{K} \alpha_k = 1 \,, \ \sum_{k=1}^{K} \alpha_k g_k = 0 \,.$$

Now remember that we are actually solving a dual problem: returning to the primal space of (1), we can write $\theta(u_k) = L(x_k, u_k)$ and $g_k = -c(x_k)$, so that[12] $\theta(u_k) - u_k^\top g_k = f(x_k)$. Thus, the above (bi)dual is

$$\max_{\alpha \in \Delta_K} \sum_{k=1}^{K} \alpha_k f(x_k) \,, \quad \sum_{k=1}^{K} \alpha_k c(x_k) = 0 \,, \tag{30}$$

where Δ_K denotes the unit-simplex in $\mathbb{R}^K$.

This problem is definitely posed in the primal space: it involves (convex combinations of) objective- and constraint-values. When the objective function $f(x) = b^\top x$ is linear and the constraints $c(x) = Ax - a$ are affine, we recognize *exactly* the master problem (19) $\mathcal{X}_K$ being the convex hull of $\{x_1, \ldots, x_K\}$. In summary:

Fact 27 *Knowing that* (16) *is just an instance of* (1) *with polyhedral/linear data* $\mathcal{X}, f, c$*, solving it by the column-generation master* (19) *amounts to solving the dual* (4) *by the cutting-plane algorithm* (28). □

This supplements Remark 20 (in particular, it makes no difference to have $\mathcal{X}$ or its convex hull as universe, remember Fact 18). As a consequence, we see that

[12] Remember that $g_k \in \partial\theta(u_k)$. An expert in convex analysis will detect the conjugate function: $\theta^*(g_k) = \theta(u_k) - g_k^\top u_k$, which "explains" Everett's Theorem 11.

– the present cutting-plane algorithm,
– the subgradient algorithm of §4.1,
– the variants of the latter (ellipsoid, r-algorithm),
– ACCPM, which will be seen below in §4.3,
– the bundle algorithms of §5,
all of these are various alternatives to solve the same problem: (1) or (16), or at least their convex relaxation.

Reverting the above formulation, we find generalized linear programming:

Fact 28 *The cutting-plane algorithm* (28)*, as well as all of the above-mentioned alternatives, generalize the column-generation mechanism of §2.6 to nonlinear/-nonpolyhedral data in* (16) *– knowing that* (28) *is just the so-called generalized linear programming algorithm.* □

A final word, with relation to Fact 16, going along the end of §4.1. Again suppose the primal points x_k, computed by the oracle at u_k, are stored along with the dual elements $\theta(u_k)$ and g_k. Then an optimal solution $\hat{\alpha}$ of the (bi)dual (30) allows the computation of the primal point $\hat{x} := \sum_k \hat{\alpha}_k x_k$. This point, which actually solves (19), approximates a primal optimal solution as mentioned in §3.2: see (23) again.

4.3 Stability Problems: ACCPM

From a practical point of view, observe that (28) is obviously feasible (take r large enough) but has no reason to be bounded from below: $\hat{\theta}$ has no reason to have a finite minimum. Think of $K = 1$: we get $r_2 = -\infty$ and u_2 at infinity, and even *no matter how close* u_1 is to a minimum of θ. The cutting-plane algorithm is inherently unstable. The mechanism to bound u_K, required by theory, has also a practical role: to stabilize the algorithm.

Although apparently naive, the above argument *is* a serious one and serves as a basis for an impressive counter-example found in [54] (and reproduced in [25, § XV.1.1]). A function is defined, not particularly pathological – essentially $\theta(u) = \max\{0, -1 + \|u\|\}$ –, and for which obtaining the approximate optimality condition $\delta \leqslant \varepsilon^2$, requires $K(\varepsilon) \simeq (1/\varepsilon)^m$ cutting-plane iterations.

By comparison, the ellipsoid algorithm performs as follows: to obtain the same accuracy ε^2 in the worst case (no matter how nasty the oracle θ can be), an order of $m^2 \log(1/\varepsilon) \simeq m \log K(\varepsilon)$ iterations are needed at most[13].

This bad behaviour of the cutting-plane algorithm is due to the fact that the model $\hat{\theta}$ is much too optimistic; something else is needed.

Remark 29 *In nonlinear programming, it is commonly admitted that an optimization algorithm can hardly be good if it is not based on a model approximating*

[13] For bundle methods to come, the worst-case number of iterations is of the order $1/\varepsilon^6$ (see [74,33]). This is definitely worse than ellipsoid, but does not depend on the dimension m of the space.

the objective function to second order. Here $\hat{\theta}$ can be considered as a "global" approximation of θ (in the sense that it provides information all over the space $\mathbb{R}^m$); but it is a good local approximation nowhere. The best we can hope is to have the 1st-order approximation property $\hat{\theta}(u_k + h) = \theta(u_k + h) + o(\|h\|)$ near each reference point u_k.

Unfortunately, if it is conceivable to construct convenient first-order approximations of convex functions, second-order is at present fictional (do not forget that the θ we are minimizing does not even have continuous first derivatives); for efforts along these lines see for example [44,45,40] and the references therein. This is true only for oracle-functions, though; when more information is known from the objective functions, more is possible; see [15, Chap. 14], the more recent [59,52], and Remark 24. □

A stabilizing idea in the spirit of interior-point methods works as follows. Let θ_K^* be the best function-value of Theorem 25 and form the set

$$\begin{aligned} P_K &:= \{(u,r) : \hat{\theta}(u) \leqslant r \leqslant \theta_K^*\} \\ &= \{(u,r) : \theta(u_k) + g_k^\top (u - u_k) \leqslant r \leqslant \theta_K^* \ \text{ for } k = 1, \dots, K\}. \end{aligned}$$

This is a polyhedron in the graph-space $\mathbb{R}^{m+1}$, which we assume bounded (possibly via the mechanism used to bound the cutting-plane iterates).

The cutting-plane algorithm consists in taking for (u_{K+1}, r_{K+1}) a point that is lowest in P_K; according to Nemirovski's counter-example, this is a bad point. A better idea is to take a point that is *central* in P_K. A good concept for this, due to G. Sonnevend [69], is the *analytic center* of P_K: the point maximizing the product of slacks (or sum of their logs) of the constraints defining P_K. Based on this idea, the method ACCPM (analytic center cutting-plane method) has been defined in [21]; see [22] for a recent overview.

To conclude this section, let us mention that the same stability problem exists for subgradient methods of §4.1: g_k jumps along the iterations, reflecting the nondifferentiability[14] of θ. It can even be said that instability is an inherent difficulty with Lagrangian relaxation, and more generally with nonsmooth optimization. Indeed remember Proposition 14: let only to prove optimality of a given u^*, the dual algorithm has to get from the oracle a number (possibly as large as $m+1$) of distinguished primal points x_k, in order to make up the point x^* of Fact 16.

5 Dual Algorithms II: Bundle Methods

Just as ACCPM, bundle methods start from cutting planes and aim at overcoming the instability problem mentioned in §4.3. However, while ACCPM preserves the global character of the cutting-plane model $\hat{\theta}$ (in the sense that P_K does not favour any of the successive interates $u_1, \dots, u_K$), they borrow from nonlinear programming the concept of local approximation – see Remark 29.

[14] This phenomenon is best explained on the column generation model of §2.6: when the $c = c_u$ of Assumption 7(iii) varies, the $\varkappa$ returned by the oracle can only jump from one $\varkappa$ to another.

5.1 The Algorithmic Scheme

The rationale of bundle methods is as follows:
- A *stability center* is chosen; denote it by $\hat{u}$: it is a point one would like u_{K+1} to be close to, its choice will be made precise below.
- As said already, the cutting-plane model $\hat{\theta}$ is much too optimistic: say $\hat{\theta} \ll \theta$ in a neighborhood of $\hat{u}$.
- As a result, minimizing $\hat{\theta}$ gives a u_{K+1} much too far from $\hat{u}$ (a disaster revealed by Nemirovski's counter-example).
- It is therefore advisable to add a positive term to $\hat{\theta}$, so as to improve its local approximation character;
- we choose a quadratic stabilizing term, of the type $\|u - \hat{u}\|^2$,
- whose effect is to pull u_{K+1} toward $\hat{u}$, thereby stabilizing the algorithm.

Using a *spring strength* $t > 0$ (which may depend on the iteration number) aimed at tuning the stabilization effect, the method therefore approximates the true function $\theta(u)$ by the "stabilized" cutting-plane model

$$\breve{\theta}(u) := \hat{\theta}(u) + \frac{1}{2t}\|u - \hat{u}\|^2 , \tag{31}$$

which is minimized at each iteration. Instead of (28), a linearly constrained quadratic problem is solved:

$$\begin{cases} \min r + \dfrac{1}{2t}\|u - \hat{u}\|^2 , \quad (u, r) \in \mathbb{R}^{m+1} , \\ r \geqslant \theta(u_k) + g_k^\top (u - u_k) \quad \text{for } k = 1, \ldots , K . \end{cases} \tag{32}$$

This problem has always a unique optimal solution (u_{K+1}, r_{K+1}). Besides, the two positive numbers

$$\begin{aligned} \delta &:= \theta(\hat{u}) - \hat{\theta}(u_{K+1}) , \\ \breve{\delta} &:= \theta(\hat{u}) - \breve{\theta}(u_{K+1}) = \delta - \frac{1}{2t}\|u_{K+1} - \hat{u}\|^2 \end{aligned} \tag{33}$$

still represent a prediction of the expected decrease $\theta(\hat{u}) - \theta(u_{K+1})$, probably more reasonable than the δ of (29) produced by (28) (remember note 10, p. 136).

To complete the iteration, we must take care of the stability center. For this we test the actual decrease, as compared to δ; say

$$\theta(u_{K+1}) \leqslant \theta(u_K) - \kappa\delta, \tag{34}$$

κ being a fixed tolerance ($\breve{\delta}$ could be preferred to δ but this is a detail: one shows that $\delta \in [\breve{\delta}, 2\breve{\delta}]$). If (34) holds, we set $\hat{u} = u_{K+1}$; in the bundle jargon, this is called a *descent-step*, as opposed to a *null-step*: if (34) does not hold, $\hat{u}$ is left as it is (but $\hat{\theta}$ is enriched). For convergence, κ must be positive. Besides, one easily sees that the descent property (34) equivalently means that the model is accurate enough:

$$\theta(u_{K+1}) \leqslant \hat{\theta}(u_{K+1}) + (1 - \kappa)\delta .$$

Thus, κ must also be smaller than 1, otherwise $\hat{u}$ will never be updated.

Let us now consider the stopping test. Because u_{K+1} minimizes $\breve{\theta}$ instead of $\hat{\theta}$, and because $\breve{\theta}$ need not be smaller than the actual objective function, neither δ nor $\breve{\delta}$ of (29) provide a safe bracket of the optimal value $\min\theta$. Unless one really trusts the approximation $\breve{\theta}$ of (31), stopping simply when δ or $\breve{\delta}$ is small may be unduly optimistic. Something safer is as follows: u_{K+1} minimizes the model (31), which is a convex function, hence $0 \in \partial\breve{\theta}(u_{K+1}) = \partial\hat{\theta}(u_{K+1}) + \frac{1}{t}(u_{K+1} - \hat{u})$. This can be written

$$u_{K+1} - \hat{u} = -t\hat{g} \quad \text{for some } \hat{g} \in \partial\hat{\theta}(u_{K+1}) \;, \tag{35}$$

which reveals an important object: $\hat{g} = (\hat{u} - u_{K+1})/t$, called the *regularized gradient* in the bundle jargon. As shown in Theorem 32 below, it can be computed from the dual of (32).

Thus we can write for all $u \in \mathbb{R}^m$:

$$\begin{aligned} \theta(u) \geqslant \hat{\theta}(u) &\geqslant \hat{\theta}(u_{K+1}) + \hat{g}^\top(u - u_{K+1}) \\ &= \theta(\hat{u}) - \delta + \hat{g}^\top(u - u_{K+1}) \,, \end{aligned} \tag{36}$$

so that we can stop when *both* δ and $\hat{g}$ are small[15]. We summarize various convergence properties of bundle methods, collected from [74,29,8], [25, Chaps. IX, XIII, XV]; see also [36].

Theorem 30 *We assume for simplicity that the stepsize t remains fixed along the iterations; say $t \equiv 1$.*

(i) There holds at each iteration

$$\begin{aligned} \theta(u) &\geqslant \theta(\hat{u}) - \delta + \hat{g}^\top(u - \hat{u}) \\ &\geqslant \theta(\hat{u}) - \delta - \|\hat{g}\|\,\|u - \hat{u}\| \quad \textit{for all } u \in \mathbb{R}^m \,. \end{aligned}$$

(ii) Any of the following equivalent events

$$\hat{g} = 0 \quad \textit{or} \quad \delta = 0 \quad \textit{or} \quad \breve{\delta} = 0 \quad \textit{or} \quad u_{K+1} = \hat{u}$$

guarantees that $\hat{u}$ minimizes θ over $\mathbb{R}^m$.

(iii) The sequences $(\hat{g})$, (δ), $(\breve{\delta})$ and $(\hat{u} - u_{K+1})$ tend to 0.

(iv) The sequence of stability centers is minimizing: $\theta(\hat{u}) \downarrow \min\theta$.

(v) If θ has a nonempty set of minimizing points, then the entire sequence of stability centers $(\hat{u})$ converges to such a minimizing point.

(vi) The events mentioned in (ii) do occur at some finite iteration K when the set of possible answers from the oracle is finite. □

Case (vi) above concerns a finite universe $\mathcal{X}$ in (1) and means that θ is polyhedral; this is for example the case in column generation.

Even though this result considers a fixed stepsize, it should be noted that numerical efficiency depends crucially on a varying t, suitably updated at each iteration. This is for sure a weak point of the approach, although some update strategies do exist: [32,43].

[15] In anticipation of §5.2 below, note that the cutting-plane algorithm of §4.2 produces (36) with $\hat{g} = 0$.

Remark 31 *Having thus derived bundle methods from the cutting-plane method of §4.2, we can also compare* (35) *with the subgradient iteration of §4.1:*
- *Here the next iterate is moved from the stability center $\hat{u}$, instead of the current iterate u_K.*
- *Alternatively, a subgradient method can be seen as a sort of bundle method whithout any check for descent such as* (34)*: one systematically sets $\hat{u} = u_K$.*
- *The move is made in a direction resembling a subgradient. Actually, it can be shown that $\hat{g} \in \partial\theta(\hat{u})$ if t is small enough.*
- *We also mention that the $\hat{g}$ of* (35) *is vaguely related to the direction used in the r-algorithm (not necessarilty with t small). Specifying this link more clearly needs investigation, though.* □

Thus, a subgradient method can in a way be seen as a particular bundle method, obtained for small t. Conversely, it is rather clear in (32) that a large t results in a weak stabilization: we come close to the pure cutting-plane method. Between these two extremes, bundle methods form a whole continuum parameterized by t.

5.2 Primal-Dual Relations: Augmented Lagrangian

Forming the dual of (32) is a good exercise illustrating §2.2, and is quite parallel to the derivation of (30). With nonnegative (bi)dual variables $\alpha \in \mathbb{R}^K$, the (bi)Lagrangian is

$$L'(u, r, \alpha) = r + \frac{1}{2t}\|u - \hat{u}\|^2 + \sum_{k=1}^{K} \alpha_k[-r + \theta(u_k) + g_k^\top (u - u_k)]\,.$$

Minimizing $L'(u, \cdot, \alpha)$ forces $\sum_k \alpha_k = 1$: the α_k's form a set of convex multipliers. Minimizing $L'(\cdot, r, \alpha)$ produces the unique $u = u_\alpha = \hat{u} - tg(\alpha)$, where we have set $g(\alpha) := \sum_k \alpha_k g_k \in \mathbb{R}^m$. Plugging this value into L' gives the (bi)dual function

$$-\frac{t}{2}\|g(\alpha)\|^2 + \sum_{k=1}^{K} \alpha_k \big(\theta(u_k) - g_k^\top u_k\big) + g(\alpha)^\top \hat{u}\,.$$

Now remember that we are actually solving a dual problem: returning to the primal space of (1), we can write $\theta(u_k) = L(x_k, u_k)$ and $g_k = -c(x_k)$, so that $\theta(u_k) - g_k^\top u_k = f(x_k)$. Recalling the notation $\Delta_K \subset \mathbb{R}^K$ for the unit-simplex, the (bi)dual problem can be written

$$\max_{\alpha \in \Delta_K} \sum_{k=1}^{K} \alpha_k f(x_k) - \frac{t}{2}\|g(\alpha)\|^2 + \hat{u}^\top g(\alpha)\,, \quad \sum_{k=1}^{K} \alpha_k c(x_k) = -g(\alpha)\,. \tag{37}$$

Duality between the pair of linear-quadratic problems (32), (37) gives (invoke for example Theorem 21, since (37) has a compact feasible set):

Theorem 32 *Let* $\hat{\alpha}$*) solve* (37)*. The optimal solution* $(u_{K+1}, r_{K+1} = \hat{\theta}(u_{K+1}))$ *of* (32) *is then obtained from* (35) *with* $\hat{g} := g(\hat{\alpha})$*.*

Besides, we have

$$\hat{f} := \sum_{k=1}^{K} \alpha_k f(x_k) = \theta(\hat{u}) - \delta + t\|\hat{g}\|^2 - \hat{u}^\top \hat{g}\,. \qquad \square$$

The interesting point is to compare (37) with (30): while the cutting-plane method imposes the constraint $\sum_k \alpha_k c(x_k) = 0$ at each iteration, a bundle method *relaxes* this constraint via two terms in the objective function:

– one is a *penalization* with the penalty coefficient $t > 0$;

– the other is a *dualization* with the value $\hat{u}$ for the dual variable; incidentally, the term $\sum_k \alpha_k f(x_k) + \hat{u}^\top g(\alpha)$ in (37) can also be written $\sum_k \alpha_k L(x_k, \hat{u})$.

This is the so-called *augmented Lagrangian* technique, well-known in nonlinear programming, and much studied in [3] for example. It combines Lagrangian relaxation with an alternative way of eliminating constraints: penalization.

Remark 33 *As already said on several occasions, a bundle algorithm can be used for column generation. After convergence of the algorithm, the primal point* $\hat{x} := \sum_k \alpha_k x_k$ *of* (37) *can still be constructed. In contrast with the end of §4.2, however, this point is not feasible during the course of the algorithm. Theorem 30 guarantees via* (36) *that this point becomes feasible* asymptotically*: $\hat{g}$ (standing for $g(\hat{x})$ or $A\hat{x} - a$) tends to 0. Besides, $\hat{f}$ (standing for $f(\hat{x})$ or $b^\top \hat{x}$) tends to* $\min \theta$ *if $\hat{g}^\top \hat{u}$ tends to 0 – for example if $\hat{u}$ is* bounded*; in this case, $\hat{x}$ becomes optimal for the relaxed primal problem.* $\square$

Augmented Lagrangian is an important theory, let us consider it from a higher point of view. To eliminate constraints in a general optimization problem such as (1), an alternative to dualizing them is to penalize them: one maximizes without constraints $f(x) - \frac{t}{2}\|c(x)\|^2$ over the whole of $\mathcal{X}$; quite a simple idea. It is not difficult to establish the asymptotic equivalence or this penalized problem with (1), when $t \to +\infty$. Augmented Lagrangian does something more subtle: it applies Lagrangian relaxation to

$$\max f(x) - \frac{t}{2}\|c(x)\|^2\,, \quad x \in \mathcal{X}\,, \quad c(x) = 0\,. \tag{38}$$

This seems of no avail: using simultaneously relaxation and penalization seems simply redundant. However, the situation is substantially different in the dual. In fact duality applied to (38) gives the *augmented Lagrangian*

$$L_t(x, u) := f(x) - \frac{t}{2}\|c(x)\|^2 - u^\top c(x)$$

and the corresponding augmented dual function

$$\theta_t(u) := \max_{x \in \mathcal{X}} L_t(x, u)\,.$$

It can be shown that

(i) if $u = u^*$ minimizes θ of (3), *and if there is no duality gap*, maximizing $L_t(\cdot, u^*)$ can only produce primal points which are feasible, hence optimal: any "parasitic" infeasible primal point is eliminated;

(ii) if $t \to +\infty$, maximizing $L_t(\cdot, u)$ produces again primal optimal solutions, even for bad u.

Concerning (ii), the influence of t can be drastic: compare the next result with Theorem 17.

Fact 34 *Under mild and natural assumptions (always satisfied, for example, if $\mathcal{X}$ is a finite set), augmented Lagrangian cancels out the duality gap: for t large enough,* $\min \theta_t = v(0)$ *of* (24). □

The price to pay is that Assumptions 2 and 7(iii) are usually no longer valid when L is replaced by L_t; as a result, augmented Lagrangian can hardly be used *in practice*. Nevertheless, (i) suggests the ability of augmented Lagrangian to stabilize primal infeasibilities, so that the idea can be translated to the numerical field – and this is what bundle methods do.

For an illustration, consider (16). Using Assumption 7(ii), formulate its convex relaxation in terms of convex multipliers α:

$$\max_{\alpha \in \Delta_\infty} \sum_{k=1}^{\infty} \alpha_k b^\top \varkappa_k \,, \quad \sum_{k=1}^{\infty} \alpha_k A \varkappa_k = a \in \mathbb{R}^m \,. \tag{39}$$

Standard column generation defines approximations $\hat{x}$, with ∞ replaced by a smaller number K; a very natural idea, just requiring an appropriate mechanism to generate $\varkappa_{K+1}$, namely the satellite. The bundle approach takes a multiplier vector $u \in \mathbb{R}^m$, a penalty coefficient $t > 0$, forms the augmented Lagrangian

$$L_t(\alpha, u) = \sum_{k=1}^{\infty} \alpha_k f(x_k) - \frac{t}{2} \Big\| \sum_{k=1}^{\infty} \alpha_k c(x_k) \Big\|^2 - u^\top \sum_{k=1}^{\infty} \alpha_k c(x_k) \,, \tag{40}$$

and makes the following reasoning:

– Standard duality/penalty theory says that minimizing $L_t(\cdot, u)$ is equivalent to (39) in two cases:
  – if u is a dual solution (and $t > 0$ is arbitrary); this results from Theorem 21(i), for example (the unit-simplex Δ_∞ is compact);
  – if $t \to +\infty$ (and u is arbitrary in $\mathbb{R}^m$); this natural result can be proved rather easily.
– Unfortunately, both problems have comparable complexity, so nothing is really gained so far.
– A column-generation-like technique is therefore applied to the minimization of $L_t(\cdot, u)$ (even though it is unconstrained – but one can always set $g := -\sum_k \alpha_k c(x_k)$, considered as a constraint): ∞ is replaced by K to obtain estimates $\hat{x} = \sum_{k=1}^{K} \alpha_k \varkappa_k$; again a very natural idea if the augmented Lagrangian idea is accepted as natural.

– However, the optimal u is not known; hence, simultaneously to the process generating $\hat{x}$, one generates dual estimates $\hat{u}$. In addition to the satellite, a mechanism is required to generate u_{K+1} – this is (32).

Inequality Constraints. As seen in Remark 3, primal inequality constraints entail nonnegative dual variables: (32) is changed to

$$\begin{cases} \min r + \dfrac{1}{2t}\|u - \hat{u}\|^2 \,, \quad (u, r) \in \mathbb{R}^{m+1} \,, \\ r \geqslant \theta(u_k) + g_k^\top (u - u_k) \quad \text{for } k = 1, \ldots, K \,, \\ u \geqslant 0 \end{cases}$$

(knowing that (28) is recovered if $t \to +\infty$). As for the primal space, the reader may check that dualizing the above quadratic problem – with the additional term $s^\top u$ in the (bi)Lagrangian – results in transforming (37) to maximizing over $\alpha \in \Delta_K$ and $s \geqslant 0$

$$\sum_{k=1}^{K} \alpha_k f(x_k) - \frac{t}{2}\|g(\alpha) - s\|^2 + \hat{u}^\top (g(\alpha) - s) \,,$$

where $g(\alpha) := -\sum_{k=1}^{K} \alpha_k c(x_k)$, as before: this formulation reveals the term $\sum_{k=1}^{K} \alpha_k c(x_k) + s$, coming from the constraint $c(x) + s = 0$, $s \geqslant 0$.

This again goes with the general theory: using the slackened formulation of an inequality-constrained primal problem, the augmented Lagrangian is

$$L'_t(x, s, u) = f(x) - \frac{t}{2}\|c(x) + s\|^2 - u^\top (c(x) + s) \,,$$

to be maximized with respect to $x \in \mathcal{X}$ and $s \geqslant 0 \in \mathbb{R}^m$. Incidentally, maximization with respect to s can be carried out explicitly; for given x, we obtain the m-vector $s = s(x)$ whose jth component is $\max\{0, -c_j(x) - u_j/t\}$. With this notation, the augmented dual function is then

$$\theta'_t(u) = \max_{x \in \mathcal{X}} L(x, u) - u^\top s(x) - \frac{t}{2}\|c(x) + s(x)\|^2 \,.$$

5.3 Quadratic Solvers and Poorman Bundle Methods

The main argument against bundle methods is that they replace an "easy" linear program (28) or (30) by a "difficult" quadratic program (32) or (37). This explains various proposals, in which the stabilizer is polyhedral instead of quadratic; the model $\check{\theta}$ of (31) is replaced by, say

$$\check{\theta}(u) := \hat{\theta}(u) + \frac{1}{t}\|u - \hat{u}\|_\infty \,.$$

A similar stabilization was proposed in the seminal *boxstep* method of [50]: the idea was to minimize the model

$$\breve{\theta}(u) := \begin{cases} \hat{\theta}(u) & \text{if } \|u - \hat{u}\|_\infty \leqslant \rho \,, \\ +\infty & \text{otherwise} \,, \end{cases}$$

i.e. to minimize $\hat{\theta}$ over the *trust-region* $\|u-\hat{u}\|_\infty \leqslant \rho$. The purpose of this section is to study the issue.

Why a Quadratic Master? First there are good reasons to use a quadratic term in the model.

(i) One is by analogy with smooth optimization. As mentioned in Remark 29, quadratic terms in the model, incorporating second-order information on the objective function, are crucial for efficiency.

Indeed it is shown in [25, § II.2.2(b)] that even a very naive quadratic term such as in (31) (i.e. an extremely coarse second-order approximation), may improve drastically the performances: the number of iterations may be divided by a factor of 10^3.

(ii) Similarly, for the so-called *composite optimization* problem (which basically means to minimize the maximum of finitely many smooth functions), a good model is the sum of a polyhedral and of a quadratic function, even quite naive as above; see [15, Chap. 14], but also [62, Chap. 3].

By contrast, a polyhedral stabilization – such as in $\breve{\theta}$ of boxstep – is strongly biased: it favors the extreme points of the corresponding polyhedral unit-ball, irrespective of the original problem (think of the basis vectors, which are the extreme points of the ℓ_1 unit-ball; their cosine with the "ideal" direction – pointing to an optimal solution – is most probably of the order $1/n$).

(iii) In our present context, motivated in §4.3, a quadratic term has several advantages.
- It guarantees compactness: the model $\breve{\theta}$ of (31) has always a minimum point, for any $t > 0$; such is not the case with the above $\check{\theta}$, for example.
- It preserves the possible first-order approximation property of the cutting-plane model: $\breve{\theta}(\hat{u}+h) = \hat{\theta}(\hat{u}+h) + o(\|h\|)$; here again, $\check{\theta}$ does not.
- It consistently stabilizes u_{K+1}, no matter how $t > 0$ is chosen, and no matter how $\hat{u}$ is close to a minimum point of θ. By contrast, $\check{\theta}$ may be minimal at $\hat{u}$, an unfortunate event; as for $\breve{\theta}$, its asymptotic efficiency necessitates to have $\rho \to 0$ at a suitable speed (if $\hat{\theta}$ reaches its minimum in the interior of the box of radius ρ centered at $\hat{u}$, the boxstep precaution could just be forgotten).

(iv) The case of ACCPM is interesting. First, it does not cure by itself the compactness problem: a center is not defined for an unbounded set P_K in §4.3. Besides, a naive quadratic term (with $\hat{u}$ fixed to 0) is again beneficial, at least theoretically: see [55].

A prospective argument could be added to the above list: in some future, research in convex analysis might very well produce appropriate quadratic models, less primitive than the mere Euclidean norm in $\breve{\theta}$. Then a quadratic master will become a must: the difference between pure cutting planes of §4.2 and such "second-order like" methods will be as large as, say, the difference between Gauss-Seidel and Newton for systems of equations.

Efficiency of Quadratic Solvers. The belief that a quadratic master cannot be solved as efficiently as a linear one is indeed misleading.

Observe first that linear and quadratic programming call essentially for the same techniques:

– either pivotal ([4], [15, Chap. 10]), the only real difference being that the dimension of the optimal basis is not known in advance;
– or using interior points [57,76].

Actual implementations solve the dual (37), which has a fairly particular form: a fixed simplicial constraint structure, and a Hessian which is simply the Gram matrix $\left[g_k^\top g_{k'}\right]$. Besides, its complexity is just proportional to K, the number of pieces in the bundle: the dimension m of the dual space does not count. Observe for example that, with $K = 1$, we obtain from (35) the explicit solution $u_2 = \hat{u} - tg_1$ (in this case, $\partial\hat{\theta}$ is everywhere reduced to the singleton $\{g_1\}$; alternatively, (37) has the single feasible point $\alpha_1 = 1$).

Another important point is that the present context lends itself to restarts, just as in column generation. From the Kth master (37) to the $(K+1)$st, the differences are:

– one more variable α_{K+1}; setting it to 0 and appending it to the old optimal solution $\hat{\alpha}$ produces a feasible point in the new master (37);
– one more row-column in the Hessian matrix, obtained by computing the $K+1$ scalar products $g_k^\top g_{K+1}$, $k = 1, \ldots, K+1$;
– one more linear term $f(x_{K+1}) - \hat{u}^\top g_{K+1}$;
– possibly a change in the old linear terms $f(x_k) - \hat{u}^\top c(x_k)$, in case the stability center $\hat{u}$ has been updated (descent step);
– a few other minor changes in case the bundle has been cleared – see below.

Among available software exploiting fully these features, let us cite [30,31,16].

Indeed, it is safe to say that the future of nonlinear optimization now requires a substantial development of quadratic programming software, analogous to the tremendous impetus of the last 50 years for linear programming. Such software, incidentally, is also required for "ordinary" smooth optimization problems, which are nowadays solved by the so-called *sequential quadratic programming* approach (SQP); see [15, §12.4], [58, Chap. 18].

Clearing the Bundle: Cheap Quadratic Masters. At least in theory, the quadratic master can be made straigtforward, as the bundle size can be kept as small as 2. In fact, it is always possible to eliminate from the bundle any number of pieces, provided that one keeps a so-called *aggregate* piece, which is the affine function

$$\theta^{\mathrm{a}}(u) := \hat{\theta}(u_{K+1}) + \hat{g}^\top (u - u_{K+1})\,. \tag{41}$$

Using (35), we see that

$$\theta^{\mathrm{a}} \leqslant \theta \quad \text{and} \quad \theta^{\mathrm{a}}(u_{K+1}) = \theta(u_{K+1})\,.$$

The aggregate piece can thus be viewed as another linearization of θ, which can be incorporated into the bundle without harm.

Remark 35 *The above aggregation has a nice primal interpretation. First observe that each piece in the bundle can be expressed in terms of the Lagrangian:*

$$\theta(u_k) + g_k^\top (u - u_k) = L(x_k, u)\,.$$

Next, using complementary slackness in (32) *and remembering the primal points* x_k*, we see that* $\hat\theta(u_{K+1}) = r_{K+1}$ *is equal to*

$$\sum_{k=1}^K \hat\alpha_k \big(\theta(u_k) - g_k u_k + \hat g^\top u_{K+1}\big) \;= \sum_{k=1}^K \hat\alpha_k L(x_k, u_{K+1}) + g_k^\top u_{K+1}\,,$$

so that the aggregate piece

$$\theta^{\mathrm{a}}(u) = \sum_{k=1}^K \hat\alpha_k L(x_k, u) = \sum_{k=1}^K \hat\alpha_k f(x_k) - u \sum_{k=1}^K \hat\alpha_k c(x_k)$$

is a convex combination of Lagrangians. Besides, this writing connotes the primal candidate $\hat x = \sum_k \hat\alpha_k x_k$*, already encountered on several occasions.* □

The aggregate piece summarizes the information contained in the bundle in such an efficient way that it suffices by itself to guarantee convergence of the dual process:

Theorem 36 *The convergence properties (i)–(v) of Theorem 30 remain valid if, prior to any iteration* $K+1$*, the following operations are performed:*
– *any existing piece* $(\theta(u_k), g_k)$ *(possibly all) is cancelled from the bundle;*
– *the aggregate piece* $(\hat\theta(u_{K+1}), \hat g)$ *is inserted;*
– *the new piece* $(\theta(u_{K+1}), g_{K+1})$ *is appended.* □

As a result, the complexity of the master can be controlled quasi at will: any number of pieces can be eliminated, at any time[16]. Naturally, if any eliminated piece k has $\hat\alpha_k = 0$, there is no need to insert the aggregate piece: it is implicitly present. On the other hand, the simplest possible master at step $K+1$ is

$$\min r + \frac{1}{2t}\|u - \hat u\|^2\,, \quad r \geqslant \theta^{\mathrm{a}}(u)\,,\; r \geqslant \theta(u_{K+1}) + g_{K+1}^\top (u - u_{K+1})\,. \tag{42}$$

Theorem 36 is explained as follows. The only existing scheme to prove convergence of a bundle method is to argue that the optimal value in (32), with K replaced by $K+1$, is certainly larger than that of (42), which has less constraints.

[16] Incidentally, this is by no means shared by the cutting-plane methods of §4.2: although some works exist to eliminate inactive pieces ([70]), a bundle of dimension at least $m+1$ must of course be maintained (otherwise $\hat\theta$ cannot be bounded from below).

Then it is rather easy to analyze the *poorman bundle method*, in which the bundle is fully cleared at each iteration and the master (42) is solved. Convergence results for this method are *a fortiori* true for any "richer" strategy.

In other words: no theoretical result establishes a difference between maximal and minimal bundles. This is rather frustrating, as a rich bundle method can reasonably be expected to converge faster than a poor one. This belief can be assessed only by computational experience, though, which is the subject of the next section.

5.4 Numerical Illustrations

For illustration, we applied bundle method to the Held-Karp dual of a traveling salesman problem [23], with datasets from tsplib; all the graphs were complete, with a dimension m of the dual space equal to the number of nodes. The dual algorithm was that of [43], with the quadratic solver of [30]. The oracle to compute the 1-tree for given u was very elementary, inspecting systematically the $O(m^2)$ possibilities, and storing no distance matrix: the m^2 (Euclidean) distances were recomputed everytime they were needed. Table 1 gives the results obtained when the algorithm was pushed to optimality[17]. The runs were performed with a maximum bundle size of 1000.

Table 1. Bundle method pushed to full optimality

Problem	m	$-\theta(u_K)$	K	K_{d}	ℓ	CPU	% master	% oracle
gr120	120	1606.3125	132	40	42	1s	35	44
pcb442	442	50499.499	556	109	116	151s	64	28
pcb1173	1173	56350.993	502	65	81	438s	31	64
pcb3038	3038	136587.50	4212	324	367	10h	30	56
fnl4461	4461	181569.21	6965	470	361	30h?	8?	87?

In this table, K is the number of iterations needed (each iteration is one resolution of the master (37) + one maximization of the Lagrangian); K_{d} is the number of descent steps, updating the stability center $\hat{u}$; ℓ is the number of active constraints in the final master (32). The last three columns concern computing times. Some observations are worth mentioning.

– We also tested fl1577 but failed: after some 80 000 iterations (about a week of computer time), we still had $\theta(u_k) \simeq -21\,833$, a value not close to $\min \theta$.
– The last three columns were obtained by the unix command `gprof` and are indicative only. In particular, `gprof` gave for fnl4461 an implausible computing time of 4200s; hence our question marks in the corresponding entries of Table 1.

[17] The stopping test is explained in [41]. Given a threshold ε set by the user, two corresponding tolerances are computed for δ of (29) and $\|\hat{g}\|$ of (35), implying that θ is minimized within relative accuracy ε. All the runs reported in Table 1 had $\varepsilon = 10^{-6}$ and terminated with $\delta \leqslant 10^{-3}$ and $\|\hat{g}\| \leqslant 10^{-5}$.

– The CPU time spent in Lagrangian maximizations (last column) becomes heavier when m is larger; by comparison, the time spent in the master problem (37) is light during the first iterations, when $k \ll 1\,000$.
– The values of ℓ are interesting. With a pure cutting-plane method, ℓ could only equal $m+1$. Now ℓ has several interpretations:
  – It is the dimensionality of $\partial\theta(\hat{u})$; if θ were differentiable, we would have $\ell = 1$, the oracle answering some $g_k = 0$.
  – From a primal point of view, ℓ is the number of x_k's making up the optimal solution $\hat{x}$ of the relaxed primal problem.
  – Because $\ell \ll m$, we have to conclude that the set of dual optimal solutions has fairly large dimensionality[18].
  – Alternatively, the optimal set of the primal relaxed problem has fairly small dimensionality.

  Note, however, that all this is highly informal, and subject to some approximation anyway; Table 4 shows that it could be true only in a perfect world with everything computed exactly – including the solution of (32).

Using the same test-problems, we report the same information with the poorman bundle method of §5.3. More precisely, the method is still that of [43], with a bundle systematically reduced to three elements: the aggregate and the new ones, as required by Theorem 36, and the g returned by the oracle at the current $\hat{u}$ (this element is useful for the t-update of [43]). Table 2 reads as Table 1, knowing that the algorithm was interrupted "manually" when 4 exact digits were obtained in the objective function.

Table 2. Poorman bundle method

Problem	$-\theta(u_K)$	K	K_{d}	CPU	% master	% oracle
gr120	1606.1746	173	26	1s	1	89
pcb442	50495.121	233	29	19s	2	94
pcb1173	56345.576	236	26	155s	1	94
pcb3038	136573.85	6920	482	5h45	0	94
fnl4461	181551.12	411	41	45min	0	95

For a more complete comparison, Table 3 reports the number of iterations respectively needed by the two versions (poor and rich), to obtain 2, 3, 4 exact digits (i.e. relative accuracies from 10^{-2} to 10^{-4}; $u_1 = 0$ is already 10^{-1}-optimal in these problems). The last double column summarizes the computing time to reach 10^{-3} relative accuracy. In each double column, the cheap version is reported first.

Our final experiments illustrate how the algorithm proves optimality of a given $\hat{u}$. Still the same test-problems are used, and the bundle method is run

[18] We have $\dim \partial\theta(\hat{u}) = \ell - 1$. Because θ is piecewise affine, $\theta(u) = \theta(\hat{u}) + \theta'(\hat{u}, u - \hat{u})$ for $u - \hat{u}$ small enough; and $\theta'(\hat{u}, u - \hat{u}) = 0$ if $u - \hat{u} \perp \partial\theta(\hat{u})$.

Table 3. Comparison of poor and rich versions

Problem	10^{-2}		10^{-3}		10^{-4}		CPU (10^{-3})	
gr120	21	24	93	72	173	102	0s	0s
pcb442	24	23	131	79	233	261	10s	6s
pcb1173	27	22	104	93	236	187	61s	63s
pcb3038	26	28	128	146	6920	782	540s	630s
fnl4461	23	21	126	120	411	811	760s	800s

starting from an optimal u_1 (obtained from the runs in Table 1); $t > 0$ is kept small. Then only null-steps are performed and every sampling point u_k is close to $\hat{u} = u_1$, so that $g_k \in \partial\theta(\hat{u})$[19].

Table 4. Proving optimality

Problem	$-\theta(u_K)$	K	K_{d}	ℓ^*	final $\|\hat{g}\|$
gr120	1606.3125	47	18	42	2×10^{-15}
pcb442	50500.	105	24	105	3×10^{-10}
pcb1173	56361.	93	27	91	5×10^{-11}
pcb3038	136587.5	700	39	570	2×10^{-6}
fnl4461	181569.21	357	35	357	1×10^{-10}

The results are given in Table 4. Because we are dealing with approximations only, the tests serve as purification: the starting u_1 is not exactly optimal, a few descent-steps may occur, the u_K's may not all be in an appropriate face of $\operatorname{epi}\theta$, the final ℓ-value may not be the same as in Table 1. This is why we report again the final θ_K, the number K_{d} of descent-steps, and ℓ. The latter is denoted by ℓ^*, to be distinguished from the ℓ of Table 1. Of course, we must have $K \geqslant \ell^*$; actually, $K - \ell^*$ represents the number of "errors" made by the algorithm in identifying the necessary optimal faces; said otherwise, the algorithm computes K subgradients at $\hat{u}$ (or close to), ℓ^* of which are finally used to make up the 0 vector. Table 4 gives also the norm of this (approximate) 0 vector, whose accuracy is assessed by the fact that each g_k from the oracle is an integer vector, and therefore has norm larger than 1.

Concluding Remarks. Inspection of the above numerical results suggests a number of comments.

(i) Bundle methods appear as reasonably efficient alternatives to traditional algorithms for Lagrangian relaxation, hence for column generation. Note in particular that all our reported experiments used a fairly simple oracle,

[19] i.e. $(u_k, \theta(u_k))$ lies in a face of $\operatorname{epi}\theta$ containing the optimal $(\hat{u}, \min\theta)$.

delivering one column at a time. Notwithstanding, serious comparisons are still to be made between bundle, subgradient, cutting planes and ACCPM.

(ii) A dual problem of large dimension m is likely to be difficult. However, ℓ^*, which resembles the dimensionality of an optimal $\partial\theta$, seems a lot more important: a problem with small ℓ^* – or with a solution set of large dimension – seems relatively "easy".

(iii) Just as finiteness, piecewise affinity is a concept that should be taken with care: there is no clearcut between a piecewise affine θ (finitely many possible answers from the oracle) and a truly nonlinear one (a continuum of possible answers). This is illustrated by some observations, in our present context:
- Any run of gr120, with various values given to various parameters (starting point, stopping tolerance, bundle size, etc.), invariably produced $\ell^* = 42$. Besides, $\|\hat{g}\|$ decreased regularly along the iterations down to about 10^{-5}, and then fell abruptly to 10^{-15}, clearly revealing finite convergence.
- For all other test-problems, variations in the parameters produced fluctuations of about 10% in the final ℓ^*. This occurred even with pcb1173 (seemingly easy), and also when starting from an optimal u_1. Besides, $\|\hat{g}\|$ decreased regularly to its minimal value around 10^{-10}. Such a behaviour definitely connotes a truly nonlinear problem.

(iv) The poorman variant is surprisingly efficient. With Remark 31 in mind, it assesses older works such as [6,5]. Good behaviour of cheap bundle methods was already observed in [34] and related works. Such variants are of course very attractive, as they use little computing time, and are relatively easy to encode (although a good knowhow is strongly advised: nonsmooth optimization is by no means an easy exercise).

The above observation can be turned the other way: "richman" bundle methods are disappointing (observe in particular the disaster for fnl4461 in Table 3, in which the richer version is the slower). The information contained in a bundle of size 1000 has to be worth something: if this information does not help to improve efficiency, for sure it is misused; and of course the culprit is the Euclidean norm (a "poorman" stabilizer). This is why research has been and still is conducted, to define richer stabilizers: see Remark 29 again. Some works already exist proposing stabilizers of the form $\frac{1}{2}(u-\hat{u})^\top M_k(u-\hat{u})$: see [51,17,53], and also [48], where convexity of θ is not even used. On this "quasi-second-order" subject, the final word is far from being said.

(v) By contrast, Table 4 illustrates quite an interesting behaviour: since ℓ^* is a lower bound for K, the algorithm could hardly do better in the situation of the experiments[20]. See also the results in [25, § IX.3.3], which display a similar behaviour. The role of the stabilizer is here blatantly demonstrated.

[20] However pcb3038 shows that fast convergence is never guaranteed. Our trials to obtain$\|\hat{g}\|$ around 10^{-10} remained fruitless: the quadratic solver ran into difficulties, K was much larger than ℓ^*, the algorithm had to be restarted several times, etc.

Indeed the conditions are here most favourable for θ to look like a piecewise affine function: many answers from the oracle are eliminated from the picture. Yet the far-from-piecewise-affine model $\breve{\theta}$ of (31) allows an accurate description of θ.

We believe that here lies the crux of nonsmooth optimization. To minimize a nonsmooth function such as θ, two processes are involved:

– To diminish θ down to its minimal value; in Lagrangian relaxation, this means to generate best possible upper bounds for the primal problem.
– To prove optimality of $\hat{u}$, i.e. to genarate an $\hat{x}$ which solves the relaxed primal problem. In cutting-plane methods of §4.2, each $\hat{x}$ is feasible, what is needed is to increase the corresponding value $\sum_k \alpha_k f(x_k)$ to $\theta(\hat{u})$. In a bundle method, the problem is mainly to decrase $\|\hat{g}\|$ down to 0.

Because of our two observations (iv) and (v), the two processes seem fairly unrelated and call for fairly different tools. The difficulty is to find an appropriate model (such as $\hat{\theta}$ or $\breve{\theta}$) in which they groove together harmoniously.

References

1. F. Alizadeh. Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM Journal on Optimization*, 5(1):13–51, 1995.
2. K. Anstreicher and L.A. Wolsey. On dual solutions in subgradient opimization. Unpublished manuscript, CORE, Louvain-la-Neuve, Belgium, 1993.
3. D.P. Bertsekas. Projected Newton methods for optimization problems with simple constraints. *SIAM Journal on Control and Optimization*, 20:221–246, 1982.
4. M.J. Best. Equivalence of some quadratic programming algorithms. *Mathematical Programming*, 30:71–87, 1984.
5. U. Brännlund. *On relaxation methods for nonsmooth convex optimization.* PhD thesis, Royal Institute of Technology - Stockholm, 1993.
6. P.M. Camerini, L. Fratta, and F. Maffioli. On improving relaxation methods by modified gradient techniques. *Mathematical Programming Study*, 3:26–34, 1975.
7. E. Cheney and A. Goldstein. Newton's method for convex programming and Tchebycheff approximations. *Numerische Mathematik*, 1:253–268, 1959.
8. R. Correa and C. Lemaréchal. Convergence of some algorithms for convex minimization. *Mathematical Programming*, 62(2):261–275, 1993.
9. A. Decarreau, D. Hilhorst, C. Lemarechal, and J. Navaza. Dual methods in entropy maximization. application to some problems in crystallography. *SIAM Journal on Optimization*, 2(2):173–197, 1992.
10. I. Ekeland and R. Temam. *Convex Analysis and Variational Problems.* North Holland, 1976; reprinted by SIAM, 1999.
11. Y.M. Ermol'ev. Methods of solution of nonlinear extremal problems. *Kibernetica*, 2(4):1–17, 1966.
12. H. Everett III. Generalized lagrange multiplier method for solving problems of optimum allocation of resources. *Operations Research*, 11:399–417, 1963.
13. J.E. Falk. Lagrange multipliers and nonconvex programs. *SIAM Journal on Control*, 7(4):534–545, 1969.
14. S. Feltenmark and K. C. Kiwiel. Dual applications of proximal bundle methods, including lagrangian relaxation of nonconvex problems. *SIAM Journal on Optimization*, 10(3):697–721, 2000.

15. R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, Chichester (second edition), 1987.
16. A. Frangioni. Solving semidefinite quadratic problems within nonsmooth optimization algorithms. *Computational Operational Research*, 23(11):1099–1118, 1996.
17. M. Fukushima and L. Qi. A globally and superlinearly convergent algorithm for nonsmooth convex minimization. *SIAM Journal on Optimization*, 6:1106–1120, 1996.
18. C. Garrod and J.K. Percus. Reduction of the N-particle variational problem. *Journal of Mathematical Physics*, 5(12), 1964.
19. A.M. Geoffrion. Lagrangean relaxation for integer programming. *Mathematical Programming Study*, 2:82–114, 1974.
20. M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 6:1115–1145, 1995.
21. J.L. Goffin, A. Haurie, and J.Ph. Vial. Decomposition and nondifferentiable optimization with the projective algorithm. *Management Science*, 38(2):284–302, 1992.
22. J.L. Goffin and J.Ph. Vial. Convex nondifferentiable optimization: a survey focussed on the analytic center cutting plane method. to appear in Optimization Methods and Software; also as HEC/Logilab Technical Report 99.02, Univ. of Geneva, Switzerland.
23. M. Held and R. Karp. The travelling salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 1:6–25, 1971.
24. C. Helmberg and F. Rendl. A spectral bundle method for semidefinite programming. *SIAM Journal on Optimization*, 10(3):673–696, 2000.
25. J.-B. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms*. Springer Verlag, Heidelberg, 1993.
26. J.-B. Hiriart-Urruty and C. Lemaréchal. *Fundamentals of Convex Analysis*. Springer Verlag, Heidelberg, 2001. to appear.
27. R.A. Horn and Ch.R. Johnson. *Matrix Analysis*. Cambridge University Press, 1989. (New edition, 1999).
28. J. E. Kelley. The cutting plane method for solving convex programs. *J. Soc. Indust. Appl. Math.*, 8:703–712, 1960.
29. K.C. Kiwiel. *Methods of Descent for Nondifferentiable Optimization*. Lecture Notes in Mathematics 1133. Springer Verlag, Heidelberg, 1985.
30. K.C. Kiwiel. A method for solving certain quadratic programming problems arising in nonsmooth optimization. *IMA Journal of Numerical Analysis*, 6:137–152, 1986.
31. K.C. Kiwiel. A dual method for certain positive semidefinite quadratic programming problems. *SIAM Journal on Scientific and Statistical Computing*, 10:175–186, 1989.
32. K.C. Kiwiel. Proximity control in bundle methods for convex nondifferentiable minimization. *Mathematical Programming*, 46(1):105–122, 1990.
33. K.C. Kiwiel. Efficiency of proximal bundle methods. *Journal of Optimization Theory and Applications*, 104(3):589–603, 2000.
34. K.C. Kiwiel, T. Larsson, and P.O. Lindberg. The efficiency of ballstep subgradient level methods for convex optimization. *Mathematics of Operations Research*, 24(1):237–254, 1999.
35. T. Larsson, M. Patriksson, and A.B. Strömberg. Ergodic, primal convergence in dual subgradient schemes for convex programming. *Mathematical Programming*, 86(2):283–312, 1999.
36. C. Lemaréchal, A.S. Nemirovskii, and Yu.E. Nesterov. New variants of bundle methods. *Mathematical Programming*, 69:111–148, 1995.

37. C. Lemaréchal, Yu. Nesterov, and F. Oustry. Duality gap analysis for problems with quadratic constraints, 2001. In preparation.
38. C. Lemaréchal and F. Oustry. Semi-definite relaxations and lagrangian duality with application to combinatorial optimization. Rapport de Recherche 3710, Inria, 1999. `http://www.inria.fr/rrrt/rr-3710.html`.
39. C. Lemaréchal and F. Oustry. Nonsmooth algorithms to solve semidefinite programs. In L. El Ghaoui and S-I. Niculescu, editors, *Advances in Linear Matrix Inequality Methods in Control*, Advances in Design and Control, 2, pages 57–77. SIAM, 2000.
40. C. Lemaréchal, F. Oustry, and C. Sagastizábal. The $\mathcal{U}$-lagrangian of a convex function. *Transactions of the AMS*, 352(2):711–729, 2000.
41. C. Lemaréchal, F. Pellegrino, A. Renaud, and C. Sagastizábal. Bundle methods applied to the unit-commitment problem. In J. Doležal and J. Fidler, editors, *System Modelling and Optimization*, pages 395–402, 1996.
42. C. Lemaréchal and A. Renaud. A geometric study of duality gaps, with applications. *Mathematical Programming*, 90(3):399–427, 2001.
43. C. Lemaréchal and C. Sagastizábal. Variable metric bundle methods: from conceptual to implementable forms. *Mathematical Programming*, 76(3):393–410, 1997.
44. C. Lemaréchal and J. Zowe. Some remarks on the construction of higher order algorithms in convex optimization. *Applied Mathematics and Optimization*, 10(1):51–68, 1983.
45. C. Lemaréchal and J. Zowe. The eclipsing concept to approximate a multi-valued mapping. *Optimization*, 22(1):3–37, 1991.
46. L. Lovász. On the Shannon capacity of a graph. *IEEE Transactions on Information Theory*, IT 25:1–7, 1979.
47. D.G. Luenberger. *Optimization by Vector Space Methods*. Wiley, New York, 1969.
48. L. Luksan and J. Vlcek. A bundle-newton method for nonsmooth unconstrained minimization. *Mathematical Programming*, 83A(3):373–391, 1998.
49. T.L. Magnanti, J.F. Shapiro, and M.H. Wagner. Generalized linear programming solves the dual. *Management Science*, 22(11):1195–1203, 1976.
50. R.E. Marsten, W.W. Hogan, and J.W. Blankenship. The boxstep method for large-scale optimization. *Operations Research*, 23(3):389–405, 1975.
51. R. Mifflin. A quasi-second-order proximal bundle algorithm. *Mathematical Programming*, 73(1):51–72, 1996.
52. R. Mifflin and C. Sagastizábal. On $\mathcal{VU}$-theory for functions with primal-dual gradient structure. *SIAM Journal on Optimization*, 11(2):547–571, 2000.
53. R. Mifflin, D.F. Sun, and L.Q. Qi. Quasi-Newton bundle-type methods for nondifferentiable convex optimization. *SIAM Journal on Optimization*, 8(2):583–603, 1998.
54. A.S. Nemirovskii and D. Yudin. Informational complexity and efficient methods for the solution of convex extremal problems. *Ékonomika i Mathematicheskie Metody*, 12:357–369, 1976. (in Russian. English translation: *Matekon*, 13, 3-25).
55. Yu.E. Nesterov. Complexity estimates of some cutting plane methods based on the analytic barrier. *Mathematical Programming*, 69(1):149–176, 1995.
56. Yu.E. Nesterov and A.S. Nemirovskii. *Interior-Point Polynomial Algorithms in Convex Programming*. Number 13 in SIAM Studies in Applied Mathematics. SIAM, Philadelphia, 1994.
57. Yu.E. Nesterov and A.S. Nemirovskii. *Interior-Point Polynomial Algorithms in Convex Programming*. SIAM Studies in Applied Mathematics 13. SIAM, Philadelphia, 1994.

58. J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer Verlag, New York, 1999.
59. F. Oustry. A second-order bundle method to minimize the maximum eigenvalue function. *Mathematical Programming*, 89(1):1–33, 2000.
60. S. Poljak, F. Rendl, and H. Wolkowicz. A recipe for semidefinite relaxation for (0,1)-quadratic programming. *Journal of Global Optimization*, 7:51–73, 1995.
61. B.T. Polyak. A general method for solving extremum problems. *Soviet Mathematics Doklady*, 8:593–597, 1967.
62. B.N. Pshenichnyi. *The Linearization Method for Constrained Optimization*. Springer Verlag, 1994.
63. C.R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell Scientific Publications, New York, 1993.
64. N. Shor. Utilization of the operation of space dilatation in the minimization of convex functions. *Cybernetics*, 6(1):7–15, 1970.
65. N.Z. Shor. *Minimization methods for non-differentiable functions*. Springer Verlag, Berlin, 1985.
66. N.Z. Shor and A.S. Davydov. Method of opbtaining estimates in quadratic extremal probems with boolean variables. *Cybernetics*, 21(2):207–211, 1985.
67. N.Z. Shor and N.G. Zhurbenko. A method for minimization, using the space-dilation operation in the direction of difference between two gradient sequences. *Cybernetics*, 7(3):450–459, 1971.
68. V.N. Solov'ev. The subdifferential and the directional derivatives of the maximum of a family of convex functions. *Izvestiya: Mathematics*, 62(4):807–832, 1998.
69. G. Sonnevend. An "analytical centre" for polyhedra and new classes of global algorithms for linear (smooth, convex) programming. In A. Prékopa, J. Szelezsan, and B. Strazicky, editors, *Proc. 12th IFIP Conf. System Modelling and Optimization*, L.N. in Control and Information Sciences, pages 866–875. Springer Verlag, Berlin, 1986.
70. D.M. Topkis. A note on cutting-plane methods without nested constraint sets. *Operations Research*, 18:1216–1224, 1970.
71. T.Terlaky. On ℓ_p programming. *European Journal of Operational Research*, 22:70–100, 1985.
72. H. Uzawa. Iterative methods for concave programming. In K. Arrow, L. Hurwicz, and H. Uzawa, editors, *Studies in Linear and Nonlinear Programming*, pages 154–165. Stanford University Press, 1959.
73. L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Review*, 38(1):49–95, 1996.
74. P. Wolfe. A method of conjugate subgradients for minimizing nondifferentiable functions. *Mathematical Programming Study*, 3:145–173, 1975.
75. L.A. Wolsey. *Integer Programming*. Wiley-Interscience, 1998.
76. S.J. Wright. *Primal-Dual Interior-Point Methods*. SIAM Publication, Philadelphia, 1997.

Branch-and-Cut Algorithms for Combinatorial Optimization and Their Implementation in ABACUS

Matthias Elf[1], Carsten Gutwenger[2], Michael Jünger[1], and Giovanni Rinaldi[3]

[1] Universität zu Köln, Pohligstraße 1, D-50969 Köln, Germany
[2] Stiftung `caesar`, Friedensplatz 16, D-53111 Bonn, Germany
[3] IASI-CNR, Viale Manzoni 30, I-00185 Roma, Italy

Abstract. Branch-and-cut (-and-price) algorithms belong to the most successful techniques for solving mixed integer linear programs and combinatorial optimization problems to optimality (or, at least, with certified quality). In this unit, we concentrate on sequential branch-and-cut for hard combinatorial optimization problems, while branch-and-cut for general mixed integer linear programming is treated in [⟶ Martin] and parallel branch-and-cut is treated in [⟶ Ladányi/Ralphs/Trotter]. After telling our most recent story of a successful application of branch-and-cut in Section 1, we give in Section 2 a brief review of the history, including the contributions of pioneers with an emphasis on the computational aspects of their work. In Section 3, the components of a generic branch-and-cut algorithm are described and illustrated on the traveling salesman problem. In Section 4, we first elaborate a bit on the important separation problem where we use the traveling salesman problem and the maximum cut problem as examples, then we show how branch-and-cut can be applied to problems with a very large number of variables (branch-and-cut-and-price). Section 5 is devoted to the design and applications of the ABACUS software framework for the implementation of branch-and-cut algorithms. Finally, in Section 6, we make a few remarks on the solution of the exercise consisting of the design of a simple TSP-solver in ABACUS.

1 Our Most Recent Story

Branch-and-cut has become a widely used method for the solution of hard integer or mixed integer problems. We refer to a recent survey of Caprara and Fischetti [12] for a view on its wide range of applications. In this chapter, the emphasis will be mostly on combinatorial optimization problems. Before going into a brief historical tour through the main algorithmic achievements that lead to or are connected with branch-and-cut, hoping to get the reader's interest before entering into more technical topics, we want to report on our most recent experience [29] with this method. The little story that follows is not only an example where branch-and-cut was quite useful, but shows also how combinatorial optimization can sometimes provide an excellent modeling tool to solve real world problems.

M. Jünger and D. Naddef (Eds.): Computat. Comb. Optimization, LNCS 2241, pp. 157–222, 2001.
© Springer-Verlag Berlin Heidelberg 2001

During the seminar "Constraint Programming and Integer Programming" in Schloß Dagstuhl, Germany, 17–21 January 2000, the participants tried to identify problems for which a fruitful interaction/competition of constraint programming techniques and integer/combinatorial optimization techniques appears challenging and is likely to enhance the interaction of both communities. One problem area the participants agreed upon consists of various feasibility/optimization problems occurring in scheduling sports tournaments. The break minimization problem for sports leagues was addressed by both communities during the workshop, in particular by Jean Charles Régin of the constraint programming community and by Michael Trick of the integer programming/combinatorial optimization community.

1.1 Break Minimization

We deal with the situation where in a sports league consisting of an even number n of teams each team plays each other team once in $n-1$ consecutive weeks. Each game is played in one of the two opponents' home towns and each feasible schedule has the following properties:

F1 For each team, the teams played in weeks $1, \ldots, n-1$ are a permutation of all other teams.
F2 If in week w team i plays team j "at home" ("+") then team j plays team i in week w in i's town, i.e., "away" ("−").

Fig. 1 shows two possible schedules for a league of eight teams. The rows show the game plan for each team, column 1 displays a team, column $w \in \{2, \ldots, n\}$ shows the opponent in week $w-1$, "+", and "−", respectively, indicate if the game is at home or away. In sports scheduling it is considered undesirable

1 : +2 −3 −4 +5 +6 −7 +8	1 : +8 +3 −5 +7 −2 +4 −6
2 : −1 +7 −5 +4 −3 −8 +6	2 : +7 −8 +4 −6 +1 −3 +5
3 : −7 +1 +8 −6 +2 +5 −4	3 : +6 −1 +8 +5 −7 +2 −4
4 : +5 −8 +1 −2 +7 −6 +3	4 : −5 +7 −2 −8 +6 −1 +3
5 : −4 −6 +2 −1 +8 −3 +7	5 : +4 −6 +1 −3 +8 +7 −2
6 : +8 +5 −7 +3 −1 +4 −2	6 : −3 +5 −7 +2 −4 −8 +1
7 : +3 −2 +6 −8 −4 +1 −5	7 : −2 −4 +6 −1 +3 −5 +8
8 : −6 +4 −3 +7 −5 +2 −1	8 : −1 +2 −3 +4 −5 +6 −7
(a)	(b)

Fig. 1. Two feasible schedules for eight teams

if any team plays two consecutive games either both at home or both away. Such a situation is called a *break*. We are given a feasible tournament schedule without home-away assignment and our task is to find a feasible home-away

assignment that minimizes the number of breaks. The schedule of Fig. 1(a) imposes 8 breaks whereas the schedule in Fig. 1(b) imposes only 6 breaks. In both cases, the number of breaks is minimum. Schreuder [66] has shown that, for an even number n of teams, it is always possible to construct a schedule that allows a home-away assignment with the minimum number of $n-2$ breaks and he has given an efficient algorithm to compute such a schedule (along with the home-away assignment). However, sport tournament schedules are subject to a number of requirements, among them restrictions such as "geographically close teams should not play at home during the same week". Some authors (like Schreuder [66]) propose to start with an optimum schedule with $n-2$ breaks and incorporate the additional requirements at the cost of more breaks, others (like Régin [62,63] and Trick [67]) propose to consider a schedule without home-away assignment that obeys the various (often not formally describable) side conditions and compute a home-away assignment as to minimize the number of breaks. It is the latter attitude we take here.

Régin formulated a constraint programming model with 0-1-variables and was able to solve instances up to size 20. Trick introduced an integer programming formulation and was able to solve instances up to size 22.

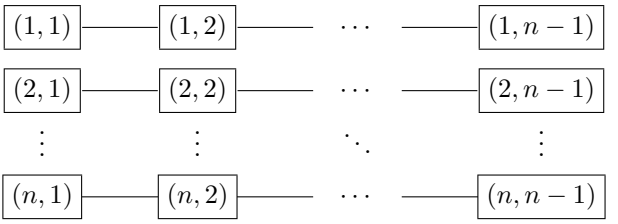
The complexity status of the break minimization problem has not yet been determined (to the best of our knowledge), we believe it is NP-hard.

1.2 From Minimizing Breaks to Maximizing Cuts

Given a feasible tournament schedule without home-away assignment

$$\begin{array}{l} 1 : t_{11}\ t_{12}\ \dots\ t_{1,n-1} \\ 2 : t_{21}\ t_{22}\ \dots\ t_{2,n-1} \\ \quad\vdots\quad\ \vdots\quad\ \vdots\quad \ddots\quad\ \vdots \\ n : t_{n1}\ t_{n2}\ \dots\ t_{n,n-1} \end{array}$$

in which $t_{ij} \in \{1, 2, \dots, n\} \setminus \{i\}$ is the opponent of team i in week j, we construct an undirected graph $G = (V, E)$ with a node $v = (i, j) \in V$ for $i \in \{1, 2, \dots, n\}$ and $j \in \{1, 2, \dots, n-1\}$. There is an edge in E between nodes (i, j) and (k, l) in V if and only if $i = k$ and $1 \le j = l - 1 \le n - 1$. I.e., $G = (V, E)$ is as follows:



A cut in $G = (V, E)$, i.e., an edge set C of the form $C = \delta(W)$, $W \subseteq V$, where $\delta(W) = \{e \in E \mid v \in W,\ w \in V \setminus W\}$, partitions V into $V = V^+ \cup V^-$ ($V^+ \cap V^- = \emptyset$), where V^+ and V^- are called the different *shores* of the cut. Any

home-away assignment corresponds to a cut in G such that, by [F2], $(i,j) \in V^+$ if and only if $(t_{ij},j) \in V^-$ and $|E|-|C|$ is the number of breaks. This condition is modeled by stipulating that (i,j) and (t_{ij},j) belong to different shores of the cut. We could model this by assigning a capacity of 1 to all edges of G introduced so far and adding edges with a value of $M \geq n(n-2)+1$ between all $\frac{n(n-1)}{2}$ such pairs of nodes: Fig. 2 shows the graph G resulting from this transformation for the instance given in Fig. 1(a).

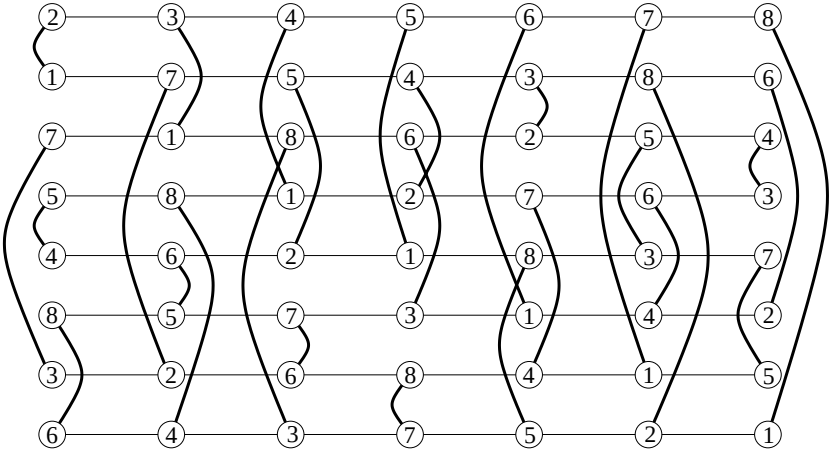


Fig. 2. Result of the big-M transformation

The choice of M would trivially guarantee a correct model. Namely, if the maximum capacity cut with these edge weights has value Δ, then assigning "+" to t_{ij} with $(i,j) \in V^+$ and "−" to t_{ij} with $(i,j) \in V^-$ gives a schedule with the minimum number of $\Delta - \frac{n(n-1)}{2}M$ breaks. But, due to a result of Barahona and Mahjoub [10] we can do much better. If we switch the signs of the capacities of all edges in the star of any node v in G, the maximum capacity cut induces a maximum capacity cut with the original capacities in which v changes shores and all other nodes stay at their shore.

Therefore, for each edge with capacity M we switch the capacity of the star of one of its end-nodes. The resulting edge with capacity $-M$ will be in no maximum cut, so we can contract the edge. We obtain a maximum cut instance with $\frac{n(n-1)}{2}$ nodes and $n(n-2)$ edges with capacities either 1 or -1. The result for our example is shown in Fig. 3, in which we have (arbitrarily) chosen to switch the cut of the lower indexed vertex each time.

The same graph is displayed in Fig. 4 along with a cut of maximum capacity 22 that is indicated by white and grey nodes, respectively.

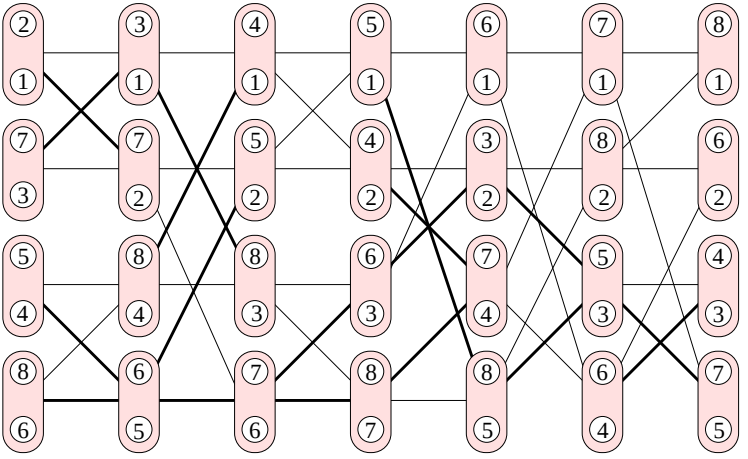


Fig. 3. Result of the transformation

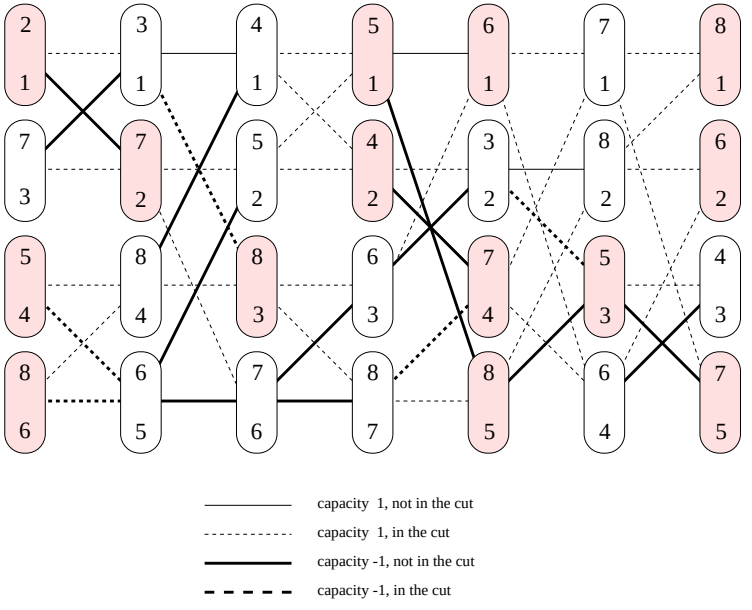


Fig. 4. A cut of maximum capacity 22

By backward transformation, this cut corresponds to the home-away assignment displayed in Fig. 1(a) with 8 breaks, which proves our claim that 8 breaks is minimum for this instance.

1.3 Computational Results

For the computation of maximum capacity cuts we have used an implementation of the branch-and-cut algorithm described in [9] that was re-implemented by Martin Diehl using the ABACUS software [44], version 2.3 with Cplex 6.5 as an LP-Solver. The same implementation has been used successfully in, e.g., [25, 26] for computing ground states of Ising spin glasses.

The instances were created by computing optimum ($(n-2)$-breaks schedules) by Schreuder's procedure and permuting the columns randomly. For each site, we created five random schedules and applied the branch-and-cut algorithm. The results displayed in Table 1 were obtained on a 296 MHz Sun Ultra SPARC machine.

Table 1. Computational Results

size	time (sec)			no. breaks	
n	minimum	average	maximum	minimum	maximum
14	0.2	0.26	0.5	24	26
16	0.2	0.77	2.5	26	34
18	1.0	2.30	18.3	38	44
20	1.4	8.10	49.2	46	56
22	2.9	10.20	44.4	56	64
24	6.7	85.73	507.5	66	78
26	15.9	465.36	5185.7	82	92

We also considered one real world instance, namely the current Bundesliga (first national German soccer league) instance. There are 18 teams and we found that the schedule is already optimum at 16 breaks!

We do not have access to the instances used in the computational studies by Régin and Trick, however, at the time of writing this, it seems that Trick's approach is slightly ahead in solving 22 teams instances in about 1 hour on a 266 MHz machine. Apparently, our approach is able to handle larger instances easily.

2 A Bit of History

Mathematical Programming, originating as a scientific discipline in the late forties, is concerned with the search for optimal decisions under restrictions. The most prominent mathematical models include linear programming, (mixed) integer linear programming and combinatorial optimization with linear objective

function. A surprisingly large variety of problems in economics, mathematics, computer science, operations research and the natural sciences can be captured by these models and effectively solved by appropriate software. Years before computer science emerged as a scientific discipline, the protagonists of mathematical programming, who were mainly applied mathematicians or mathematically oriented economists, aimed at the development of algorithms that could solve the problem instances, in the beginning primarily economic and military planning applications, by hand calculations and, when electronic computers became available in the early fifties, via computer software.

Linear Programming

The father of linear programming is George B. Dantzig, who proposed the model

$$\begin{aligned} &\text{maximize } c^T x \\ &\text{subject to } Ax \leq b \\ &\qquad\qquad x \geq 0 \end{aligned} \tag{1}$$

where $c \in \mathbb{R}^n, A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$, and invented the celebrated simplex algorithm for its solution. In the first sentence of his textbook "Linear Programming and Extensions" [20] he states: "The final test of a theory is its capacity to solve the problems which originated it" and before he proceeds to the acknowledgments, he writes: "The viewpoint of this work is constructive. It reflects the beginning of a theory sufficiently powerful to cope with some of the challenging decision problems upon which it was founded." Not only the model and the simplex algorithm have pertained to this day, but also the philosophy he summarizes in these two sentences remained a leitmotif for mathematical programmers until today, as Michel Balinski puts it in [6]: "First the real problems, then the theory ... ", and we can safely add: " ... and then implementing and testing it on the real problems." The simplex algorithm was implemented in the US National Bureau of Standards on the Standards Eastern Automatic Computer (SEAC) as early as 1952, and computational testing and comparison with competing methods was performed in a way that meets today's standards for such experiments. Slightly later, around 1954, William Orchard-Hays of the RAND Corporation designed the first commercial implementations of the (revised) simplex algorithm for the IBM CPC, IBM 701 and IBM 704 machines. Also direct commercial applications in the oil industry were started as early as 1952 by A. Charnes, W.W. Cooper and B. Mellon [13].

Mixed Integer Programming

It became soon clear, though, that in real problems integrality of some of the decision variables is required. If an optimal plan of activities requires using 1.3 airplanes, this makes no sense. Also, very often yes/no decisions are desired that can be modeled using binary variables. So the (mixed) integer (binary) linear programming model is the linear programming model in which (some)

variables are required to take integral (binary) values. In 1958, Ralph Gomory [32] presented an elegant algorithmic solution to the integer linear programming problem that was later refined to the mixed integer case. Interestingly, he not only invented the "cutting plane" method and proved its finiteness and correctness, but he also implemented it in the same year on an E 101 and an IBM 704 computer in the then brand-new FORTRAN language, in order to see how it performs. Unfortunately, the early experiments where not very satisfactory in terms of practical efficiency, and Gomory's cutting plane method, even though appreciated for its theoretical beauty, was not recommended for practical computations until only a few years ago Balas, Ceria, Cornuéjols, and Natraj [4] gave computational evidence that it should be reconsidered. Instead, the branch-and-bound method proposed by A.H. Land and A.G. Doig in 1960 [47] became the method of choice in all commercial computer codes that were provided by many computer companies starting with linear programming in the sixties and mixed integer programming in the seventies.

Combinatorial Optimization

In combinatorial optimization with a linear objective function, the task consists of selecting from a family of subsets $\mathcal{F} \subseteq 2^E$ of a finite ground set E one $F \in \mathcal{F}$ that maximizes (minimizes) a linear objective function $\sum_{e \in F} c_e$ for coefficients $c_e \in \mathbb{R}, e \in E$. Mathematically, this is trivial, because an optimizing set F can be found by finite enumeration, however such a strategy is clearly unsatisfiable for practical computation. Some of the finest algorithms in computer science and mathematical programming that have been formulated for problems like the minimum spanning tree problem, the shortest path problem, or the matching problem by Kruskal [46], Dijkstra [27] and Edmonds [28], respectively, fit into this framework, as well as many others in combinatorics and (hyper-) graph theory.

In our examples, the finite ground set E consist of the edges of a graph G and the feasible solutions are the spanning trees, the paths connecting two specified nodes and the matchings in G, respectively.

Branch-and-Cut

Unlike these examples, most examples with practical applications have turned out to be NP-hard. Nevertheless, an obvious connection to binary integer programming leads to an algorithmic methodology that has been found out to be able to solve real world instances to optimality anyway. Any $F \subseteq E$ is represented by its *characteristic vector* $\chi^F \in \{0,1\}^E$ in which $\chi^F_e = 1$ if and only if $e \in F$. Passing from the feasible subsets to their characteristic vectors, it is usually not hard to define the problem as a binary linear programming problem, whereas it is usually a long way, theoretically and in terms of implementation effort, to make the well developed linear programming techniques exploitable in effective computation.

We demonstrate this on a prominent example, the traveling salesman problem (TSP), in which the task is to find a Hamiltonian cycle ("tour") with minimum total edge weight in a complete undirected graph. Incidentally, this is the problem for which now commonly used optimization techniques were first outlined by G.B. Dantzig, D.R. Fulkerson and S.M. Johnson in 1954 [21]. If x_{ij} is a variable corresponding to the edge connecting nodes i and j, and c_{ij} is the weight (length) of this edge, then

$$\begin{array}{lll} \text{minimize} & \sum_{ij \in E} c_{ij} x_{ij} & \\ \text{subject to} & \sum_{j \in V} x_{ij} = 2 & \text{for all } i \in V \\ & \sum_{i \in S, j \in V \setminus S} x_{ij} \geq 2 & \text{for all } S \subset V,\ 3 \leq |S| \leq \left\lfloor \frac{|V|}{2} \right\rfloor \\ & 0 \leq x_{ij} \leq 1 & \text{for all } ij \in E \\ & x_{ij} \in \{0,1\} & \text{for all } ij \in E \end{array} \tag{2}$$

is an integer programming formulation in which the equations make sure that in the solution every node has degree two and the inequalities, called *subtour elimination constraints*, guarantee that we do not obtain a collection of subtours. The variables with value 1 in an optimum solution of this integer program are in one-to-one correspondence with the edges of an optimum tour. Discarding the integrality conditions, we obtain a linear programming relaxation (subtour relaxation) that can be used in a branch-and-bound environment: A simple scheme would solve the relaxation at every subproblem in which some variables have been set to 0 or 1. If the solution is not integral, two new subproblems, with one non-integral variable set to 1 in one and to 0 in the other, are created in a branching step. However, this linear programming relaxation contains an exponential number of constraints ($\Theta(2^n)$) for an instance on n nodes. This leads to the idea to solve the relaxation with a small subset of all constraints, check if the optimum solution violates any other of the constraints, and if so, append them to the relaxation. Thus we get a cutting plane method at each node of the enumeration tree that arises by branching.

This method is called *branch-and-cut*. It was first formulated for and successfully applied to the linear ordering problem by Grötschel, Jünger, and Reinelt in [34]. The first state-of-the-art branch-and-cut algorithm was published by Padberg and Rinaldi in [56] where it was used to solve large TSP instances. The major new features of the Padberg-Rinaldi algorithm are the branch-and-cut approach in conjunction with the use of column/row generation/deletion techniques, sophisticated separation procedures and an efficient use of the LP optimizer.

There are many ways to strengthen the subtour relaxation by further classes of inequalities. Identifying some of them that violate a given fractional solution is called the *separation problem*. This problem can be solved in polynomial time for the subtour elimination constraints. Of course, it would be desirable to compute a complete description by linear equations and inequalities of the convex hull of the characteristic vectors of tours, which is the *TSP polytope*

$P^n_{\text{TSP}} = \text{conv}\{\chi^T \in \{0,1\}^{E_n} \mid \chi^T$ is the characteristic vector of a Hamiltonian cycle in $K_n\}$. Here, $K_n = (V_n, E_n)$ denotes the complete undirected graph on n nodes. Such a description exists by classical theorems of Minkowski and Weyl. However, results of Karp and Papadimitriou [45] make it unlikely that we can compute it except for very small n. Even when we restrict ourselves to non-redundant inequality systems, in which each inequality defines a *facet* of the polytope, i.e., a proper face of maximum dimension, the number of needed inequalities is enormous, e.g., 42,104,442 for $n = 9$ and at least 51,043,900,866 for $n = 10$ [16,15].

The field of polyhedral combinatorics deals with identifying subsystems of such complete and non-redundant linear descriptions. In order to make a subclass computationally exploitable, we must devise according separation algorithms, or at least useful heuristics, if the problem is proven to be or appears to be hard.

Computer implementations for branch-and-cut algorithms for the TSP have been devised by a few research groups, the first, in which the term "branch-and-cut" was used for the first time, by Padberg and Rinaldi [55]. Such programs solve most instances with a few hundred cities routinely and also some instances with a few thousand cities, see [39] for a recent survey of the theory and practice of this method for the TSP, [41] for an account how the same methodology can be applied to other combinatorial optimization problems, and [12] for a recent annotated bibliography on branch-and-cut algorithms. The first essential ingredient that makes such an approach work consists of a theoretical part that requires creative analytical investigations on identifying appropriate classes of inequalities (polyhedral combinatorics) and the design of separation algorithms. Together with the implementation of the separation algorithms this is highly problem specific (see Section 4.1). The second ingredient is integrating the separation software into a cutting plane framework and this into an enumerative frame. This latter part is much less problem specific but requires a considerable implementation effort that can be reduced by an appropriate software system.

Branch-and-Price

We use the *binary cutting stock*, another example of a combinatorial optimization problem, in order to introduce a special version of "branch-and-cut", namely *branch-and-price*. In branch-and-price the cut generation phase never takes place, but only columns are dynamically added to the LP relaxation at every node of the enumeration tree. Further details on this approach will be given in Section 4.2.

In the binary cutting stock problem, a set of n rolls of lengths $a_1, a_2, \ldots a_n$ has to be cut out of base rolls with length L. The problem is to determine a cutting strategy that minimizes the number of base rolls used. In 1961, P.C. Gilmore and R.E. Gomory [31] proposed the following model: The vector $b \in \{0,1\}^n$ represents a cutting pattern for a base roll if $\sum_{i=1}^n a_i b_i \leq L$. Let the columns of the matrix $B \in \{0,1\}^{n \times m}$ represent all possible cutting patterns. Then the problem can be modeled as the following binary linear programming problem:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{m} z_j \\ \text{subject to} \quad & \sum_{j=1}^{m} b_{ij}\, z_j = 1 \qquad \text{for all } i \in \{1, 2, \ldots, n\} \\ & 0 \le z_j \le 1 \qquad \text{for all } j \in \{1, 2, \ldots, m\} \\ & z_j \in \{0, 1\} \quad \text{for all } j \in \{1, 2, \ldots, m\} \end{aligned} \tag{3}$$

In any feasible solution $z_j = 1$ if and only if the j-th cutting pattern is used. In contrast to our previous example in which we had a polynomial number of variables and an exponential number of constraints, here we have the opposite situation. If we solve the problem on a small subset of the columns, we have to make sure that the missing columns can be safely assumed to be associated with 0 components of the optimum vector z. The simplex algorithm does not only give us a basic feasible solution (geometrically this corresponds to a vertex of the polyhedron defined by the restrictions) but also a short certificate for optimality consisting of a quantity y_i for each row i such that $\sum_{i=1}^{n} b_{ij} y_i \le 1$ for all cutting patterns $B_{\cdot j} = (b_{1j}, b_{2j}, \ldots, b_{nj})^T$ that are present in the chosen subset. In order to determine if the same relation holds for all missing cutting patterns as well we can solve the knapsack problem

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} y_i b_i \\ \text{subject to} \quad & \sum_{i=1}^{n} a_i b_i \le L \\ & b_i \in \{0, 1\} \end{aligned} \tag{4}$$

for which effective pseudo-polynomial algorithms exist. If the maximum is at most 1, our solution is optimum for the complete problem, otherwise the optimum pattern $b_1, b_2, \ldots, b_n$ found by the algorithm is appended as a new column to the formulation. This process of evaluating the missing columns is called "pricing" in linear programming terminology, and embedding the method into an enumerative frame leads, as we already said, to a *branch-and-price* algorithm. Cutting and pricing can be combined (and they are in all published state-of-the-art algorithms for the optimum solution of the TSP) even when the problem involves, like in this example, exponentially many variables. This methodology is called *branch-and-cut-and-price* and will be described in more detail in Section 4.2.

3 Components of Branch-and-Cut

Figure 5 shows a flowchart of a generic branch-and-bound algorithm for a minimization problem. A basic branch-and-cut algorithm is a branch-and-bound algorithm in which the bounds are solutions of LP-relaxations that are iteratively strengthened by problem specific cutting planes at every node of the enumeration tree. This feature incurs several technicalities that make the design and implementation of branch-and-cut algorithms a nontrivial task. In this section

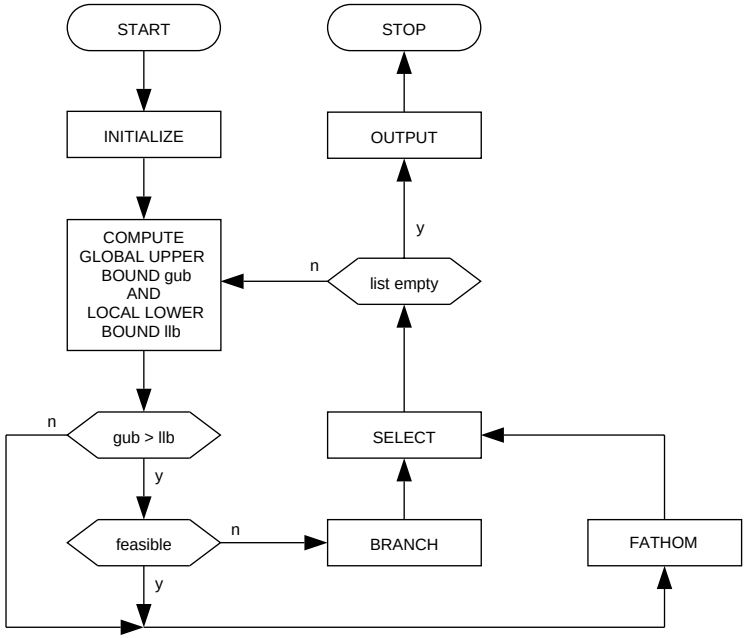


Fig. 5. Flowchart of a branch-and-bound algorithm for minimization problems.

we will present a basic branch-and-cut algorithm, address such technical details and give some ideas for an efficient implementation that proved to be useful in practice.

A first outline of a basic branch-and-cut algorithm is given in the flowchart of Fig. 6 in which the dashed clusters correspond to boxes in the flowchart of Fig. 5. Roughly speaking, the two leftmost of the four columns describe the cutting plane phases within a single subproblem, the third column shows the preparation and execution of a branching operation, and in the rightmost column, the fathoming of a subproblem is performed. We give informal explanations of all steps of the flowchart together with some problem specific details. In the remainder of this section the underlying optimization problem is always assumed to be a minimization problem. Moreover, during the description of the basic version of the algorithm we assume that in all its phases the set of variables remains unchanged. In particular, each LP problem has a fixed number of columns, while the number of rows increases or decreases after the modules called SEPARATE and ELIMINATE have been executed (see Section 3.3). The full version of the algorithm, where also columns are added to the LP, is described in Section 3.5.

Although most of the components of the presented algorithmic framework are problem independent in general, we choose the TSP as our main example. This has several reasons. First, the TSP is probably one of the most prominent combinatorial optimization problems, then it is the prototype problem for which in [55]

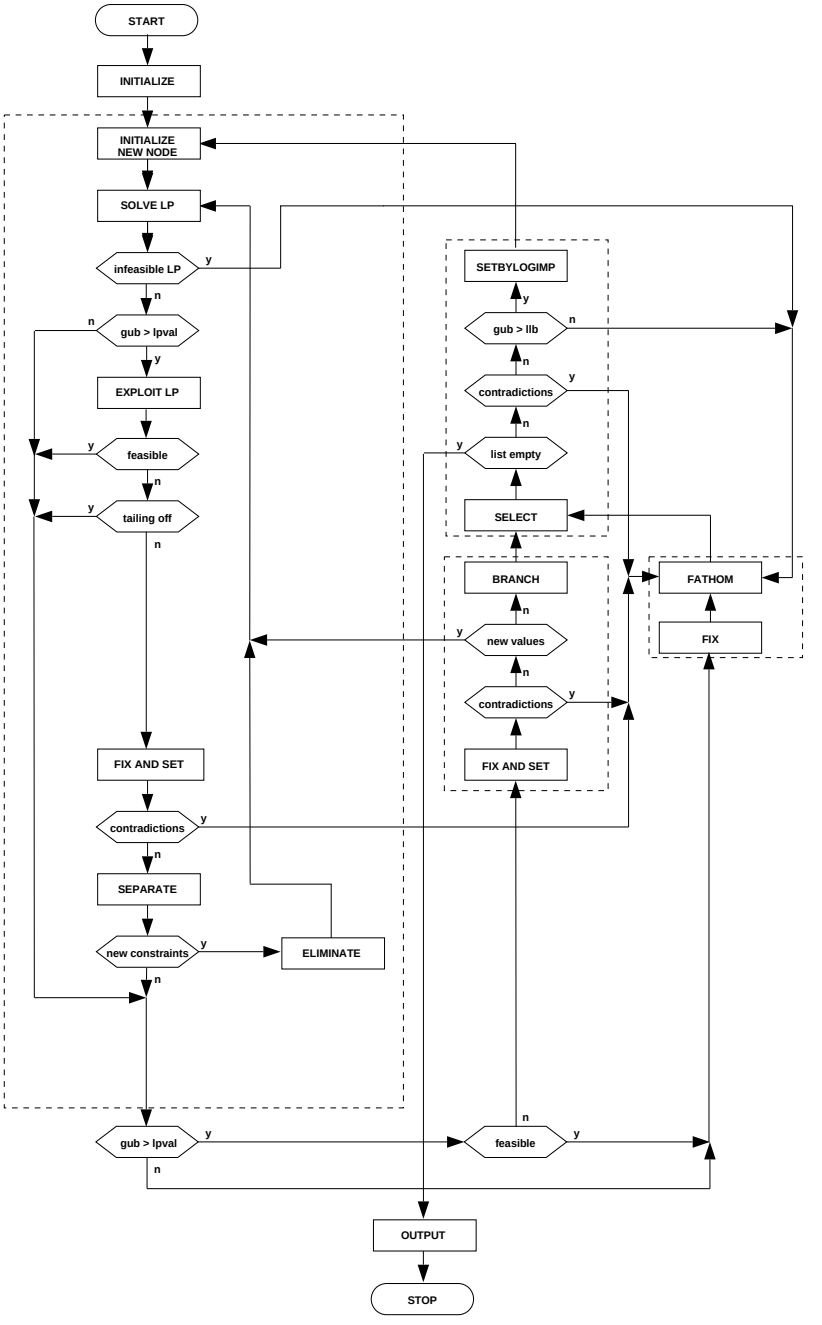


Fig. 6. Flowchart of a basic branch-and-cut algorithm.

and [56] the superiority of branch-and-cut over the other known approaches was shown, and finally implementations of state of the art TSP solver like the one of Jünger/Naddef/Reinelt/Thienel [52] or Applegate/Bixby/Chvátal/Cook [2] (see also [⟶ Applegate/Bixby/Chvátal/Cook] contain all algorithmic techniques we want to demonstrate. Adaptions of these techniques can easily be applied to branch-and-cut algorithms for other combinatorial optimization problems.

In our description, we proceed as follows. First we describe the enumerative part of the algorithm, i.e., we discuss in detail how branching and selection operations can be performed. Then we explain the work done in a subproblem of the enumeration. We also discuss sparse graph techniques which lead to column generation.

There are two major ingredients of a branch-and-bound algorithm, the computation of global upper and local lower bounds. The lower bounds are produced by performing an ordinary cutting plane algorithm for each subproblem. Two basic techniques for the computation of upper bounds (corresponding to feasible solutions of the original problem) are currently being used. The first method is to compute a good upper bound by some heuristics before the root node of the complete branch-and-cut tree is processed. Later this bound can only be improved, if the solution of a linear program is the characteristic vector of a better feasible solution. The other method is the computation of upper bounds in the cutting plane phase by exploiting fractional LP-solutions. This technique may require more running time spent for heuristics, yet may decrease the total running time, since the size of the enumeration tree may be smaller. For the TSP, we describe how the LP-solution can be utilized in oder to find good feasible solutions, i.e., upper bounds.

3.1 Terminology

Before going into details, we have to define some terminology that is used not only in this section but also in Section 5 where we discuss the object oriented software framework ABACUS which implements the generic branch-and-cut algorithm of Fig. 6. Due to a corresponding naming scheme in ABACUS every algorithmic component or variable of the described algorithm is easily identified as a module, variable or data structure in the software.

Since in a branching step like in a branch-and-bound algorithm two (or more) new subproblems are generated, the set of all subproblems can be represented by a binary (k-nary) tree, which we call the *branch-and-cut tree*. Hence we call a subproblem a *branch-and-cut node*. Fig. 7 shows an example of a branch-and-cut tree. We distinguish between four types of branch-and-cut nodes. The node which is currently being processed is called the *current branch-and-cut node*. The other unfathomed leaves of the branch-and-cut tree still have to be processed and are called the *active nodes*. Finally, there are the already processed *non-active nodes*. A non-active node can either be *fathomed* or *not fathomed*.

Each variable has one of the following attributes during the computation: `atlowerbound`, `basic`, `atupperbound`, `settolowerbound`, `settoupperbound`, `fixedtolowerbound`, `fixedtoupperbound`. When we say that a variable is *fixed*

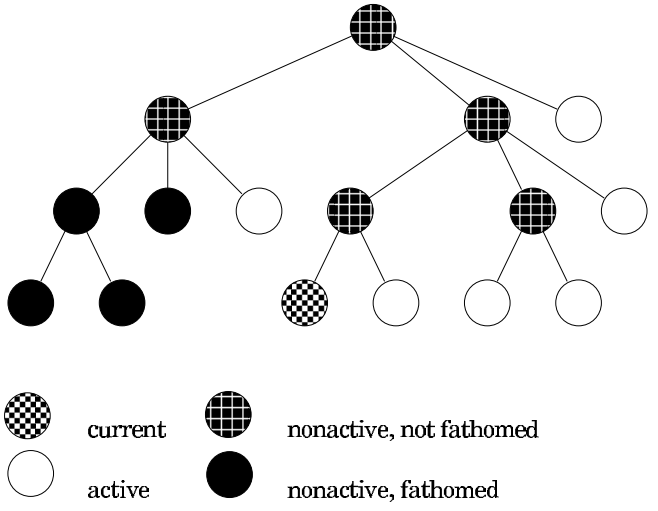


Fig. 7. A Branch-and-cut tree.

to zero (lower bound) or one (upper bound), it means that it is at this value for the rest of the computation. If it is *set* to zero (lower bound) or one (upper bound), this value remains valid only for the current branch-and-cut node and all branch-and-cut nodes in the subtree rooted at the current one in the branch-and-cut tree. The conditions for fixing and setting variables will be explained later in Section 3.2. The meanings of the other attributes are obvious: As soon as an LP has been solved, each variable which has not been fixed or set receives one of the attributes `atlowerbound`, `basic` or `atupperbound` by the revised simplex method with lower and upper bounds.

Finally, the variable `lpval` always denotes the optimum value of the last LP that has been solved, which is also a local lower bound `llb` for the currently processed node, the global variable `gub` (global upper bound) gives the value of the currently best known feasible solution. The minimum lower bound of all active branch-and-cut nodes and the current branch-and-cut node is the global lower bound `glb` for the whole problem. The subtree rooted at the highest common ancestor of all active and the current branch-and-cut nodes is called the *remaining branch-and-cut tree*. Therefore, we call this highest common ancestor also the *root of the remaining branch-and-cut tree* and the local lower bound of this node is called `rootlb`. The difference between `glb` and `rootlb` will be discussed below.

Like in branch-and-bound terminology we call a subproblem *fathomed*, if the local lower bound `lpval` of this subproblem is greater than or equal to the global upper bound `gub`, or if the subproblem becomes infeasible (e.g., if branching variables have been set in a way that the subproblem does not contain any

feasible solution), or if the subproblem is solved, i.e., the solution of the LP-relaxation is a feasible solution of the original problem.

In many applications all objective function coefficients are integer. In that case all feasible solutions have an integer value. Therefore, all terms of the computation that express a lower bound may be rounded up, e.g., one can fathom a node with global upper bound `gub` and local lower bound `llb`, if $\lceil$`llb`$\rceil \geq$ `gub`. For some specially structured problems it might be valid to round up the local lower bound more than one unit, for example to the next even (or odd) integer. These extended roundings should always be applied since tightening the bounds may be essential for practical efficiency.

The branch-and-cut algorithm consists of three different parts: The enumerative frame, the computation of lower bounds and the computation of upper bounds. It is easy to identify the boxes of the flowchart of Fig. 5 with the dashed boxes of the flowchart of Fig. 6.

The central part is the lower bounding part that is performed after the selection of a new current subproblem. It consists of trying to solve the current problem by optimizing over LP-relaxations that are tightened by adding cutting planes. This bounding part is left,

- if the local lower bound is greater than or equal to the global upper bound,
- if the LP-solution is the characteristic vector of a feasible solution,
- if no more cutting planes can be generated,
- if infeasibility of the current subproblem is detected,
- if the upper bound does not decrease significantly, although cutting planes are added (`tailing off`).

It is advantageous, although not necessary for the correctness of the algorithm, to reenter the bounding part if variables are fixed or set to new values by FIX AND SET, instead of creating new subproblems in BRANCH.

3.2 Enumerative Frame

The enumerative frame consists of all parts of the branch-and-cut algorithm except the bounding part (the leftmost dashed box of Fig. 6).

Initialize. During the computation the algorithm stores a set of active branch-and-cut nodes. After input of the problem, the set of active branch-and-cut nodes is initialized as the empty set. To initialize the global upper bound `gub`, feasible solutions are computed by some heuristic methods. For the TSP, we can construct a tour with the *nearest neighbor heuristic* and improve it with a *Lin-Kerninghan* procedure [48]. Afterwards the root node of the complete branch-and-cut tree becomes the current branch-and-cut node which is now processed by the bounding part.

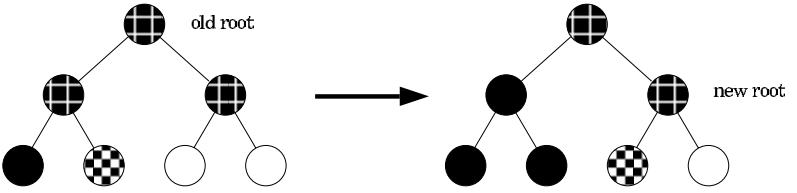


Fig. 8. Root change of a branch-and-cut tree.

Bounding. The computation of the lower and upper bounds is outlined in the sections 3.3 and 3.4. We continue the explanation of the enumerative frame at the ordinary exit of the bounding part (at the end of the first column of the dashed bounding box). If the current branch-and-cut node cannot contain a feasible solution that is better than the best known one (`lpval` $\geq$ `gub`), or the final LP-solution is the characteristic vector of a feasible solution (`feasible`), the node is fathomed. Otherwise a branching operation and the selection of another branch-and-cut node for further processing (third column of the flowchart) is prepared.

Fix and Set. The routine FIX AND SET of Fig. 6 consists of the four procedures FIXBYREDCOST, FIXBYLOGIMP, SETBYREDCOST and SETBYLOGIMP. If a branching operation is prepared, and the current branch-and-cut node is the root node of the branch-and-cut tree, the reduced cost of the non-basic variables obtained from the LP-solver can be used to fix them forever at their current values by the routine FIXBYREDCOST. Namely, if for an edge e the variable x_e is non-basic has value $\bar{x}_e$ and reduced cost is r_e, we can fix x_e to zero if $\bar{x}_e = 0$ and `rootlb` $+ r_e >$ `gub` and we can fix x_e to one if $\bar{x}_e = 1$ and `rootlb` $- r_e >$ `gub`.

During the computational process, the value of `gub` decreases, so that at some later point in the computation, one of these criteria can be satisfied, even though it is not satisfied at the current point of the computation. Therefore, each time when we get a new root of the remaining branch-and-cut tree, we make a *list of candidates for fixing* of all non-basic variables along with their values (0 or 1) and their reduced costs and update `rootlb`. We get a new root of the remaining branch-and-cut tree, if all nodes in all subtrees except one subtree of the old root are fathomed (see Fig. 8).

Since storing these lists in every node, which might eventually become the root node of the remaining active nodes in the branch-and-cut tree, would use too much memory space, we process the complete bounding part a second time for the node, when it becomes the new root. If we could initialize the constraint system for the recomputation by those constraints that were present in the last LP of the first processing of this node, we would need only a single call of the simplex algorithm. However, this would require too much memory. So we initialize the constraint system with the constraints of the last solved LP. As some facets

are separated heuristically, it is not guaranteed that we can achieve the same local lower bound as in the previous bounding phase. Therefore we not only have to use the reduced costs and statuses of the variables of this recomputation, but also the corresponding local lower bound as `rootlb` in the subsequent calls of the routine FIXBYREDCOST. If we initialize the basis by the variables contained in the best known solution and call the primal simplex algorithm, we can avoid phase 1 of the simplex method. Of course this recomputation is not necessary for the root of the complete branch-and-cut tree, i.e., the first processed node. The list of candidates for fixing is checked by the routine FIXBYREDCOST whenever it has been freshly compiled or the value of the global upper bound `gub` has improved since the last call of FIXBYREDCOST.

FIXBYREDCOST may find that a variable can be fixed to a value opposite to the one it has been set to (`contradiction`). This means that earlier in the computation, somewhere on the path of the current branch-and-cut node to the root of the branch-and-cut tree, we have made an unfavorable decision which led to this setting either directly in a branching operation or indirectly via SETBYREDCOST or SETBYLOGIMP (to be discussed below).

Before starting a branching operation and if no contradiction has occurred, some fractional (basic) variables may have been fixed to new values (0 or 1). In this case we solve the new LP rather than performing the branching operation.

Fixbylogimp. After variables have been fixed by FIXBYREDCOST, we call FIXBYLOGIMP. In contrast to reduced cost fixing this is not problem independent. For the TSP, this routine might try to fix more variables by logical implication as follows: If two edges incident to a node v have been fixed to 1, all other edges incident to v can be fixed to 0. Or it might fix the edge connecting two given nodes to 0 if they are connected by a path of edges fixed to 1. Like in FIXBYREDCOST, contradictions to previous variable settings may occur. If variables are fixed to new values, we proceed as explained in FIXBYREDCOST.

In principle also fixing or setting variables to zero could have logical implications. If all incident edges of a node but two are fixed or set to zero, these two edges can be fixed or set to one. However, this occurs quite rarely and can therefore be disregarded.

Setbyredcost. While fixings of variables are globally valid for the whole computation, variable settings are only valid for the current branch-and-cut node and all branch-and-cut nodes in the subtree rooted at the current branch-and-cut node. SETBYREDCOST sets variables by the same criteria as FIXBYREDCOST, but based on the local reduced cost and the local lower bound `llb` of the current subproblem rather than "globally valid reduced cost" and the lower bound of the root node `rootlb`. Contradictions are possible if in the meantime the variable has been fixed to the opposite value. In this case the current branch-and-cut node is fathomed.

Setbylogimp. This routine is called whenever SETBYREDCOST has successfully set variables, as well as after a SELECT operation. It tries to set more variables by logical implication as follows: If two edges incident to a node v have been set or fixed to 1, all other edges incident to v can be set to 0 (if not fixed already). Like in SETBYREDCOST, all settings are associated with the current branch-and-cut node. If variables are set to new values, we proceed as explained in FIXBYREDCOST.

After the selection of a new node in SELECT, we check if the branching variable of the father is set to 1 for the selected node. If this is the case, SETBYLOGIMP may also set additional variables.

Branch. Branching in general is splitting the current subproblem in two or more new one. There are many different strategies to achieve this. For example:

- a fractional 0/1 variable is set to 0 and 1
- upper and lower bounds for integer variables are changed
- dividing the polytope by hyperplanes
- other problem specific strategies

If we choose the first method some variable is chosen as the *branching variable* and two new branch-and-cut nodes, which are the two sons of the current branch-and-cut node, are created and added to the set of active branch-and-cut nodes. In the first son the branching variable is set to 1, in the second one to 0. If no constraints of the integer programming formulation are violated, then there is at least one fractional variable that is a reasonable candidate for a branching variable. However if a constraint of the integer programming formulation is violated, it is possible that all variables have an integral LP-value, yet the LP-solution is not a feasible solution of the original problem. In this case a variable with an integral LP-value has to be chosen as branching variable.

There is a variety of different strategies for the selection of the branching variable, so that we can present here only some of them. Let $\overline{x}$ be the solution of the last solved LP.

1. Select a variable with value close to 0.5 that has a big objective function coefficient in the following sense. Find L and H with $L = \max\{\overline{x}_e \mid \overline{x}_e \leq 0.5, e \in E\}$ and $H = \min\{\overline{x}_e \mid \overline{x}_e \geq 0.5, e \in E\}$. Let $C = \{e \in E \mid 0.75L \leq \overline{x}_e \leq H + 0.25(1-H)\}$ be the set of variables with value "close" to 0.5. From the set C the variable with maximum cost is selected, i.e., with maximum objective function coefficient.
2. Select the variable that has an LP-value closest to 0.5.
3. Select the fractional variable (if available) that has maximum objective function coefficient.
4. If there are fractional variables that are equal to 1 in the currently best known feasible solution, select the one with maximum cost of them, otherwise, apply strategy 1.
5. Select a fractional variable (if available) that is closest to one, i.e., find a variable $e^\star$ with $\overline{x}_{e^\star} = \max\{\overline{x}_e \mid \overline{x}_e < 1\}$.

6. Select a set L of promising branching variable candidates. Let $Ax \leq b$ be the constraint system of the last solved LP. Solve for each variable $i \in L$ the two Linear Programs

$$v_0^i = \max\{c^T x \mid Ax \leq b,\ x_i = 0\}$$

$$v_1^i = \max\{c^T x \mid Ax \leq b,\ x_i = 1\}$$

and select the branching variable b with

$$\max\{v_0^b, v_1^b\} = \min_{i \in L} \max\{v_0^i, v_1^i\}.$$

Some running time can be saved if instead of the solution of the Linear Programs to optimality only a restricted number of iterations of the simplex-method is performed. Then the objective function value might already indicate the "quality" of the branching variable, especially if a steepest edge pivot selection criterion is applied.

Computational experiments for the strategies (1) and (3) – (5) applied to the TSP can be found in [43]. Other branching variable selection strategies can be found in [5].

Instead of partitioning the set of feasible solutions by branching on a variable, it is also possible to use a hyperplane intersecting the polytope defined by the current subproblem. This alternative way of branching was proposed for the first time in [18] and used with a problem specific hyperplane for the TSP.

Another modification of the branching process is branching on $k \geq 2$ variables or hyperplanes. In this case we get a 2^k-nary instead of a binary branch-and-cut tree.

Select. If the list of active branch-and-cut nodes is empty, the best known feasible solution is the optimum solution. Otherwise a branch-and-cut node is selected and removed from the set of active branch-and-cut nodes and the processed. After a selection the set variables (including the branching variables) must be adjusted. If it turns out that some variable must be set to 0 or 1, yet has been fixed to the opposite value in the meantime, we have a contradiction. In this case the branch-and-cut node is fathomed. If the local lower bound `llb` of the selected node exceeds the global upper bound `gub`, the node is fathomed immediately and the selection process is continued.

Up to now we have not specified which node is selected from the set of active branch-and-cut nodes. There are three well-known enumeration strategies in branch-and-bound (branch-and-cut) algorithms: depth-first search, breadth-first search and best-first search. We define the *level* of a branch-and-cut node B as the number of edges on the path from the root of the branch-and-cut tree to B. In the case of depth-first search a branch-and-cut node with maximum level in the branch-and-cut tree is selected from the set of active nodes in SELECT, whereas in breadth-first search a subproblem with minimum level is selected. In best-first search the "most promising" node becomes the current branch-and-cut

node. For a minimization problem the node with maximum local lower bound among all active nodes is often considered as most promising.

Computational experiments for the TSP (see [43]) show that depth-first search is an enumeration strategy with the "risk" of spending a lot of time in a branch of the tree, which is useless for computing better upper and lower bounds. Often the local lower bound of the current subproblem exceeds the objective function value of an optimum solution, however, this node cannot be fathomed, because no good upper bound is known. The same phenomenon occurs also sometimes when using breadth-first search, but it is very rare if the enumeration is performed in best-first search order.

Fathom. If for a node the global upper bound `gub` does not exceed the local lower bound `llb`, or a contradiction occurred, or an infeasible branch-and-cut node has been generated, or if the LP-solution is a characteristic vector of a feasible solution, the current branch-and-cut node is deleted from further consideration. Even though a node is fathomed, the global upper bound `gub` may have changed during the last iteration, so that additional variables may be fixed by FIXBYREDCOST and FIXBYLOGIMP. The fathoming of nodes in FATHOM may lead to a new root of the branch-and-cut tree for the remaining active nodes.

Output. The currently best known solution, which is either an optimum solution or satisfies a desired guarantee requirement, is written to an output file.

3.3 Computation of Local Lower Bounds

The computation of local lower bounds consists of all elements of the leftmost dashed bounding box of Fig. 6 except EXPLOIT LP. In EXPLOIT LP the upper bounds are updated, if the solution of the Linear Program is the characteristic vector of a better feasible solution. Also improvement heuristics, using information from the LP-solutions, can be incorporated here as suggested in Section 3.4.

For the computation of lower bounds LP-relaxations are solved iteratively, violated valid inequalities are added, and non-binding constraints are deleted from the constraint matrix.

In this section we will also point out that an additional data structure for inequalities, called *pool*, is very useful, although not necessary for the correctness of the algorithm. For now, we can think of a pool just as a collection of constraints or variables.

The *active inequalities* are the ones in the current LP and are both stored in the pool and in the constraint matrix, whereas the *inactive constraints* are only present in the pool. The pool is initially empty. If an inequality is generated by a separation algorithm, it is stored both in the pool and added to the constraint matrix. Further details of the pool are outlined in Section 5.3.

Initialize New Node. If the current branch-and-cut node is the root node of the branch-and-cut tree the LP is initialized by some small constraint system. Often the upper and lower bounds on the variables are a sufficient choice (e.g., for the maximum cut problem). For the TSP, the degree constraints for all nodes are normally added. Augmenting the initial system by other promising cutting planes can sometimes reduce the overall running time (see [35]). A primal feasible basis derived from a feasible solution can be used as a starting basis in order to avoid phase 1 of the simplex algorithm.

Any set of valid (preferably facet defining) inequalities can be used to initialize the first constraint system of subsequent subproblems. Yet, in order to guarantee monotonicity of the values of the local lower bounds in each branch of the enumeration tree, and to save running time, it is appropriate to initialize the constraint matrix by the constraints that were binding when the last LP of the father in the branch-and-cut tree was solved.

Since the basis of the father is dual feasible for the initial LP of its sons, phase 1 of the simplex method can be avoided by starting with this basis. The columns of non-basic set and fixed variables can be removed from the constraint matrix to keep the LP small. If their value is non zero, the right hand side of the constraint has to be adjusted, and the corresponding coefficients of the objective function must be added to the optimum value returned by the simplex algorithm in order to get the correct value of the variable `lpval`. Set or fixed basic variables should not be deleted, because this would lead to a neither primal nor dual feasible basis and require phase 1 of the simplex method. The adjustment of these variables can be performed by adapting their upper and lower bounds.

Solve LP. The LP is solved by the primal simplex method, if the basis is primal feasible (e.g., if variables have been added) or by the dual simplex method if the basis is dual feasible (e.g., if constraints have been added). The two-phase simplex method is required if the basis is neither primal nor dual feasible. This can happen if constraints necessary for the initialization of the first LP of a subproblem are not available since they had to be removed from the pool as we will describe in Section 5.3.

The LP-solver is one of the bottlenecks of a branch-and-cut algorithm. Sometimes more than 90% of the computation time is spent in this procedure. Today, efficient implementations of the simplex method, like Cplex [17] or XPress [69] are competitive on solving Linear Programs from scratch. However, a branch-and-cut algorithm requires a LP-solver with very efficient post-optimization routines.

The simplex method satisfies all the requirements of a branch-and-cut algorithm and it is used by nearly all implementations of cutting plane algorithms. Therefore we have outlined the algorithm in this section under the assumption that the simplex method is used. Since the LPs that have to be solved in a cutting plane algorithm, are often highly degenerate, good pivot variable selection strategies, like the steepest-edge pivot variable selection criterion, are necessary. These degeneracies might even require some preprocessing of the LPs (see, e.g., [35]).

Exploit LP. First, we have to check if the current LP-solution is the characteristic vector of a feasible solution. If this is the case we leave the bounding part and fathom the current branch-and-cut node. Otherwise, most implementations of branch-and-cut algorithms proceed with the cutting plane generation phase. However sometimes we can do better by exploiting the fractional LP-solutions to improve the upper bound before additional cutting planes are generated. We will discuss these ideas in Section 3.4. Before the separation phase is performed in SEPARATE, variables may be fixed or set as explained in FIX AND SET.

Tailing Off. Often it is reasonable to abort the cutting plane part if no significant increase of `lpval` in the most recent LP-solutions has taken place. This phenomenon is called *tailing-off* (cf. [56]). If during the last k (e.g., $k = 10$) iterations in the bounding part, `lpval` did not increase by more than p % (e.g., $p = 0.01$), new subproblems are created instead of generating further cutting planes. Good choices for the parameters p and k are both rather problem specific and dependent on the quality of the available cutting plane generation procedures.

Separate. The separation phase is the central part of a branch-and-cut algorithm. We try to find violated globally valid (preferably facet-defining) constraints, which are added to the LP. We say an inequality is globally valid, if it is a valid inequality for every subproblem of the branch-and-cut algorithm. We call a constraint locally valid, if it is only a valid inequality of a subproblem S and all subproblems of the subtree rooted at S.

It may not always be a good strategy to call any available separation algorithm in each iteration of the cutting plane generation. Experiments show that a hierarchy of separation routines is often preferable. Certain separation methods should only be performed, if others have failed. Before calling a time consuming exact separation algorithm, one should attempt to generate cutting planes with faster heuristic methods. However, this hierarchy is rather problem specific so that we cannot give a general recipe for its application. We refer to the publications on specific implementations.

The constraint pool provides us with another cutting plane generation technique. Inactive constraints that are violated by the current LP-solution can be regenerated from the pool. Of course this method requires an efficient algorithm to perform this test and to transform the storage format of the constraint used in the pool into the storage format for the LP-solver. The pool-separation can be advantageous for classes of inequalities for which only heuristic separation routines are available. In this case it can happen that a constraint of this class is violated, yet cannot be identified by the heuristic. However, this cutting plane might have been generated earlier in the computational process (at a different LP solution which has been more "convenient" to our heuristic). If this constraint is still contained in the pool, it can be reactivated now.

It can also happen that the pool-separation for a class of constraints is more efficient than a direct separation by a time consuming heuristic or an exact algo-

rithm. Therefore, the pool-separation should always be performed before calling these algorithms. However, for other classes of constraints it can sometimes be observed that the pool-separation is very slow in comparison to direct separation methods.

Since the pool can become very large during the computational process, it is necessary to limit the search in the pool for violated inequalities. For instance, the pool-separation can be restricted to some classes of constraints. Therefore the pool-separation should be carefully included into the hierarchy of separation algorithms and it requires many computational tests to find a strategy that is efficient for a specific combinatorial optimization problem.

For some combinatorial problems like the maximum cut problem, often several hundred violated inequalities can be generated. However, it would be sufficient to add those constraints to the LP that will be binding after the solution of the next LP. Unfortunately we do not know this subset of the generated inequalities. On the other hand, adding all the constraints to the matrix of the LP-solver can slow down the overall computation time. Therefore, depending on the performance of the LP-solver, only a limited number of constraints should be added to the LP. A straightforward approach is just stopping the cut generation when this limit is reached. A more sophisticated method might be generating as many constraints as possible, and afterwards selecting the "best" of them. A simple classification criterion is the degree of violation given by the value of the corresponding slack. For the TSP, Padberg and Rinaldi [56] propose as a measure the distance of the LP-solution from the projection of the cut into the affine space defined by the degree equations. The larger this distance the better the cut, yet, this method is computationally expensive. Other quality measures for cutting planes like the angle of the cut defining hyperplane to the objective function vector have been investigated in [15].

The representation of the inequality for the LP-solver can have significant influence on its running time. For instance, equations of the integer programming formulation can be added to any valid inequality without changing the half-space which it defines. However, the number of the non-zero coefficients in the inequality may differ. Normally, LP-solvers are more efficient if the number of non-zeros in the constraint matrix is small.

The solution of the separation problem is very problem specific. Therefore we only want to present an example for the TSP.

A polynomial time algorithm for the solution of the exact separation problem of subtour elimination constraints can be directly derived from their definition in Section 2: If the value of the minimum weight cut in the support graph (the graph with the LP-solution as edge weights) is greater than or equal to 2, the current LP-solution does not violate any subtour elimination constraint. Each cut with a value less than 2 induces a violated subtour elimination constraint.

So the separation problem for subtour elimination constraints reduces to a minimum capacity cut problem for which the practically most efficient solution was given in Padberg and Rinaldi [57] and refined in Jünger, Rinaldi, and Thienel [42].

Eliminate. If inequalities are added to the constraint matrix of the LP-solver, and no inequalities are eliminated, soon the size of the matrix might become too large to solve the linear programs in reasonable time and even the storage of the constraints in the matrix would require too much memory. Moreover, there are inequalities that become redundant for the rest of the computation. Therefore a strategy is required to maintain a reasonable sized matrix, yet not to eliminate important inequalities.

It is an obvious and simple strategy for the elimination of constraints to delete all active inequalities that are non-binding in the last LP-solution from the constraint structure before the LP is solved after a successful cutting plane generation phase. To avoid cycling, i.e., a constraint is eliminated, but already violated after the next LP-solution, either constraints should be only removed if the value of the slack s is big enough (e.g., $s > 0.001$), or if they are non-binding during several successive LP-solutions.

3.4 Computation of Global Upper Bounds

For most combinatorial optimization problems a host of heuristics is available to compute feasible solutions that provide global upper bounds for the branch-and-cut algorithm. Traditionally the computation of a global upper bound is performed in the procedure INITIALIZE before the cutting plane generation and enumeration phase starts. Later better lower bounds are only found if the LP-solution is the characteristic vector of a feasible solution. However, it can be observed that this happens rather seldomly. Therefore sophisticated heuristics must be applied in INITIALIZE to generate a good lower bound. Otherwise, the enumeration tree may grow too large.

In [43] a dynamic strategy, integrated in the cutting plane generation part, for the computation of lower bounds is presented, which we briefly outline. It turns out that the fractional LP-solutions occurring in the lower bound computations in a branch-and-cut algorithm give hints on the structure of optimum or near optimum feasible solutions.

The basic requirement for the upper bound computations is efficiency in order not to inhibit the optimization process. While in the first stages high emphasis is laid on providing good feasible solutions, this emphasis is less in the later stages of the computational process. On the other hand, computing upper bounds can always be reasonable since new knowledge about the structure of optimum feasible solutions is acquired (e.g., because of fixed and set variables).

Exploiting the LP-Solution. Integer optimum solutions, i.e., characteristic vectors of feasible solutions, will almost never result from the LPs occurring in the branch-and-cut algorithm. But, it can be observed that these solutions, although having many fractional components, give information on good feasible solutions. They have a certain number of variables equal to 1 or 0 and also a certain number of variables whose values are close to 1 or 0. This effect can be exploited to form a starting feasible solution for subsequent improvement heuristics.

We show how the information of the LP-values of the variables can be used for the construction of a feasible solution for the TSP. We use the terms edge and variable of the integer programming formulation interchangeably, since they are in a one-to-one correspondence in our examples. First, we check if the current LP-solution is the characteristic vector of a tour. If this is the case we terminate the procedure EXPLOIT LP. Otherwise, edges are sorted according to their values in the current LP-solution. We give decreasing priorities to edges as follows:

- edges that are fixed or set to 1,
- edges equal to 1 or close to 1 in the current LP,
- edges occurring in several successive LPs.

This list is scanned and edges become part of the tour if they do not produce a subtour with the edges selected so far. This gives a system of paths and isolated nodes which now have to be connected. To this end a savings heuristic of Clarke and Wright [14], originally developed for vehicle routing problems, can be used, since the TSP can be considered as a special vehicle routing problem involving only one vehicle. The previous step gives us a feasible solution that can be improved by local improvement heuristics like Lin-Kernighan [48].

This heuristic basically consists of successively merging partial tours to obtain a Hamiltonian tour. We select one node as base node and form partial tours by connecting this base node to the end nodes of each of the paths obtained in the selection step and also adding a pair of edges to nodes not contained in any path. Then, as long as more than one subtour is left, we compute for every pair of subtours the savings that is achieved if the subtours are merged by deleting in each subtour an edge to the base node and connecting the two open ends. The two subtours giving the largest savings are merged. Edges that are fixed or set to 0 should be avoided for connecting paths.

3.5 Sparse Solution and Column Generation

Often combinatorial optimization problems involve a very large number of variables, yet a feasible solution is comparatively sparse. For instance, the TSP on a complete graph of n nodes has $\binom{n}{2}$ variables. Yet, a tour consists only of n edges. Hence, the computational process can be accelerated, if a suitable subset of the edges is initially selected and appropriately augmented during the solution of the problem, if this is required for the correctness of the algorithm. However, sparse graph techniques can not be applied to problems with a dense solution structure like the maximum cut problem (see Section 4.1). Sparse graph techniques for the TSP have been introduced by Grötschel and Holland [36].

We present techniques exploiting the sparsity of solutions only for combinatorial optimization problems defined on graphs. However this technique can be generalized for other problems, if the structure of the solutions is sparse, suitable subsets of the variables can be computed efficiently, and a method to generate the columns of non-active variables is available.

In order to integrate this technique into the basic algorithm in Fig. 6 described so far, we have to deal with LP problems where not only rows but also

columns of the constraint matrix are dynamically created. The resulting more general algorithm that in [56] was called *branch-and-cut* is described in the flowchart shown in Fig. 9. With respect to the basic algorithm, the gray boxes in the flowchart have to be added or changed. A subproblem, in which an infeasible LP is detected, cannot be fathomed at once, but rather it must be checked if the addition of non-active variables can regenerate the feasibility. We explain this process in ADDVARIABLES. Before leaving the bounding part, it has to be verified in PRICE OUT, if the LP-solution computed on the sparse graph is also optimum on the complete graph. Only in this case the variable `lpval` becomes a local lower bound `llb`. The application of the routine FIX AND SET has to be performed now more carefully. The procedure SETBYREDCOST can only be applied after an additional pricing step, in which no variable has to be added. This is also the case for FIXBYREDCOST if the root node of the remaining branch-and-cut tree is currently processed.

Suitable Sparse Graphs. The initial sparse graph is generated in the procedure INITIALIZE. For the TSP, a good choice for a sparse graph is the k-nearest neighbor graph. Another suitable subset of the edges may be the Delaunay graph (see also [60] and [18]). Figures 10, 11 and 12 show an optimum tour through 127 beer gardens of Augsburg (Germany) together with the 5-nearest neighbor graph and the Delaunay triangulation. If it cannot be guaranteed that the sparse graph contains a feasible solution, it should be augmented by the edges of a solution computed by a heuristic. Padberg and Rinaldi [56] suggest to create a series of feasible solutions heuristically and initialize the sparse graph with all involved edges.

In addition to the sparse graph, the edges of the "reserve graph" can be computed. These edges are additional "promising" edges that do not belong to the sparse graph. For instance, if the sparse graph is the 5-nearest neighbor graph, a suitable reserve graph is given by the edges that have to be added to get the 10-nearest neighbor graph. The reserve graph can be used in PRICE OUT and ADDVARIABLES.

The algorithm starts working on G, adding and deleting edges (variables) dynamically during the optimization process. We refer to the edges in G as *active* edges and to the other edges as *non-active* edges.

Add Variables. Variables have to be added to the sparse graph if indicated by the reduced costs (handled by PRICE OUT) or if the current LP is infeasible. The latter may be caused by two reasons.

First, some active inequality has a void left hand side, since all involved variables are fixed or set and removed from the LP, but is violated. If all coefficients of non-active variables in this inequality are nonnegative, it is clear from our strategy for variable fixings and settings that the branch-and-cut node is fathomed (all constraints are assumed to be of the form $a^T x \leq b_i$). However, if there is a non-active variable with a negative coefficient, this variable may remove the violation. So it is added to the LP.

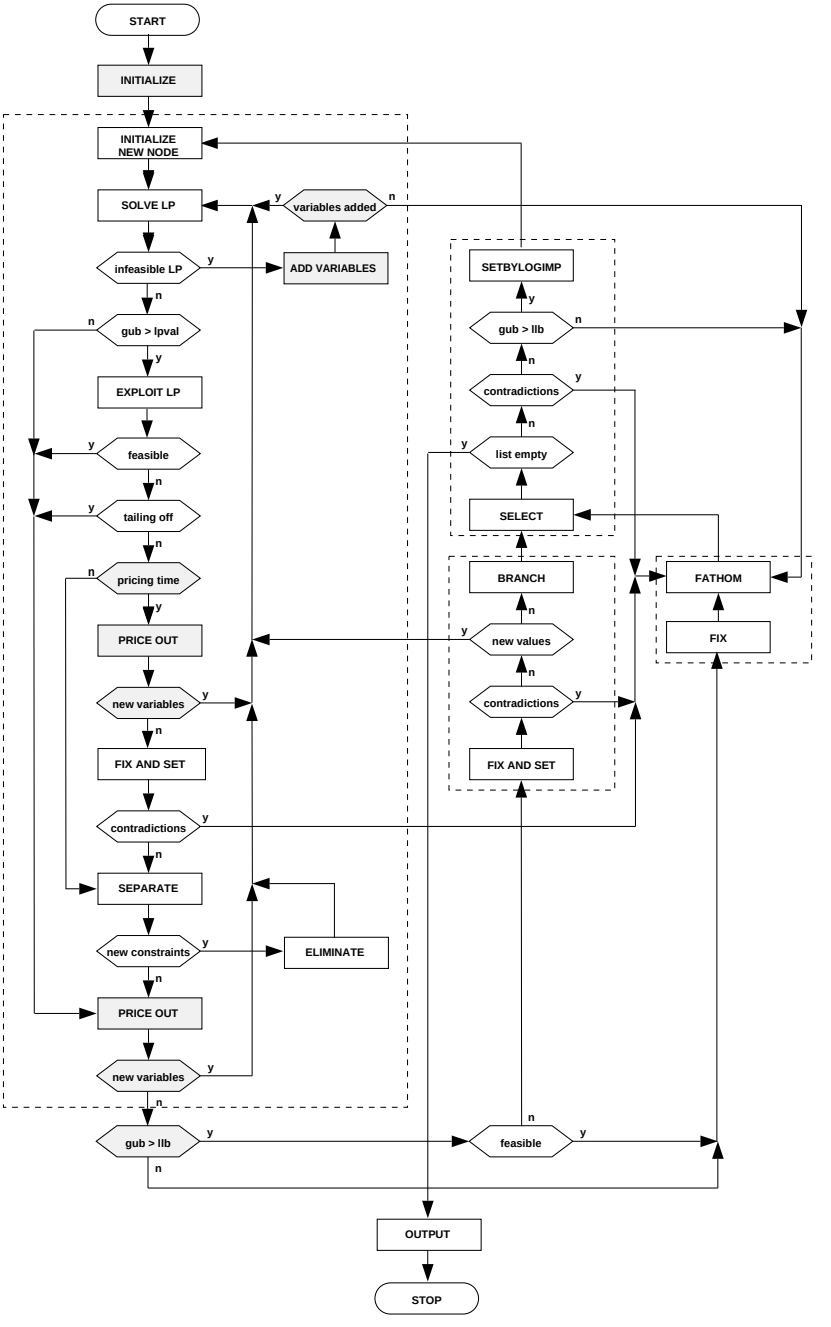


Fig. 9. Flowchart of a branch-and-cut algorithm.

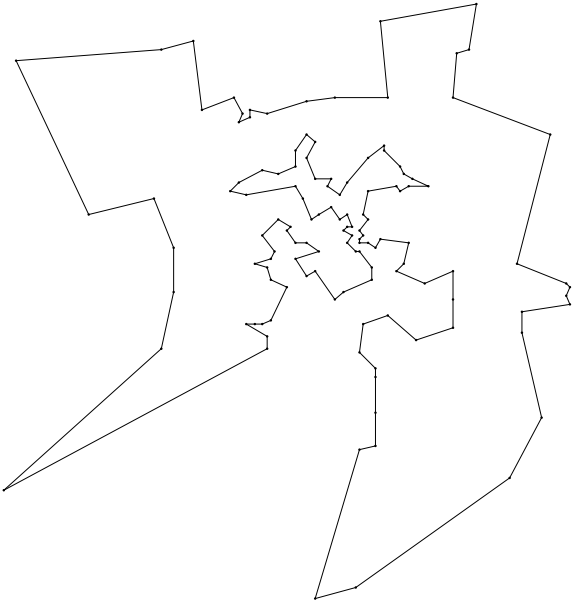


Fig. 10. Shortest tour through 127 beer gardens in Augsburg (Germany).

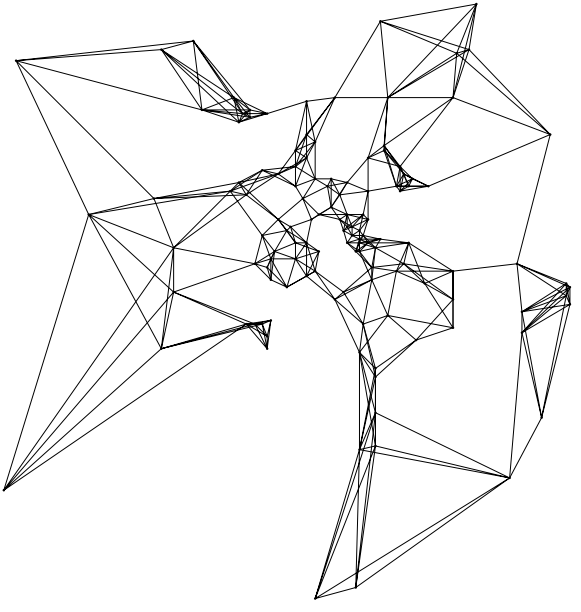


Fig. 11. The 5-nearest neighbor graph for the beer gardens in Augsburg contains all but 3 edges of the best tour.

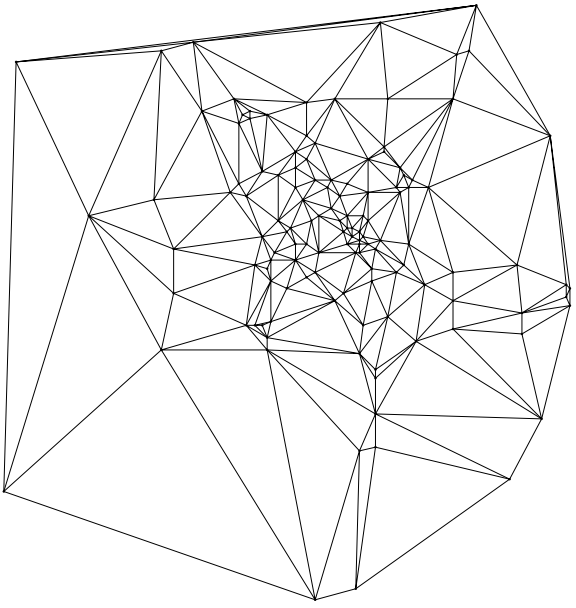


Fig. 12. In the Delaunay graph for the beer gardens in Augsburg only 2 edges of the best tour are missing.

Second, the above condition does not apply, and the infeasibility is detected by the LP-solver. In this case a pricing step is performed in order to find out if the dual feasible LP-solution is dual feasible for the entire problem. Variables that are not in the current sparse graph (i.e., are assumed to be at their lower bound 0) and have negative reduced cost are added to the current sparse graph. An efficient way of computing the reduced costs is outlined in PRICE OUT.

If new variables have been added, then the new LP is solved. Otherwise, by a more elaborated method we try to add new variables to the LP in order to make it feasible. The LP-value `lpval`, which is the objective function value corresponding to the dual feasible basis where primal infeasibility is detected, is a lower bound for the objective function value obtainable in the current branch-and-cut node. So if $\texttt{lpval} \geq \texttt{gub}$, the branch-and-cut node can be fathomed.

Otherwise, we first mark all infeasible variables, i.e., all those that violate the lower or the upper bound and all the negative slack variables.

Let e be a non-active variable and r_e be the reduced cost of e. An edge e is taken as a candidate only if $\texttt{lpval} + r_e \leq \texttt{gub}$. Let B be the basis matrix corresponding to the dual feasible LP-solution, at which the primal infeasibility was detected. For each candidate e let a_e be the column of the constraint matrix corresponding to e and solve the system $B\overline{a}_e = a_e$. Let $\overline{a}_e(b)$ be the component of $\overline{a}_e$ corresponding to basic variable x_b. Increasing x_e "reduces some infeasibility" if one of the following holds:

- x_b is an infeasible structural variable (i.e., corresponding to an edge of G) and
$$(x_b < 0 \text{ and } \overline{a}_e(b) < 0) \qquad \text{or} \qquad (x_b > 1 \text{ and } \overline{a}_e(b) > 0)$$
- x_b is a negative slack variable and
$$\overline{a}_e(b) < 0.$$

In such a case the variable e is added to the set of active variables and the marks are removed from all infeasible variables whose infeasibility can be reduced by increasing x_e. This can be done in the same hierarchical fashion as described below in PRICE OUT.

If variables can be added, the new LP is solved, otherwise the branch-and-cut node is fathomed. Note that all systems of linear equations that have to be solved have the same matrix B, and only the right hand side a_e changes. This can be utilized by computing a factorization of B only once, in fact, the factorization can be obtained from the LP-solver for free. For further details on this algorithm, see [56].

Price Out. Pricing is necessary before a branch-and-cut node can be fathomed. Its purpose is to check if the LP-solution computed on the sparse graph is valid for the complete graph, i.e., all non-active variables "price out" correctly. If this is not the case, non-active variables with negative reduced cost are added to the sparse graph and the new LP is solved using the primal simplex method starting with the previous (now primal feasible) basis, otherwise the local lower bound `llb` and possibly the global lower bound `glb` can be updated.

Although the correctness of the algorithm does not require this, additional pricing steps can be performed every k (e.g., $k = 10$) solved LPs (see [56]). The effect is that non-active variables which are required in a good or optimum feasible solution tend to be added to the sparse graph early in the computational process. If no variables are added, it can also be tried to fix or set variables by reduced cost criteria.

Let y be the vector of the dual variables, and A_e the column of an inactive variable e in the matrix A defined by the active constraints, and c_e the corresponding objective function coefficient, then the reduced costs of the variable e are given by $r_e = c_e - y^T A_e$.

The computation of the reduced cost for all inactive edges takes a great computational effort, but it can be performed significantly faster by an idea of Padberg and Rinaldi [56]. If our current branch-and-cut node is the root of the remaining branch-and-cut tree, it can be checked if the reduced cost r_e of a non-active variable e satisfies the relation $\texttt{lpval} + r_e > \texttt{gub}$. In this case this non-active edge can be discarded forever. During the systematic enumeration of all edges of the complete graph, an explicit list of those edges which remain possible candidates can be made. In the early steps of the computation, too many such edges remain, so that this list cannot be completely stored with reasonable memory consumption. Instead, a partial list is stored in a fixed size buffer and

the point where the systematic enumeration has to be resumed after considering the edges in the buffer is memorized. In later steps of the computation there is a good chance that the complete list fits into the buffer, so that later calls of the pricing routine become much faster than early ones.

In [43] a further modification of the procedure PRICE OUT is presented. It can be observed that the reduced costs of edges not belonging to the reserve graph are seldomly positive, if the reserve graph is appropriately chosen. Hence, it turns out that a hierarchical approach is advantageous. Only if the "partial pricing" considering the edges of the reserve graph has not added variables, the reduced cost of all non-active variables have to be checked.

Computation of Global Upper Bounds. Sparse graph techniques can also be used for the computation of feasible solutions both if heuristics are applied only during the initialization phase or if they are integrated in the cutting plane generation phase.

A candidate subgraph is a subgraph of the complete graph on n nodes containing reasonable edges in the sense that they are "likely" to be contained in a good feasible solution. These edges are taken with priority in the various heuristics, thus avoiding the consideration of the majority of edges, which are assumed to be of no importance. Various candidate subgraphs and the question of how to compute them efficiently are discussed in [60].

The candidate subgraph can be related to the set of active variables in the linear programming problems, if the heuristics are integrated into the cutting plane generation part as described before. Basically, the candidate subgraph is initialized with some graph (e.g., the empty graph) and then edges are added whose corresponding values are close to one. In order to avoid too extensive growing of the candidate subgraph and to avoid being biased by LPs that were not recently solved, the candidate subgraph should be cleared in certain intervals (e.g., every 20th cutting plane phase) and reinitialized.

It should be noted that the feasible solution found by the heuristics should not be restricted to only using edges of the sparse graph. These edges are only considered with priority and lead to an acceptable CPU time. Usually, heuristics will introduce edges that are not active in the LP. These edges are added to the set of active variables. This is based on the assumption that these edges are also important for the upper bound computations and would be added to the LP in some pricing step anyway. This way the set of active variables is augmented without pricing.

4 Some More Advanced Topics

The success of a branch-and-cut algorithm relies, to a great extent, on a careful design of all its components. This is true, in particular, when one wants to treat instances of very large size. Since in this chapter we are dealing with combinatorial optimization problems, we show here how the structure of these problems can sometimes be used to handle instances of large size by branch-and-cut. Due to

space limitations we only address two issues. In the first we describe a reduction technique that is used in the separation problem for TSP and the maximum cut problem. In the second we deal with problems where the LP relaxation used for the branch-and-cut algorithm has exponentially many inequalities and variables.

Before we start, let us compile a few notions in polyhedral combinatorics. A polytope $P \subseteq \mathbb{R}^n$ can either be characterized as the convex hull of a finite set $I \subseteq \mathbb{R}^n$, i.e.,

$$P = \text{conv}(I),$$

or as the solution set of a finite system of inequalities $Ax \leq a$ and equations $Bx = b$, i.e.,

$$P = \{x \in \mathbb{R}^n \mid Ax \leq a, Bx = b\}.$$

As we have seen, the first characterization, also called *inner description*, arises naturally when the points in I are the characteristic vectors of certain combinatorial objects over which we want to optimize a linear objective function. The second characterization is also called an *outer description*. Let us assume that $Bx = b$ is a minimal equation system, i.e., the matrix B has full row rank m and there are no hidden equations in the system $Ax \leq a$. Then the polytope P has dimension $\dim(P) = n - m$. (For a full dimensional polytope the equation system $Bx = b$ is void.)

The inequality $a^T x \leq a_0$ is *valid* for P if $P \subseteq \{x \in \mathbb{R}^n \mid a^T x \leq a_0\}$. For any valid inequality $a^T x \leq a_0$ for P, the polytope $\{x \in P \mid a^T x = a_0\}$ is called a *face* of P. Faces of P different from P itself are called *proper faces* of P. A proper face of minimum dimension 0 is called a *vertex* of P. If P arises as the convex hull of the characteristic vectors of certain combinatorial objects, the vertices of P are exactly the integral (0/1 valued) characteristic vectors. A polytope in which all vertices have only integral components is called an *integral polytope*. A proper face of maximum dimension $\dim(P) - 1$ is called a *facet* of P. The most compact outer description of P consists of a minimal equation system $Bx = b$ and an inequality system $Ax \leq a$ that consists only of facet defining inequalities, one for each facet of P.

A special case of a celebrated result of Grötschel, Lovász, and Schrijver [37] says that we can optimize a linear objective function over a polytope P in polynomial time if and only if we can solve the separation problem for P in polynomial time.

4.1 Reduction and Lifting for Separation

In the previous section, we have used the TSP as an illustrating example. We have utilized the fact that the degree equations and the subtour elimination constraints constitute an integer linear programming formulation of the TSP. We have also discussed the use of sparse graph techniques that are crucial for a successful branch-and-cut approach to the TSP. The efficient solution of the separation problem for the subtour elimination constraints via a minimum capacity cut calculation has also been mentioned. However, the striking success of the branch-and-cut approach to the exact solution of large TSP instances relies on the much

more detailed knowledge of the facial structure of the TSP polytope P^n_{TSP} and practically efficient heuristics for the separation of inequalities other than the subtour elimination constraints. Over the years, many classes of facet defining, or simply valid, inequalities for P^n_{TSP} have been identified, see Naddef [52] for a recent survey. The state of the art in TSP solving largely depends on how much of this knowledge can be exploited by efficient separation algorithms. In contrast to a very successful new "projection cut paradigm" for TSP-separation [⟶ Applegate/Bixby/Chvátal/Cook], the traditional approach of attempting to separate a priori known inequalities is called the "template paradigm". We have not the space to describe various "template" classes of (facet defining) valid inequalities for the TSP, but we would like to point out how reduction and lifting helps in separation, regardless of which paradigm is used. We sketch the situation for the TSP (and refer to [⟶ Applegate/Bixby/Chvátal/Cook] for details on the "project and lift" paradigm) and then discuss the case of the maximum cut problem in more detail.

Let P^n_{SUBTOUR} be the solution set of (2) without the integrality constraints, P^n_{SUBTOUR} is called the *subtour polytope* and clearly

$$P^n_{\text{SUBTOUR}} \supset P^n_{\text{TSP}} = \text{conv}\left(P^n_{\text{SUBTOUR}} \cap \{0,1\}^{E_n}\right) \quad .$$

By the polynomial equivalence of separation and optimization [37] we can optimize over P^n_{SUBTOUR} in polynomial time, and a pure cutting plane algorithm can provide an $\hat{x} \in P^n_{\text{SUBTOUR}}$ efficiently. With such a point we associate its *support graph*, which is a weighted graph with n nodes whose edges correspond to nonzero components of $\hat{x}$; each edge has a weight given by the value of the associated component of $\hat{x}$. In a branch-and-cut algorithm for the TSP it is therefore reasonable to make sure that the point $\hat{x}$ to be separated by further inequalities is contained in the subtour polytope, i.e., for any $\emptyset \neq W \subsetneqq V$ we have that $\hat{x}(\delta(W)) := \sum_{e \in \delta(W)} x_e \geq 2$. While, as mentioned in Section 3.3, the exact separation of the subtour elimination inequalities can be solved very efficiently even for points $\hat{x}$ whose support graph has several thousand nodes and edges, for these graphs it would be next to impossible to solve the separation problem for other classes of inequalities in a reasonable amount of time.

A set $S \subset V$ of at least two nodes is called *tight* if $\hat{x}(\delta(S)) = 2$. Typically the support graph of $\hat{x}$ has several tight sets. If we *contract* a tight set, i.e., if we identify all its nodes into a single node, remove the loops generated by this process, and replace the resulting parallel edges by a single edge with weight equal to the sum of the weights of the removed parallel edges, we obtain a smaller graph. It is not difficult to see that this graph is the support graph of a point $\hat{x}'$ belonging to $P^{n'}_{\text{SUBTOUR}}$, where $n' < n$.

These observations suggest the following procedure:

1. Contract some selected tight sets and produce a new point $\hat{x}' \in P^{n'}_{\text{SUBTOUR}}$.
2. Find inequalities valid for $P^{n'}_{\text{TSP}}$ violated by $\hat{x}'$.
3. Extend such inequalities to inequalities valid for P^n_{TSP} and violated by the original point $\hat{x}$.

Unfortunately, it may happen that after contracting some tight sets, the new point $\hat{x}'$ belongs to $P^{n'}_{\mathrm{TSP}}$, i.e., it can be expressed as a convex combination of the characteristic vectors of a set of Hamiltonian cycles. Therefore, no valid inequality for $P^{n'}_{\mathrm{TSP}}$ can be violated by $\hat{x}'$ in this case. In [58] Padberg and Rinaldi show how this happens and give sufficient conditions for a tight set to be contracted without producing such undesired situations.

Step 3 of the above procedure can be effectively solved using a lifting theorem of Naddef and Rinaldi [49] that says that, under conditions that are satisfied by all facet defining inequalities for the TSP polytope known to date, a facet defining inequality for $P^{n'}_{\mathrm{TSP}}$ violated by $\hat{x}'$ by an amount v can be easily lifted to a facet defining inequality for P^{n}_{TSP} violated by $\hat{x}$ by the same amount v.

Successful approaches to TSP separation differ mainly in how the separation for $P^{n'}_{\mathrm{TSP}}$ is performed, e.g., Padberg/Rinaldi [56], Naddef/Thienel [50,51], and Christof/Reinelt [15] make exclusive use of the template paradigm (the latter makes special use of a library of all facets for P^{n}_{TSP} $(n \leq 10)$), [$\longrightarrow$ Applegate/Bixby/Chvátal/Cook] go beyond. Naddef [52] is the latest survey of all state of the art techniques in exact TSP solving by branch-and-cut.

In Section 1, we have transformed the break minimization problem to the maximum cut problem in a special sparse graph $G = (V, E)$ with edge weights. Exact solutions to maximum cut problems in sparse graphs play an important role in statistical physics, since the determination of a minimum energy state of a spin glass reduces to a maximum cut problem in a sparse graph, see, e.g., De Simone et al. [25] for details. With the branch-and-cut algorithm described there, maximum cut instances on toroidal grids of size up to 100×100 (see Fig. 13) could be routinely solved to optimality.

Analogously to (2), we can give an integer linear programming formulation for the maximum cut problem:

$$\begin{array}{lll} \text{maximize} & \displaystyle\sum_{e\in E} c_e x_e & \\ \text{subject to} & \displaystyle\sum_{e\in F} x_e - \sum_{e\in C\setminus F} x_e \leq |F| - 1 & \text{for all } F \subseteq C,\ |F| \text{ odd,} \\ & & \text{for each cycle } C \text{ of } G \\ & 0 \leq x_e \leq 1 & \text{for all } e \in E \\ & x_e \in \{0,1\} & \text{for all } e \in E \end{array} \tag{5}$$

The nontrivial inequalities are called *cycle inequalities*. Therefore, we call the polytope P^G_{CYCLE} defined by the cycle inequalities and the trivial inequalities the *cycle polytope*. As usual, the *cut polytope* P^G_{CUT} is the convex hull of the characteristic vectors of all cuts and we have

$$P^G_{\mathrm{CYCLE}} \supset P^G_{\mathrm{CUT}} = \mathrm{conv}\left(P^G_{\mathrm{CYCLE}} \cap \{0,1\}^E\right)$$

A cycle inequality defines a facet of P^G_{CUT} when the defining cycle C is chordless. A nontrivial inequality $x_e \geq 0$ or $x_e \leq 1$ defines a facet of P^G_{CUT} when e is not contained in a triangle of G. All non-facet defining inequalities can be eliminated from the linear description of P^G_{CYCLE} and P^G_{CUT}. Therefore, when G

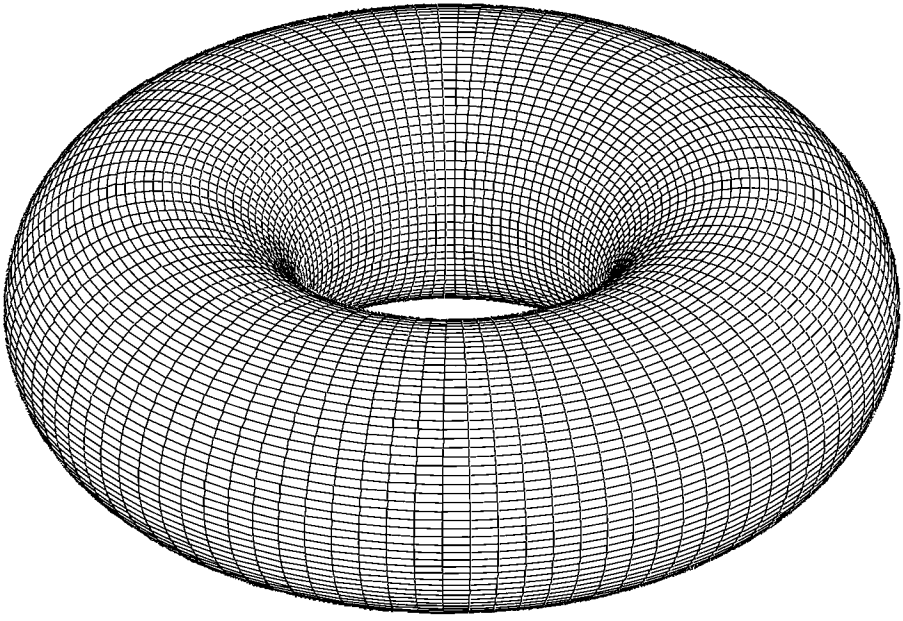


Fig. 13. 100×100 grid on a torus.

is the complete graph K_p, only the inequalities

$$\begin{aligned} x_{ij} + x_{ik} + x_{jk} &\leq 2 \\ x_{ij} - x_{ik} - x_{jk} &\leq 0 \\ -x_{ij} + x_{ik} - x_{jk} &\leq 0 \\ -x_{ij} - x_{ik} + x_{jk} &\leq 0 \end{aligned}$$

remain.

The polytope P^G_{CUT} has been extensively studied and a number of families of valid inequalities, several of them facet defining, have been described (see, e.g., the surveys [59] and [23]). For some of these inequalities separation procedures have been proposed (see, e.g., [23], [11], and [24]).

All these results concern the maximum cut problem on complete graphs. But how can they be used when the graph is not complete? A trivial way to exploit the current knowledge about P^G_{CUT} for the case of an arbitrary graph G is to add the missing edges to G in order to obtain an artificial complete graph, and to assign a zero weight to them. Such a technique has been successfully used for other combinatorial problems, where the sparsity of the original graph can actually be exploited to handle the artificial complete graph efficiently. This is the case, for example, in the TSP, where even if the original graph is not sparse, all the computations are carried out on a very sparse subgraph. To do so, it is

assumed that the edges of a suitable large subset have no intersection with an optimum solution and thus their corresponding variables are permanently set to zero. A proof that this assumption is correct is eventually provided by the solution algorithm. On the contrary, for the maximum cut problem there is no obvious way to exploit the sparsity of the original problem or to use a small working subset of the original edges. This means that if one uses the above technique of completing the graph with edges of zero weight, the exact solution of the maximum cut problem on sparse graphs has the same computational difficulties as on complete graphs.

Unfortunately, interesting applications of the maximum cut problem, like the study of minimum energy configurations in spin glasses, require the exact solution of instances with several thousand nodes. Therefore, the solution of these instances is out of reach, unless the problem is solved in the original sparse graph.

Why is there such a difference between the TSP and the maximum cut problem, as far as the exploitation of the descriptions of their polyhedra is concerned? We try to make this point more clear by making some simple observations.

Let $\mathcal{H}$ and $\mathcal{H}'$ be the sets of all the Hamiltonian cycles of the graphs G and $G \setminus e$, respectively, where $G \setminus e$ is obtained from G by removing edge e. Let P and P' be the convex hulls of the characteristic vectors of all the elements of $\mathcal{H}$ and $\mathcal{H}'$, respectively. Clearly, $\mathcal{H}'$ is a subset of $\mathcal{H}$ and is made of all the elements of $\mathcal{H}$ that do not contain edge e. Correspondingly, P' is the intersection of P with the hyperplane $\{x \mid x_e = 0\}$, thus P' is a face of P. As a consequence, any valid inequality for P is also valid for P' and, by combining it with the equation $x_e = 0$, valid for P', it can be turned into an equivalent inequality obtained by the original one by dropping the term in x_e. Thus, a linear description of P' is readily obtained from a linear description of P.

Let now $\mathcal{K}$ and $\mathcal{K}'$ be the sets of all cuts of G and $G \setminus e$, respectively, and $P^G_{\rm CUT}$ and $P^{G\setminus e}_{\rm CUT}$ the corresponding polytopes. All the elements of $\mathcal{K}$ that do not contain edge e are in $\mathcal{K}'$. Moreover, if $K \in \mathcal{K}$ and $e \in K$ then $K \setminus \{e\} \in \mathcal{K}'$. Therefore, the characteristic vectors of elements of $\mathcal{K}'$ are projections of characteristic vectors of elements of $\mathcal{K}'$ onto the subspace $\{x \mid x_e = 0\}$. Consequently, $P^{G\setminus e}_{\rm CUT}$ is the projection of $P^G_{\rm CUT}$ onto that subspace. If we have a system of linear inequalities for $P^G_{\rm CUT}$, or for one of its relaxations that can be described in a compact combinatorial way, we would like to have a similar compact description also for the linear system of its projection. However, the linear system of the projection, that can be obtained via a Fourier-Motzkin procedure [$\longrightarrow$ Balas], can get extremely complex and it is very unlikely that a general criterion can be devised to describe it in a compact way. The only lucky (and non-trivial) case we are aware of is when the relaxation of $P^G_{\rm CUT}$ is the cycle polytope of the graph G described by the system (5). Its projection onto $\{x \mid x_e = 0\}$, as proved by Barahona [8], is again the cycle polytope of graph $G \setminus e$.

After these observations it seems that the only reasonable way to proceed in order to have combinatorial descriptions of linear systems of relaxations of $P^G_{\rm CUT}$ is to consider graphs with a special structure. Unfortunately, after the publication

of the paper by Barahona and Mahjoub [10], where the study of a description of P^G_{CUT} by linear inequalities was initiated and where the inequalities (5) were introduced, very little effort was devoted to the study of P^G_{CUT} on arbitrary graphs. Nevertheless, the optimum solution of spin glass instances on toroidal grid graphs with size up to 22 500 is reported in [25]. The algorithm used to obtain these results was a branch-and-cut algorithm based only on the cycle inequalities (5), for which "ad hoc" very effective separation procedures were designed.

Unfortunately, the cycle inequalities are far from being sufficient to solve spin glass instances on more complex graphs. Some work in progress [40] concerns new projection and lifting approaches to the separation for P^G_{CUT}. We now give a brief outline and refer to the publication for the details that are much more complicated than in the case of the TSP.

We are given a point $\bar{x}_n \in \mathbb{R}^E$ that satisfies all the inequalities (5) but does not belong to P^G_{CUT}, G being an arbitrary graph with n nodes. We want to find an inequality valid for P^G_{CUT} (possibly facet defining) that is not satisfied by $\bar{x}_n$. To do so, we want to use the algorithmic and the structural results that are available for $P^{K_p}_{\mathrm{CUT}}$, the cut polytope for a complete graph.

First, by a sequence of operations on $\bar{x}_n$ that amount to contracting 2-node sets of its support graph corresponding to the endnodes of integral weighted edges, such a point is transformed to $\bar{x}_{n'} \in \mathbb{R}^{E'}$, where n' is usually much smaller than n. Differently from the corresponding TSP case, the point $\bar{x}_{n'}$ is always guaranteed to be outside $P^{G'}_{\mathrm{CUT}}$ but to satisfy (5). It can be seen as a fractional solution of a cutting plane algorithm applied to a maximum cut instance on a smaller and denser graph $G' = (V', E')$ where $|V'| = n'$.

At this point all the machinery available for the maximum cut problem on complete graphs can be used for each complete subgraph K_p of G'. Therefore, some separation procedures for the cut polytope on complete graphs are applied to the restriction of $\bar{x}_{n'}$ to the edges of these components that (hopefully) generate an inequality $a_{n'}x_{n'} \geq \alpha$, valid for$P^{G'}_{\mathrm{CUT}}$ and violated by $\bar{x}_{n'}$ by an amount v.

Finally, a sequence of lifting procedures is applied to $a_{n'}x_{n'} \geq \alpha$ that transforms it to an inequality $a_n x_n \geq \beta$ valid for P^G_{CUT} and violated by $\bar{x}_n$ by the same amount v.

As a by-product, one of these lifting procedures provides a simple way to generate facet defining inequalities for P^G_{CUT}. Namely, under certain conditions, this procedure, applied to a facet defining inequality for $P^{G'}_{\mathrm{CUT}}$, produces not only a valid, but also a facet defining inequality for P^G_{CUT}.

In conclusion, these separation and lifting procedures enrich the description by linear inequalities of the cut polytope on arbitrary graphs and, at the same time, constitute an algorithmic tool for exactly solving the maximum cut problem on these graphs.

4.2 Branch-and-Cut-and-Price

Suppose we are given a combinatorial optimization problem $\mathcal{P}$ defined over a ground set E. A *feasible solution* for $\mathcal{P}$ is a suitable subset F of E. Let us denote by $\mathcal{F}$ the set of all feasible solutions for $\mathcal{P}$. A weight w_e is given for each element e of E and the problem consists of finding an element $F \in \mathcal{F}$ of maximum total weight $\sum_{e \in F} w_e$.

Consider the polytope defined as the convex hull of the characteristic vectors of all the feasible solutions $F \in \mathcal{F}$. This polytope can be described by a system of linear inequalities in the space $\mathbb{R}^E$

$$\begin{array}{l} Dx \leq g \\ 0 \leq x \leq 1 \end{array} \tag{6}$$

that typically has exponentially many rows. Let us assume that we have two alternative ways to solve $\mathcal{P}$: we have at hand a separation procedure for the associated polytope, thus we can solve it by a polyhedral cutting plane algorithm, but $\mathcal{P}$ can also be solved by a combinatorial algorithm. For example, $\mathcal{P}$ could be the *weighted matching problem*, in this case E would be the edge set of a given weighted graph and $\mathcal{F}$ the set of all its matchings. We know that this problem can be solved by a polyhedral cutting plane algorithm using the Padberg-Rao algorithm to generate violated blossom inequalities, but we can also solve the problem with Edmonds' blossom algorithm, which is of combinatorial type.

Let us consider now a different situation where not all elements of the set $\mathcal{F}$ are feasible, but only those whose characteristic vectors satisfy a set of linear inequalities $Ax \leq b$ (we assume that the cardinality of this set is bounded by a polynomial in $|E|$). In other words, we want to find an integral optimum solution of the system

$$\begin{array}{l} Ax \leq b \\ Dx \leq g \\ 0 \leq x \leq 1. \end{array} \tag{7}$$

The system (7) does not define, in general, an integral polytope as the system (6) does; moreover, the corresponding optimization problem $\mathcal{P}'$ is typically more difficult to solve than $\mathcal{P}$. Therefore, we may need to use branch-and-cut techniques to solve it. First of all, let us see how we can solve the linear programming relaxation defined by the system (7). There are two possible ways. One is to use a polyhedral cutting plane algorithm by generating a subset of the inequalities of (6), as we assumed that we know how to generate violated inequalities from this system. The second possibility is to use the decomposition of Dantzig-Wolfe [22] in the following way.

We know that any point x of the polytope described by (7) is a convex combination of the characteristic vectors of some elements of $\mathcal{F}$. Denoting by χ^F the characteristic vector of $F \in \mathcal{F}$, we have that

$$x = \sum_{F \in \mathcal{F}} y_F \chi^F, \text{ with } \sum_{F \in \mathcal{F}} y_F = 1 \text{ and } y \geq 0. \tag{8}$$

By substituting (8) into (7) we obtain a new system of inequalities

$$\begin{aligned} \tilde{A}y &\leq b \\ \mathbb{1}^T y &= 1 \\ y &\geq 0, \end{aligned} \tag{9}$$

where $\tilde{A} = [\tilde{A}_F]_{F\in\mathcal{F}} = [A\chi^F]_{F\in\mathcal{F}}$ and $\mathbb{1}$ is a vector of all ones. Optimizing a linear function $w^T x$ over the system (7) is equivalent to optimizing the linear function $\tilde{w}^T y$ over the system (9), where $\tilde{w} = [\tilde{w}_F]_{F\in\mathcal{F}} = [w^T\chi^F]_{F\in\mathcal{F}}$. We have thus replaced a linear program with $|E|$ variables and exponentially many constraints with another linear program with a number of constraints that is polynomial in $|E|$ and with exponentially many variables.

We call the system (7) the *ground set formulation* and the system (9) the *subset formulation* of problem $\mathcal{P}'$.

The maximization for the subset formulation can be solved with the following *column generation algorithm*. We select a "small" subset X of $\mathcal{F}$ and we solve the following linear program, where the subscript X denotes the restriction of the vectors and of the constraint matrix to the subset of variables associated with X,

$$\begin{aligned} \text{maximize} \quad & \tilde{w}_X \\ \text{subject to} \quad & \tilde{A}_X y_X \leq b \\ & \mathbb{1}_X^T y_X = 1 \\ & y_X \geq 0. \end{aligned} \tag{10}$$

Let $\bar{y}$ be the extension of the optimum solution of (10) to the whole set $\mathcal{F}$ obtained by adding zero valued components. Let u and v be the dual optimum solutions associated with the first set of constraints and with the equation, respectively.

In order to prove that $\bar{y}$ is the vector that maximizes $\tilde{w}$ over the set defined by (9), we have to compute the maximum among the reduced costs of all its components, by solving the following *column generation subproblem*:

$$\begin{aligned} \text{maximize} \quad & \tilde{w}_F - u^T\tilde{A}_F - v \\ & = w\chi^F - u^T A\chi^F - v \\ & = (w - u^T A)\chi^F - v \\ \text{subject to} \quad & F \in \mathcal{F} \end{aligned} \tag{11}$$

If the optimum value of (11) is less than or equal to zero, we have proved the optimality of $\bar{y}$; otherwise, let the maximum be attained at the solution $\bar{F}$. Then we add $\bar{F}$ to the set X, we extend the matrix $\tilde{A}_X$, and the vector $\tilde{w}_X$ with the column $A\chi^{\bar{F}}$ and the component $w\chi^{\bar{F}}$, respectively, and we solve the linear program (10) again.

Observe that the column generation subproblem is precisely the original optimization problem $\mathcal{P}$ (a weighted matching problem in our example) with $w - u^T A$ as the vector of weights. Since we have assumed up front that an (efficient) algorithm for $\mathcal{P}$ is available, (9) can be solved with a linear optimizer and a sequence of calls to this algorithm.

The column generation algorithm is typically preferable to a cutting plane algorithm applied to the system (7) when the optimization for $\mathcal{P}$ is simpler or more effectively solvable than the corresponding separation problem. Even though optimization and separation are polynomially equivalent, it is usually more effective to use a polynomial time combinatorial algorithm rather than a cutting plane algorithm. This is, for example, true for the weighted matching problem where state of the art combinatorial algorithms (see [3]) perform much better than cutting plane algorithms based on the Padberg-Rao separation procedure for the blossom inequalities. Going to NP-hard problems, it is known that certain instances of the binary knapsack problem can be solved effectively by dynamic programming or by some simple enumerative scheme, while is it quite unlikely that the same instances can be solved equally fast by polyhedral cutting plane methods, if they can be solved at all without resorting to enumeration.

The scheme we have discussed so far can be generalized to the more interesting case where k combinatorial optimization problems are given on k disjoint ground sets. The variables associated to the elements of all the ground sets are then linked together by some additional constraints. Because of these additional constraints not all k-tuples of feasible solutions (one taken from each problem) are feasible for the whole problem. Many interesting problems can be formulated in this way. For example, the binary cutting stock problem described in Section 2, where each of the k problems is a knapsack, is of this type. Another example is given by the capacitated vehicle routing problem, there each of the k building blocks is a minimum length cycle problem with side constraints. For the sake of simplicity, we will not consider this more general setting here and we will stick to the simple case, where $k = 1$, developed so far.

If the solution $\bar{y}_X$ of (10) is not integer, we have to resort to one of the two main operations of branch-and-cut: either we branch on a fractional variable $\bar{y}_{\bar{F}}$ or we add a new valid inequality that cuts the point $\bar{y}$ away. Both these operations present some difficulties when the linear programming relaxation is solved by a column generation algorithm.

Suppose we branch on variable $y_{\bar{F}}$, so we do a variable setting that is equivalent to adding the constraint $y_{\bar{F}} = 0$ to the problem (10) and we solve it again. Then we must check if any variables corresponding to feasible solutions in $\mathcal{F} \setminus X$ have positive reduced cost. However, while without variable setting all variables associated with the set X have non-positive reduced cost, now $y_{\bar{F}}$ does have positive reduce cost and $\bar{F} \in X$. Therefore, it may be the case that the column generation subproblem selects right $\bar{F}$ as the new element to be put in the set X. To avoid this dead-lock, in the column generation subproblem we have to look for the second best solution or, when in general $i-1$ variables have been already set, for the i-th best solution. However, finding the i-th best solution is in general more difficult than simply finding an optimum one. There are polynomially solvable problems for which the i-th best version is NP-hard. Thus branching may produce inefficiencies in the column generation subproblem.

To avoid such inefficiencies, some "ad hoc" branching strategies have been proposed in the literature for specific column generation subproblems. For example, in the case when (11) is a knapsack problem, Vance et al. [68] propose

a branching strategy that leaves the structure of the column generation subproblem almost unchanged. Nevertheless, even in this nice case, there are some unfortunate situations where the subproblem becomes considerably harder and needs to be solved by a general IP optimizer.

It is even more difficult to add cutting planes to the linear program. The program (10), with the addition of the integrality requirement for the variables y, is an integer linear program for which several techniques exist to generate a valid inequality $c^T y_X \leq d$ that cuts the solution $\bar{y}_X$ away. However, in order to exploit this cutting plane to strengthen the current linear programming relaxation within a column generation scheme, the following two conditions must be satisfied:

a) For each $F \notin X$ it must be easy to compute the lifting coefficient c_F.
b) The column generation subproblem must keep the original structure, or, at least, must maintain the same level of difficulty after the cut has been added.

Condition a) is quite difficult to satisfy in general. Moreover, even in case the lifting coefficient can be computed, i.e., when we can find a function $c : \mathcal{F} \to \mathbb{R}$ that associates a coefficient with every feasible solution of $\mathcal{F}$, the situation may not be tractable yet. After introducing the inequality $c^T y \leq d$ into the linear program, denoting by t its corresponding dual variable in the new optimum solution, the column generation subproblem becomes

$$\begin{array}{ll} \text{maximize} & (w - u^T A)\chi^F - tc_F - v \\ \text{subject to} & F \in \mathcal{F}. \end{array} \tag{12}$$

The complexity of such a problem depends on the function c and it is conceivably hard in most of the cases.

For these reasons the cut generation phase never takes place when the linear programming relaxation is solved with column generation techniques. Such a simplified version of branch-and-cut where only new columns are added to the constraint matrix, and never new rows, is called *branch-and-price* and has been successfully applied to solve some large problems. It is evident, though, that such a technique can produce good results only when the gap between the objective function value of the integral optimum solution and the optimum of the linear programming relaxation (10) is sufficiently small. When this is not the case, the lack of the cutting plane phase may induce poor performance of the algorithm.

To overcome these difficulties, Felici, Gentile, and Rinaldi [30] have proposed the following method that makes it possible to use the branch-and-cut methodology at its full power.

Instead of choosing one of the *ground set* and the *subset* formulations, the solution algorithm considers both of them. The latter is used to solve the current linear programming relaxation while the former is used to find violated cuts and to do the branching. Let us see in more detail how this "double-formulation" algorithm works.

We solve the current linear program (10) and we find the solution $\bar{y}_X$. Then we translate $\bar{y}_X$ into a feasible solution $\bar{x}$ for the system (7) by

$$\bar{x} = \sum_{F \in X} y_F y_F.$$

It is easy to see that $\bar{x}$ optimizes the linear function $w^T x$ over the system (7). Now we assume that $\bar{x}$ is not integral and that we are able to solve the separation problem for this point with respect to the convex hull of the integral solutions of (7). Let $c^T x \leq d$ be the inequality identified by the separation algorithm. Then the inequality $\tilde{c}_X^T y_X \leq d$ is generated using the following map that assigns a coefficient to each feasible solution in $\mathcal{F}$:

$$\tilde{c}_F = c^T \chi^F. \tag{13}$$

The inequality $\tilde{c}_X^T y_X \leq d$ is valid for all the integral solutions of the linear program (9) and is violated by $\bar{y}_X$. Thus, it is added to the constraint matrix of the linear program (10). Observe that by (13) we can now compute the lifting coefficient $\tilde{c}_F$ for each $F \in \mathcal{F}$. Moreover, due to the special form of the lifting coefficients, the column generation subproblem after adding the new constraint, whose associated dual variable is denoted by t, becomes

$$\begin{aligned} \text{maximize} \quad & \tilde{w}_F - u^T \tilde{A}_F - t\tilde{c}_F - v \\ & = w\chi^F - u^T A\chi^F - tc^T\chi^F - v \\ & = (w - u^T A - tc^T)\chi^F - v \\ \text{subject to} \quad & F \in \mathcal{F} \end{aligned} \tag{14}$$

which differs from the original subproblem (11) only in the objective function. As the structure of this subproblem remains unchanged after adding the new inequality, the column generation phase and the cutting plane phase can be integrated without interfering with each other. The branching phase can be handled analogously: if the current solution $\bar{x}$ has a fractional component, say $\bar{x}_e$, we generate two new problems to which we add the two (non-valid) inequalities $x_e \leq 0$ and $x_e \geq 1$, respectively. Branching on inequalities rather than on variables is treated in the same way. The counterpart of these inequalities in the corresponding subset formulation is produced using (13).

In conclusion, using this technique, a full branch-and-cut scheme can be used even when columns are dynamically added to the constraint matrix via a column generation process that extracts them from an exponentially large set. To stress this aspect, we use the name *branch-and-cut-and-price* for this version of the algorithm.

In [30] Gentile, Felici, and Rinaldi used a branch-and-cut-and-price algorithm to produce optimum or nearly optimum solutions to a complex mixed integer problem arising in the scheduling of ships for product distribution in the oil industry. The model is originated by k distinct combinatorial problems whose variables are linked by additional constraints.

5 Design and Usage of ABACUS

In Section 3 we introduced a generic algorithmic framework for branch-and-cut (and price) algorithms that is common to almost all software implementations of such algorithms. In contrast to this generality almost all implementations of branch-and-cut algorithms had been done from scratch for a long time. In order to overcome the tremendous waste of time used for repeatedly implementing similar computer codes sharing the same algorithmic techniques the software system ABACUS—A Branch And CUt System has been developed. From an early predecessor that was a C-library developed by Michael Jünger and Gerhard Reinelt in the late eighties, the system underwent several minor and two complete revisions, until Stefan Thienel [64] developed ABACUS as an object oriented system in his doctoral thesis.

ABACUS provides a framework for implementation of branch-and-bound algorithms using linear programming relaxations that can be completed with the dynamic generation of cutting planes or columns. This system allows the software developer to concentrate merely on the problem specific parts, i.e., the cutting plane part, the column generation, and the heuristics. Moreover ABACUS provides a variety of general algorithmic concepts, e.g., enumeration and branching strategies, from which the user of the system can choose the best alternative for his application. If the provided features do not suffice or there are better problem specific techniques it is easy to extend the system. Finally, ABACUS provides many basic data structures and useful tools for implementation of extensions. ABACUS is designed both for general mixed integer problems and combinatorial optimization problems but is especially useful for combinatorial optimization. Other systems (like BC-OPT [19], MINTO [53], Cplex [17] and XPress [69]) emphasize on mixed integer optimization.

Simple reuse of code and design of abstract data structures are essential for a modern software framework. These requirements are met by object oriented programming techniques. Therefore ABACUS was implemented as a C++ class library, i.e., a collection of C++ classes. ABACUS uses extensively the object oriented features of C++ and is not only a C++ adaption of C code. In order to understand its design and use, some experience with C++ is a prerequisite. Since C++ has become some of the most popular programming languages and will probably not loose its popularity in the near future, choosing C++ was a reasonable decision.

5.1 Basic Design Ideas

From the point of view of a user, who wants to implement a linear programming based branch-and-bound algorithm, ABACUS provides a small system of base classes from which the application specific classes can be derived. All problem independent parts are "invisible" for the user such that one can concentrate on the problem specific algorithms and data structures.

The basic ideas are pure virtual functions, virtual functions, and virtual dummy functions. A pure virtual function has to be implemented in a class

derived by the user of the framework, e.g., the initialization of the branch-and-bound tree with a subproblem associated with the application. In virtual functions default implementations are provided. They are often useful for a big number of applications, but can be redefined if required. The branching strategy is a good example. Finally, virtual dummy functions are used that are virtual functions that do nothing in their default implementations, but can be redefined in derived classes, e.g., the separation of cutting planes. They are not pure virtual functions as their definition is not required for the correctness of the algorithm.

Moreover, an application based on ABACUS can be refined step by step. Only the derivation of a few new classes and the definition of some pure virtual functions is required to get a branch-and-bound algorithm running. Then, this branch-and-bound algorithm can be enhanced by the dynamic generation of constraints and/or variables, heuristics, or the implementation of new branching or enumeration strategies.

Default strategies are available for numerous parts of the branch-and-bound algorithm, and they can be controlled via a parameter file. If none of the system strategies meets the requirements of the application, the default strategy can simply be replaced by the redefinition of a virtual function in a derived class.

5.2 Structure of the System

The inheritance graph of any set of classes in C++ must be a directed acyclic graph. Very often these inheritance graphs form forests or trees. Also the inheritance graph of ABACUS is designed as a tree with a single exception where multiple inheritance is used.

Basically, the classes of ABACUS can be divided into three different main groups shown in Table 2. The application base classes are the most important ones for the user. From these classes the user of the framework has to derive the classes for her or his applications. The pure kernel classes are usually invisible for the user. To this group belong, e.g., classes for supporting the branch-and-bound algorithm, for the solution of linear programs, and for the management of constraints and variables. Finally, there are the auxiliaries, i.e., classes providing basic data structures and tools that can optionally be used for the implementation of an application.

Table 2. The classes of ABACUS.

Pure Kernel	Application Base	Auxiliaries
Linear Program	Master	Basic Data Structures
Pool	Subproblem	Tools
branch-and-bound	Constraints	
	Variables	

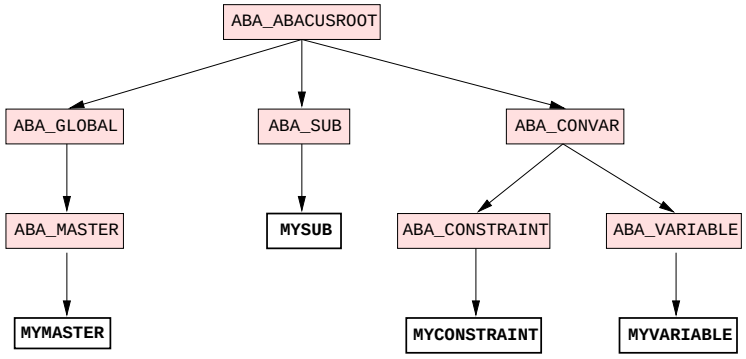


Fig. 14. Inheritance relations between ABACUS kernel classes and user defined derivations.

5.3 Essential Classes for an Applications

In this section we describe the ABACUS classes that are usually involved in the derivation process for the implementation of a new application. We give their description not in the form of a manual by describing each member of the class (this is done in [1]), but we try to explain the concepts, design ideas, and usage. Fig. 14 shows the class graph of the application base classes and the derived application specific classes (represented by white boxes with bold font).

The Root of the Class-Tree. It is well known that global variables, constants, or functions, can cause a lot of problems within a big software system. This is even worse for frameworks such as ABACUS that are used by other programmers and may be linked together with other libraries. Here, name conflicts and undesired side effects are almost inevitable.

All functions and enumerations that might be used by all other classes are embedded in the class `ABA_ABACUSROOT`. This class is used as a base class for all classes within the system. Currently, `ABA_ABACUSROOT` implements only an enumeration with the different exit codes of the framework and implements some public member functions. The most important one of them is the function `exit()` that calls the system function `exit()`. This construction turns out to be very helpful for debugging purposes.

The Master. In an object oriented implementation of a linear programming based branch-and-bound algorithm we require one object that controls the optimization, in particular the enumeration and resource limits, and stores data that can be accessed from any other object involved in the optimization of a specific instance. This task is performed by the class `ABA_MASTER` (that is not identical with the root node of the enumeration tree). For each application of

ABACUS we have to derive a class from ABA_MASTER implementing problem specific "global" data and functions.

Every object that requires access to this "global" information, stores a pointer to the corresponding object of the class ABA_MASTER. This holds for almost all classes of the framework. For example, the class ABA_SUB, implementing a subproblem of the branch-and-bound tree, has as a member a pointer to an object of the class ABA_MASTER:

```
class ABA_SUB {
  ABA_MASTER *master_;
};
```

Then, we can access within a member function of the class ABA_SUB, e.g., the global upper bound by calling

```
master_->upperBound();
```

where upperBound() is a member function of the class ABA_MASTER.

Within a specific application there are always some global data members as the output and error streams, zero tolerances, a big number representing "infinity", and some functions related with these data. Instead of implementing this data directly in the class ABA_MASTER, ABACUS has an extra class ABA_GLOBAL, from which the class ABA_MASTER is derived. The reason is that there are several classes, especially some basic data structures that might be useful in programs that are not branch-and-bound algorithms. To simplify their reuse, these classes have a pointer to an object of the class ABA_GLOBAL instead of one to an object of the class ABA_MASTER.

Branch-and-bound Data and Functions. The class ABA_MASTER augments the data inherited from the class ABA_GLOBAL with specific data members and functions for branch-and-bound. It has objects of classes as members that store the list of subproblems that still must be processed in the implicit enumeration (class ABA_OPENSUB), and that store the variables that might be fixed by reduced cost criteria in later iterations (class ABA_FIXCAND). Moreover, the solution history, timers for parts of the optimization, and a lot of other statistical information is stored within the class ABA_MASTER.

The class ABA_MASTER also provides default implementations of pools for the storage of constraints and variables. We explain the details later in this section.

A branch-and-bound framework requires a flexible way for defining enumeration strategies. The corresponding virtual functions are defined in the class ABA_MASTER, but for a better understanding we explain this concept below, when we discuss the data structure for the open subproblems.

Limits on the Optimization Process. The control of limits on the optimization process, e.g., the amounts of CPU time and wall-clock time, and the size of the enumeration tree, are performed by members of the class ABA_MASTER during the optimization process. Also the quality guarantee of the solution is monitored by the class ABA_MASTER.

The Initialization of the Branch-and-Bound Tree. When the optimization is started, the root node of the branch-and-bound tree must be initialized with an object of the class `ABA_SUB`. However, the class `ABA_SUB` is an abstract class, from which a class implementing the problem specific features of the subproblem optimization is derived. Therefore, the initialization of the root node is performed by a pure virtual function that must return a pointer to a class derived from the class `ABA_SUB`. This function must be defined by a problem specific class derived from the class `ABA_MASTER`.

The Sense of the Optimization. For simplification software systems for minimization and maximization problems use internally only one sense of the optimization, e.g., minimization. Within a framework this strategy is dangerous, because if we access internal results, e.g., the reduced costs, from an application, we might misinterpret them. Therefore, ABACUS also works internally with the true sense of optimization. The value of the best known feasible solution is denoted *primal bound*, the value of a linear programming relaxation is denoted *dual bound* if all variables price out correctly. The functions `lowerBound()` and `upperBound()` interpret the primal or dual bound, respectively, depending on the sense of the optimization. An equivalent method is also used for the local bounds of the subproblems.

The Subproblem. The class `ABA_SUB` represents a subproblem of the implicit enumeration, i.e., a node of the branch-and-bound tree. It is an abstract class, from which a problem specific subproblem can be derived. In this derivation process problem specific functions can be added, e.g., for the generation of variables or constraints.

The Root Node of the Branch-and-Bound Tree. For the root node of the optimization, the constraint and variable sets can be initialized explicitly. By default, the first linear program is solved with the barrier method followed by a crossover to a basic solution, but a flexible mechanism for the selection of the LP-method is provided.

The Other Nodes of the Branch-and-Bound Tree. As long as only globally valid constraints and variables are used, it would be correct to initialize the constraint and variable system of a subproblem with the system of the previously processed subproblem. However, ABACUS is designed also for locally valid constraints and variables. Therefore, each subproblem inherits the final constraint and variable system of the father node in the enumeration tree. This system might be modified by the applied branching rule. Moreover, this approach avoids tedious recomputations and makes sure that heuristically generated constraints do not get lost.

If conventional branching strategies, like setting a binary variable, changing the bounds of an integer variable, or even adding a branching constraint are applied, then the basis of the last solved linear program of the father is still dual

feasible. As the basis status of the variables and slack variables is stored phase 1 of the simplex method can be avoided if we use the dual simplex method. If due to another branching method, e.g., for branch-and-price algorithms, the dual feasibility of the basis is lost, another LP-method can be used.

Branch-and-Bound. A linear programming based branch-and-bound algorithm in its simplest form is obtained if linear programming relaxations in each subproblem are solved that are neither enhanced by the generation of cutting planes nor by the dynamic generation of variables. Such an algorithm requires only two problem specific functions: one to check if a given LP-solution is a feasible solution of the optimization problem, and one for the generation of the sons.

The first function is problem specific, because, if constraints of the integer programming formulation are violated, the condition that all discrete variables have integer values is not sufficient. For safety, this function is declared pure virtual. The second required problem specific function is usually only a one-liner, that returns the problem specific subproblem generated by a branching rule. Hence, the implementation of a pure branch-and-bound algorithm does not require very much effort.

The Optimization of the Subproblem. The core of the class `ABA_SUB` is its optimization by a cutting plane algorithm. As dynamically generated variables are dual cuts, we also use the notion cutting plane algorithm for a column generation algorithm. By default, the cutting plane algorithm only solves the LP-relaxation and tries to fix and set by reduced costs. Within the cutting plane algorithm four virtual dummy functions for the separation of constraints, for the pricing of variables, for the application of LP-based heuristics, and for fixing variables by logical implications are called. These virtual functions can be redefined in a problem specific class derived from the class `ABA_SUB`. In addition to the mandatory pricing phase before the fathoming of a subproblem, the inactive variables are priced out every k iterations in a branch-and-cut and price algorithm. Other strategies for the separation/pricing decision can be implemented by the redefinition of a virtual function.

Adding Constraints. Cutting planes may not only be generated in the function `separate()` but also in other functions of the cutting plane phase. E.g., for the maximum cut problem it is useful if the generation of cutting planes is also possible in the LP-based heuristic. If not all constraints of the integer programming formulation are active, then it might be necessary to solve a separation problem also for the feasibility test. Therefore, the generation of cutting planes is allowed in every subroutine of the cutting plane algorithm.

Adding Variables. Like for constraints, also the generation of variables is allowed everywhere in the subproblem optimization.

Buffering New Constraints and Variables. New constraints and variables are not immediately added to the subproblem, but stored in buffers and added at the beginning of the next iteration. We present the details of this concept below.

Removing Constraints and Variables. In order to avoid corrupting the linear program and the sets of active constraints and variables, and to allow the removal of variables and constraints in any subroutine of the cutting plane phase, also these variables and constraints are buffered. The removal is executed before constraints and variables are added at the beginning of the next iteration of the cutting plane algorithm.

Moreover, a default function for the removal of constraints according to the value or the basis status of the slack variables is provided. Variables can be removed according to the value of the reduced costs. These operations can be controlled by parameters and the corresponding virtual functions can be redefined if other criteria should be applied. ABACUS tries to remove constraints also before a branching step is performed.

The Active Constraints and Variables. In order to allow a flexible combination of constraint and variable generation, every subproblem has its own set of active constraints and variables, that are represented by the generic class `ABA_ACTIVE`. By default, the variables and the constraints of the last solved linear program of the father of the subproblem are inherited. Therefore, the local constraint and variable sets speed up the optimization.

Together with the active constraints and variables ABACUS also stores in every subproblem the LP-statuses of the variables and slack variables, the upper and lower bounds of the variables, and if a variable is fixed or set.

The Linear Program. Every subproblem has its own linear program that is only set up for an active subproblem. Of course, the initialization of the linear program at the beginning and its deletion at the end of the subproblem optimization costs some running time in comparison to the considerable maintenance of a global linear program that could be stored in the master. Our current computational experience shows that this overhead is not too big and pays because bookkeeping is greatly facilitated and parallelization is easy possible.

The LP-Method. Currently, three different methods are available in state of the art LP-solvers: the primal simplex method, the dual simplex method, and the barrier method in combination with cross over techniques for the determination of an optimum basic solution. The choice of the method can be essential for the performance. If a primal feasible basis is available, the primal simplex method is often the right choice. If a dual feasible basis is available, the dual simplex method is usually preferred. And finally, if no basis is known, or the linear programs are very large, often the barrier methods yield the best running times.

Therefore, by default a linear program is solved by the barrier method, if it is the first linear program solved in the root node or constraints and variables have been added at the same time, by the primal simplex method, if constraints have been removed or variables have been added, and by the dual simplex method, if constraints have been added, or variables have been removed, or it is the first linear program of a subproblem that is not the root node.

However, it should be possible to add problem specific decision criteria. Again, a virtual function gives us all flexibility. We keep control when this function is invoked, namely at the point when all decisions concerning addition and removal of constraints and variables have been taken. The function has as arguments the correct numbers of added and removed constraints and variables. If we want to choose the LP-method problem specifically, then we can redefine this function in a class derived from the class `ABA_SUB`.

Generation of Non-liftable Constraints. If constraint and variable generation are combined, then the active constraints must be lifted if a variable is added, i.e., the column of the new variable must be computed. This lifting can not always be done in a straightforward way, it can even require the solution of another optimization problem. Moreover, lifting is not only required when a variable is added, but this problem has to be attacked already during the solution of the pricing problem.

In order to allow the usage of constraints that cannot be lifted or for which the lifting cannot be performed efficiently, ABACUS provides a management of non-liftable constraints. Each constraint has a flag if it is liftable. If the pricing routine is called and non-liftable constraints are active, then all non-liftable constraints are removed, the linear programming relaxation is solved again, and the cutting plane algorithm is continued before the pricing phase is reentered. In order to avoid an infinite repetition of this process we forbid the further generation of non-liftable constraints during the rest of the optimization of this subproblem.

Reoptimization. If the root of the remaining branch-and-bound tree changes, but the new root has been processed earlier, then it can be advantageous to optimize the corresponding subproblem again, in order to get improved conditions for fixing variables by reduced costs. Therefore, ABACUS provides the reoptimization of a subproblem. The difference to the ordinary optimization is that no branching is finally performed even if the subproblem is not fathomed. If it turns out during the reoptimization that the subproblem is fathomed, then all subproblems contained in the subtree rooted at this subproblem are fathomed.

Branching. Virtual functions for the flexible definition of branching strategies are implemented in the class `ABA_SUB`. We explain below.

Memory Allocation. Since constraints and variables are added and removed dynamically, ABACUS provides a dynamic memory management system, that requires no user interaction. If there is not enough memory to add a constraint or variable, memory reallocations are performed automatically. As the reallocation of the local data, in particular of the linear program, can require a lot of CPU time, if it is performed regularly, some extra space is allocated for the addition of variables and constraints, and for the nonzero entries of the matrix of the LP-solver.

Activation and Deactivation. In order to save memory, ABACUS sets up those data structures that are only required if the subproblem is active, e.g., the linear program, at the beginning of the subproblem optimization, and frees the memory again when the subproblem becomes inactive. We observed that the additional CPU time required for these operations is negligible, but the memory savings are significant.

Constraints and Variables. Motivated by linear programming duality, common features of constraints and variables are embedded in a joint base class `ABA_CONVAR`.

Constraint/Variable versus Row/Column. Usually, the notions constraint and row, and the notions variable and column, respectively, are used equivalently. We also followed this terminology so far since, e.g., the notion column generation algorithm is more common than variable generation algorithm. Within ABACUS, constraints and rows are different items. Constraints are stored in the pool, and a subproblem has a set of active constraints. Only if a constraint is added to the linear program, then the corresponding row is computed. More precisely, a row is a representation of a constraint associated with a certain variable set.

The subtour elimination constraints for the TSP provide a good example for the usefulness of this differentiation: Storing such an inequality

$$\sum_{i \in S} \sum_{j \in V \setminus S} x_{ij} \geq 2$$

would require storing all edges (variable) in $\delta(S)$ with space requirement $\Omega(|S|^2)$. But storing the elements in S requires only $O(S)$ space. Given a variable x_e associated with edge e the coefficient of the subtour elimination constraint is 1 if $e \in \delta(S)$ and 0 otherwise. Thus not only the generation of the constraint for a given set of active variables, but also the determination of of the lifting coefficients for other variables during pricing or variable generation is easy in this case.

Efficient memory management and dynamic variable generation are the reason why ABACUS distinguishes between constraints and rows. Each constraint must have a member function that returns the coefficient for a variable such that we can determine the row corresponding to a set of variables.

In these considerations "constraint" can be also replaced by "variable" and "row" by "column". A column is the representation of a variable corresponding to a given constraint set. Again, we use the TSP as an example. A variable for the TSP corresponds to an edge in a graph. Hence, it can be represented by its end nodes. The column associated with this variable consists of the coefficients of the edge for all active constraints.

ABACUS implements these concepts in the classes `ABA_CONSTRAINT` and `ABA_VARIABLE` that are used for the representation of active constraints and variables and for the storage of constraints and variables in the pools, and `ABA_ROW` and `ABA_COLUMN` that are used in the interface to the LP-solver.

Common Features of Constraints and Variables. Constraints and variables have several common features that are presented in a common base class. A constraint/variable is active if it belongs to the constraint/variable set of an active subproblem. An active constraint/variable must not be removed from its pool. Besides being active there can be other reasons why a constraint/variable should not be deleted from its pool, e.g., if the constraint/variable has just been generated, then it is put into a buffer, but is not yet activated. In such a case we want to set a lock on the constraint that it cannot be removed (we explain the details below).

ABACUS also distinguishes between dynamic variables/constraints and static ones. As soon as a static variable/constraint becomes active it cannot be deactivated. An example for static variables are the variables in a general mixed integer optimization problem, examples for static constraints are the constraints of the problem formulation of a general mixed integer optimization problem or the degree constraints of the TSP. Dynamic constraints are usually cutting planes. Column generation algorithms feature dynamic variables.

A crucial point in the implementation of a special variable or constraint class is the tradeoff between performance and memory usage. It has been observed that a memory efficient storage format can be one of the keys to the solution of larger instances. Such formats are in general not very useful for the computation of the coefficient of a single variable/constraint. Moreover, if the coefficients of a constraint for several variables or the coefficients of a variable for several constraints have to be computed, e.g., when the row/column format of the constraint/variable is generated in order to add it to the LP-solver, then these operations can become a bottleneck. However, given a different format, using more memory, it might be possible to perform these operations more efficiently.

Therefore, ABACUS provides a compressed format and an expanded format of a constraint/variable. Before a large number of time consuming coefficient computations is performed, the system tries to generate the expanded format, and afterwards the constraint/variable is compressed. The implementation of the expansion and compression is optional.

We use again the subtour elimination constraint of the TSP as an example for the compressed and expanded format. For an inequality $\sum_{i \in S} \sum_{j \in V \setminus S} x_{ij} \geq 2$ we store the nodes of the set S in the compressed format. The computation of the coefficient of an edge $e = (u, v)$ requires $O(|S|)$ time and space. As expanded format we use an array `inSubtour` of type `bool` of length n (n is the number of nodes of the graph) and `inSubtour[v]` is true if and only if $v \in S$. Now, we can determine the coefficient of an edge (variable) in constant time.

Constraints. ABACUS provides all three different types of constraints: equations, $\leq$-inequalities and $\geq$-inequalities. The only pure virtual function is the computation of a coefficient of a variable. It is used to generate the row format of a constraint, to compute the slack of an LP-solution, and to check if an LP-solution violates a constraint. All these functions are declared virtual such that they can be redefined for performance enhancements.

If variables are generated dynamically, ABACUS distinguishes between liftable and non-liftable constraints. Non-liftable constraints must be removed before the pricing problem can be solved.

Variables. ABACUS supports continuous, integer, and binary variables in the class `ABA_VARIABLE`. Each variable has a lower and an upper bound that can be set to plus/minus infinity if the variable is unbounded. We also memorize if a variable is fixed.

The corresponding functions have their dual analogs in the class `ABA_CONSTRAINT`. The only pure virtual function is now the function that returns a coefficient in a constraint. With this function the generation of the column format and the computation of the reduced cost can be performed. We say a variable is violated if it does not price out correctly.

Constraint and Variable Pools. Every constraint and variable either induced by the problem formulation or generated in a separation or pricing step is stored in a pool. A pool is a collection of constraints and variables. We will see later that it is profitable to keep separate pools for variables and constraints. Then, we will also discuss when it is useful to have different pools for different types of constraints or variables. But for simplicity we assume now that there is only one variable pool and one constraint pool.

There are two reasons for the usage of pools: saving memory and an additional separation/pricing method.

A constraint or variable usually belongs to the set of active constraints or variables of several subproblems that still have to be processed. Hence, it is useful to store in the sets of active constraints or variables only pointers to each constraint or variable that is stored at some central place, i.e., in a pool that is a member of the corresponding master of the optimization. Our practical experiments show that this memory sensitive storage format is of very high importance, since already this pool format uses a large amount of memory.

Pool-Separation/Pricing. From the point of view of a single subproblem a pool may not only contain active but also inactive constraints or variables. The inactive items can be checked in the separation or pricing phase, respectively. We call these techniques pool-separation and pool-pricing. Again, motivated by duality theory we use the notion "separation" also for the generation of variables, i.e., for pricing. Pool-separation is advantageous in two cases. First, pool-separation might be faster than the direct generation of violated constraints or variables. In this case, we usually check the pool for violated constraints or variables, and only if no item is generated, we use the more time consuming direct methods. Second, pool-separation turns out to be advantageous, if a class of constraints or variables can be separated/priced out only heuristically. In this case, it can happen that the heuristic cannot generate the constraint or variable although it is violated. However, earlier in the optimization process this constraint or variable might have been generated. In this case the constraint or variable can be regenerated from the pool. Computational experiments in [41] show that this additional separation or pricing method can decrease the running time significantly.

Pool-separation is also one of the reasons why it can be helpful to provide several constraint or variable pools. E.g., some constraints might be more important during the pool-separation than other constraints. In this case, we might check this "important" pool first and only if we fail in generating any item we might proceed with other pools or continue immediately with direct separation techniques.

Other classes of constraints or variables might be less important in the sense that they cannot or can only very seldomly be regenerated from the pool (e.g., locally valid constraints or variables). Such items could be kept in a pool that immediately removes all items that do not belong to the active constraint or variable set of any subproblem that still has to be processed. A similar strategy might be required for constraints or variables requiring a big amount of memory.

Finally, there are constraints for which it is advantageous to stay active in any case (e.g., the constraints of the problem formulation in a general mixed integer optimization problem, or the degree constraints for the TSP). Also for these constraints separate pools are useful.

Garbage Collection. In any case, as soon as a lot of constraints or variables are generated dynamically we can observe that the pools become very large. In the worst case this might cause an abnormal termination of the program if it runs out of memory. But already earlier the optimization process might be slowed down since pool-separation takes too long. Of course, the second point can be avoided by limited strategies in pool-separation, which we will discuss later. But to avoid the first problem we require suitable cleaning up and garbage collection strategies.

The simplest strategy is to remove all items belonging not to any active variable or constraint set of any active or open subproblem in a garbage collection process. The disadvantage of this strategy might be that good items are removed that are accidentally momentarily inactive. A more sophisticated strategy might be counting the number of linear programs or subproblems where this item has been active and removing initially only items with a small counter.

Unfortunately, if the enumeration tree grows very large or if the number of constraints and variables that are active at a single subproblem is high, then even the above brute force technique for the reduction of a pool turns out to be insufficient.

Hence, ABACUS divides constraints and variables into two groups. On the one hand the items that must not be removed from the pool, e.g., the constraints and variables of the problem formulation of a general mixed integer optimization problem, and on the other hand those items that can either be regenerated in the pricing or separation phase or are not important for the correctness of the algorithm, e.g., cutting planes. If we use the data structures we will describe now, then we can remove safely an item of the second group.

Pool Slots. So far, we have assumed that the subproblems store pointers to variables or constraints, respectively, which are stored in pools. If we remove the variable or constraint, i.e., delete the memory we have allocated for this

object, then errors will occur if we access the removed item from a subproblem. ABACUS contains a data structure that can avoid this problem very efficiently.

A pool is not a collection of constraints or variables, but a collection of pool slots (class ABA_POOLSLOT). Each slot stores a pointer to a constraint or variable or a 0-pointer if it is void. The sets of active constraints or variables in subproblems consist of pointers to the corresponding slots instead of storing pointers to the constraints or variables directly. If a constraint or variable has been removed, a 0-pointer will be found in the slot and the subproblem recognizes that the constraint or variable must be eliminated since it cannot be regenerated. The disadvantage of this method is that finally our program may run out of memory since there are many useless slots.

In order to avoid this problem, a version number is added as data member to each pool slot. Initially the version number is 0 and becomes 1 if a constraint or variable is inserted in the slot. After an item in a slot is deleted, a new item can be inserted into the slot. Each time a new item is stored in the slot the version number is incremented. The sets of active constraints and variables do not only store pointers to the corresponding slots but also the version number of the slot when the pointer is initialized. If a member of the active constraints or variables is accessed we compare its original and current version number. If these numbers are not equal we know that this is not the constraint or variable we were originally pointing to and remove it from the active set. We call the data structure storing the pointer to the pool slot and the original version number a reference to a pool slot (class ABA_POOLSLOTREF). Hence, the sets of active constraints and variables are arrays of references to pool slots. This pool concept is illustrated in Fig. 15.

Standard Pool. The class ABA_POOL is an abstract class that does not specify the storage format of the collection of pool slots. The simplest implementation is an array of pool slots. The set of free pool slots can be implemented by a linked list. This concept is realized in the class ABA_STANDARDPOOL. Moreover, a ABA_STANDARDPOOL can be static or dynamic. A dynamic ABA_STANDARDPOOL is automatically enlarged, when it is full, an item is inserted, and the cleaning up procedure fails. A static ABA_STANDARDPOOL has a fixed size and no automatic reallocation is performed. More sophisticated implementations might keep an order of the pool slots such that "important" items are detected earlier in a pool-separation and a limited pool-separation might be sufficient. A criterion for this order could be the number of subproblems where this constraint or variable is active or has been active.

Default Pools. The number of the pools is very problem specific and depends mainly on the separation and pricing methods. Since in many applications a pool for variables, a pool for the constraints of the problem formulation, and a pool for cutting planes are sufficient, ABACUS implements this default concept. If not specified differently these default pools are used in the initialization of the pools, in the addition of variables and constraints, and in the pool-pricing and pool-separation. ABACUS uses a static ABA_STANDARDPOOL for the default constraint and cutting planes pools. The default variable pool is a dynamic ABA_STANDARDPOOL, because the correctness of the algorithm requires that a

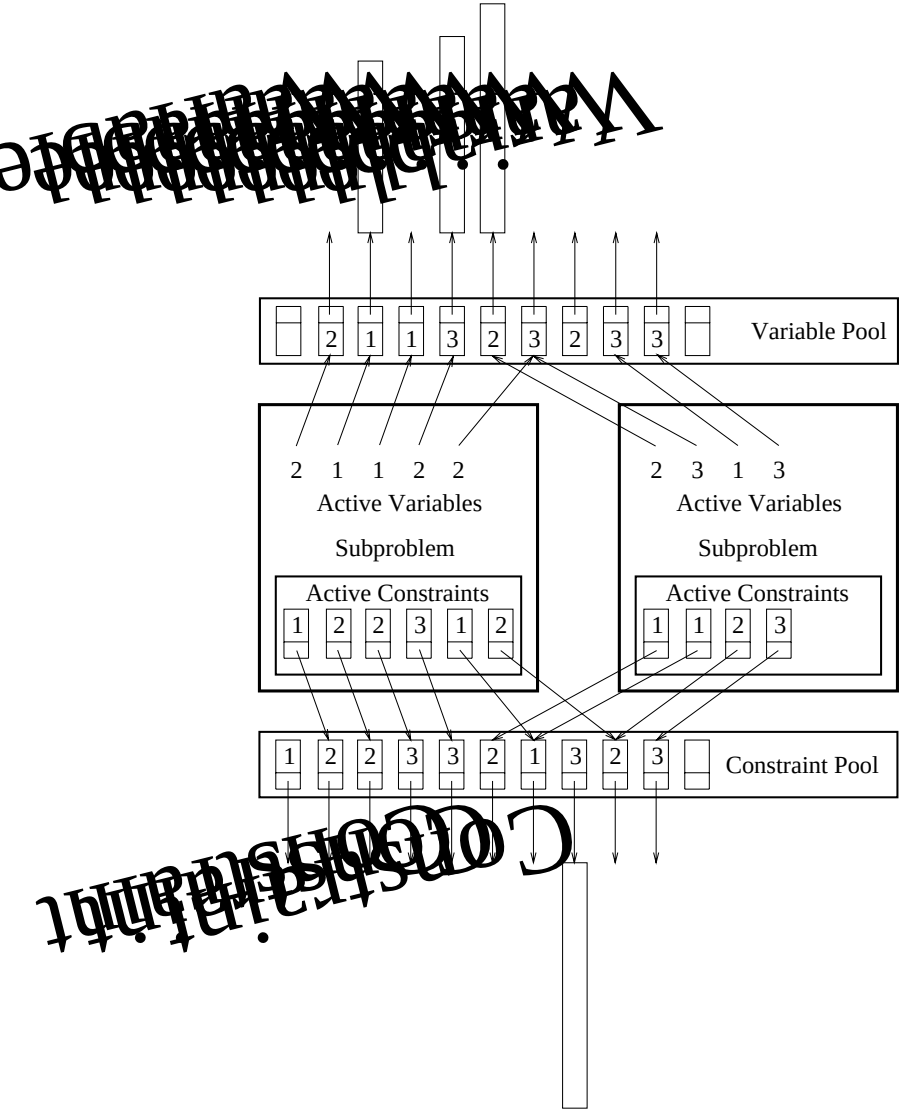


Fig. 15. Schematic illustration of the pool/poolslot concept.

variable that does not price out correctly can be added in any case, whereas the loss of a cutting plane that cannot be added due to a full pool has no effect on the correctness of the algorithm as long as it does not belong to the integer programming formulation.

If the default pool concept is replaced by an application specific pool concept, the user of the framework must make sure that there is at least one variable pool and one constraint pool and these pools are embedded in a class derived from the class `ABA_MASTER`.

With this concept ABACUS provides a high flexibility: An easy to use default implementation that can be changed by the redefinition of virtual functions and the application of non-default function arguments. All classes involved in this pool concept are designed as generic classes such that they can be used both for variables and constraints.

Linear Programs. Since ABACUS is a framework for the implementation of linear programming based branch-and-bound algorithms, it is obvious that the solution of linear programs plays a central role, and we require a class concept for the representation of linear programs. Moreover, linear programs might not only be used for the solution of LP-relaxations in the subproblems, but they can also be used for other purposes, e.g., within heuristics for the determination of good feasible solutions in mixed integer programming.

Therefore, ABACUS provides two basic interfaces for a linear program. The first one is in a very general form for linear programs defined by a constraint matrix stored in some sparse format. The second one is designed for the solution of the LP-relaxations in a subproblem. The main differences to the first interface are that the constraint matrix is stored in the abstract variable/constraint format instead of the column/row format and that fixed and set variables are eliminated.

Another important design criterion is that the solution of the linear programs should be independent from the used LP-solver, and plugging in a new LP-solver should be simple.

The Basic Interface. The result of these requirements is the class hierarchy of Fig. 16 The class `ABA_LP` is an abstract base class providing the public functions that are usually expected: initialization, optimization, addition of rows and columns, deletion of rows and columns, access to the problem data, the solution, the slack variables, the reduced costs, and the dual variables. These functions do some minor bookkeeping and call a pure virtual function having the same name but starting with an underscore (e.g, `optimize()` calls `_optimize()`). These functions starting with an underscore are exactly the functions that have to be implemented by a LP-solver.

The LP-Solvers Cplex, XPress, and SoPlex. The classes `ABA_LPSUBCPLEX`, `ABA_LPSUBXPRESS`, and `ABA_LPSUBSOPLEX` implement these solver specific functions for the LP-solvers Cplex, XPress and SoPlex. If a linear program should be solved with Cplex, an object of the class `ABA_LPSUBCPLEX` is instantiated. Only public members that are inherited from the class `ABA_LP` are used, except the constructors. Using another LP-solver means only replacing the name Cplex by its name in the instantiation after a similar class for this solver as the class `ABA_LPSUBCPLEX` has been implemented.

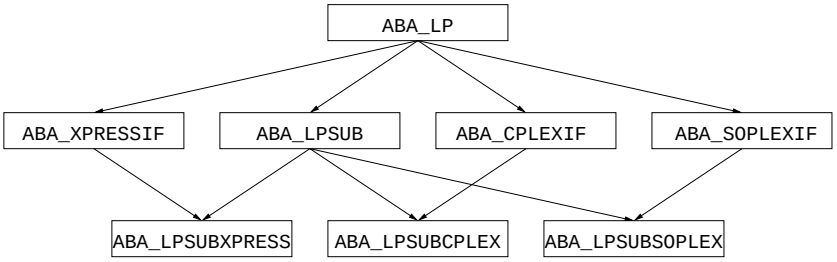


Fig. 16. Inheritance structure of Linear Programming classes.

Linear Programming Relaxations. The most important linear programs within this system are the LP-relaxations that arise during the optimization of the subproblems. However, the active constraints and variables of a subproblem are not stored in the format required by the class `ABA_LP`. Therefore, we have to implement a transformation from the variable/constraint format to the column/row format. This is done in the virtual functions `genRow()` and `genColumn()` of `ABA_CONSTRAINT` and `ABA_VARIABLE`.

The transformation is not invoked by the class `ABA_LP` but by an interface class `ABA_LPSUB`. This class works like a preprocessor for the linear programs solved in the subproblem. Fixed and set variables can be eliminated from the linear program submitted to the solver. It depends on the used solution method if all fixed and set variables should be eliminated. If the simplex method is used and a basis is known, then only non-basic fixed and set variables should be eliminated. If the barrier method is used, we can eliminate all fixed and set variables. The encapsulation of the interface between the subproblem and the class `ABA_LP` supports a more flexible adaption of the elimination to other LP-solvers in the future and also enables us to use other LP-preprocessing techniques, e.g., constraint elimination, or changing the bounds of variables under certain conditions, without modifying the variables and constraints in the subproblem. Preprocessing techniques other than elimination of fixed and set variables are currently not implemented.

Solving Linear Programming Relaxations with Cplex, XPress, and SoPlex. The subproblem optimization in the class `ABA_SUB` uses only the public functions of the class `ABA_LPSUB`, which is again an abstract class independent of the used LP-solver. A linear program solving the relaxations within a subproblem with, e.g., the LP-solver Cplex, is defined by the class `ABA_LPSUBCPLEX`, which is derived from the classes `ABA_LPSUB` and `ABA_CPLEXIF`. The class `ABA_LPSUBCPLEX` only implements a constructor that passes the arguments to the base classes. Using a different LP-solver in this context requires the definition of a class equivalent to the class `ABA_LPSUBCPLEX` and a redefinition of the virtual function `ABA_LPSUB *generateLp()`, which is a one-line function allocating an object of the class `ABA_LPSUBCPLEX` and returning a pointer to this object.

Therefore, it is easy to use different LP-solvers for different ABACUS applications and it is also possible to use different LP-solvers in a single ABACUS application. For instance, if there is a very fast method for the solution of the linear programs in the root node of the enumeration tree, but all other linear programs should be solved by Cplex, then only a simple modification of `ABA_SUB::generateLp()` is required.

Auxiliary Classes for branch-and-bound. In this section we are going to discuss the design of some important classes that support the linear programming based branch-and-bound algorithm. These are classes for the management of the open subproblems, for buffering newly generated constraints and variables, and for the implementation of branching rules.

The Set of Open Subproblems. During a branch-and-bound algorithm subproblems are dynamically generated in branching steps and later optimized. Therefore, we require a data structure that stores pointers to all unprocessed subproblems and supports the insertion and the extraction of a subproblem.

One of the important issues in a branch-and-bound algorithm is the enumeration strategy, i.e., which subproblem is extracted from the set of open subproblems for further processing. It would be possible to implement the different classical enumeration strategies, like depth-first search, breadth-first search, or best-first search within this class. But then an application-specific enumeration strategy could not be added in a simple way by a user of ABACUS. Of course, with the help of inheritance and virtual functions a technique similar to the one for the usage of different LP-solvers for the subproblem optimization could be applied. However, there is a much simpler solution for this problem.

In the class `ABA_MASTER` a virtual member function is defined that compares two subproblems according to the selected enumeration strategy and returns -1 if the first subproblem has higher priority, 1 if the second one has higher priority, and 0 if both subproblems have equal priority. Application specific enumeration strategies can be integrated by a redefinition of this virtual function. This comparison function of the associated master is called in order to compare two subproblems within the extraction operation of the class `ABA_OPENSUB`.

The class `ABA_OPENSUB` implements the set of open subproblems as a doubly linked linear list. Each time when another subproblem is required for further processing the complete list is scanned and the best subproblem according to the applied enumeration strategy is extracted. This implementation has the additional advantage, that it is very easy to change the enumeration strategy during the optimization process, e.g., to perform a diving strategy that uses best-first search but performs a limited depth-first search every k iterations. The drawback of this implementation is the linear running time of the extraction of a subproblem. If the set of open subproblems would be implemented as a heap, then the insertion and the extraction of a subproblem would require logarithmic time, whereas in the current implementation the insertion requires constant, but the extraction requires linear time. But if the enumeration strategy is changed, the heap must be reinitialized from scratch, which requires linear time.

However, it is typical for linear programming based branch-and-bound algorithm that a lot of work is performed in the subproblem optimization, but the total number of subproblems is comparatively small. The performance analysis of our current applications shows that the running time spent in the management of the set of open subproblems is negligible. Due to the encapsulation of the management of the set of open subproblems in the private part of the class `ABA_OPENSUB`, it will be no problem to change the implementation, as soon as it is required.

ABACUS provides four rather common enumeration strategies per default: best-first search, breadth-first search, depth-first search, and a simple diving strategy performing depth-first search until the first feasible solution is found and continuing afterwards with best-first search.

Buffering Generated Variables and Constraints. Usually, new constraints are generated in the separation phase. However, it is possible that in some applications violated constraints are also generated in other subroutines of the cutting plane algorithm. In particular, if not all constraints of the integer programming formulation are active in the subproblem, a separation routine might have to be called to check the feasibility of the LP-solution. Another example is the maximum cut problem, for which it is rather convenient if new constraints can also be generated while we try to find a better feasible solution after the linear program has been solved. Therefore, it is necessary that constraints can be added by a simple function call from any part of the cutting plane algorithm.

This requirement also holds for variables. For instance, when we perform a special rounding algorithm on a fractional solution during the optimization of the TSP (see [43]), we may detect useful variables that are currently inactive. It should be possible to add such important variables before they may be activated in a later pricing step.

It can happen that too many variables or constraints are generated such that it is not appropriate to add all of them, but only the "best" ones. Measurements for "best" are difficult. For constraints this can be the slack or the distance between the fractional solution and the associated hyperplane, for variables this can be the reduced costs.

Therefore, ABACUS implements a buffer for generated constraints and variables in the generic class `ABA_CUTBUFFER`, that can be used both for variables and constraints. There is one object of this class for buffering variables, the other one for buffering constraints. Constraints and variables that are added during the subproblem optimization are not added directly to the linear program and the active sets of constraints and variables, but are added to these buffers. The size of the buffers can be controlled by parameters. At the beginning of the next iteration items out of the buffers are added to the active constraint and variable sets and the buffers are emptied. An item added to a buffer can receive an optional rank given by a floating point number. If all items in a buffer have a rank, then the items with maximal rank are added. As the rank is only specified by a floating point number, different measurements for the quality of the constraints or variables can be applied. The number of added constraints and variables can be controlled again by parameters.

If an item is discarded during the selection of the constraints and variables from the buffers, then usually it is also removed from the pool and deleted. However, it may happen that these items should be kept in the pool in order to regenerate them in later iterations. Therefore, it is possible to set an additional flag while adding a constraint or variable to the buffer that prevents it from being removed from the pool if it is not added. Constraints or variables that are regenerated from a pool receive this flag automatically.

Another benefit of this buffering technique is that adding a constraint or variable does not change immediately the current linear program and the active sets. The update of these data structures is performed at the beginning of the cutting plane or column generation algorithm before the linear program is solved. Hence, this buffering method together with the buffering of removed constraints and variables relieves us also from some nasty bookkeeping.

Branching. It should be possible that in a framework for linear programming based branch-and-bound algorithms many different branching strategies can be embedded. Standard branching strategies are branching on a binary variable by setting it to 0 or 1, changing the bounds of an integer variable, or splitting the solution space by a hyperplane such that in one subproblem $a^T x \leq \beta$ and in the other subproblem $a^T x \geq \beta$ must hold. A straightforward generalization is that instead of one variable or one hyperplane we use k variables or k hyperplanes, which results in a 2^k-nary instead of a binary enumeration tree.

Another branching strategy is branching on a set of equations $a_1^T x = \beta_1, \ldots, a_l^T x = \beta_l$. Here, l new subproblems are generated by adding one equation to the constraint system of the father in each case. Of course, as for any branching strategy, the complete set of feasible solutions of the father must be covered by the sets of feasible solutions of the generated subproblems.

It is obvious that we require on the one hand a rather general concept for branching that does not only cover all mentioned strategies, but should also be extendible to "unknown" methods.

On the other hand it should be simple for a user of the framework to adapt an existing branching strategy like branching on a single variable by adding a new branching variable selection strategy.

Again, an abstract class is the basis for a general branching scheme, and overloading a virtual function provides a simple method to change the branching strategy. ABACUS uses the concept of branching rules. A branching rule defines the modifications of a subproblem for the generation of a son. In a branching step as many rules as new subproblems are instantiated. The constructor of a new subproblem receives a branching rule. When the optimization of a subproblem starts, the subproblem makes a copy of the member data defining its father, i.e., the active constraints and variables, and makes the modifications according to its branching rule.

The abstract base class for different branching rules is the class `ABA_BRANCHRULE`, which declares a pure virtual function modifying the subproblem according to the branching rule. We have to declare this function in the class `ABA_BRANCHRULE` instead of the class `ABA_SUB` because otherwise adding a new branch-rule would require a modification of the class `ABA_SUB`.

ABACUS derives from the class ABA_BRANCHRULE classes for branching by setting a binary variable (class ABA_SETBRANCHRULE), for branching by changing the upper and lower bound of an integer variable (class ABA_BOUNDBRANCHRULE), for branching by setting an integer variable to a value (class ABA_VALBRANCHRULE, and branching by adding a new constraint (class ABA_CONBRANCHRULE).

This concept of branching rules should allow almost every branching scheme. Especially, it is independent of the number of generated sons of a subproblem. Further branching rules can be implemented by deriving new classes from the class ABA_BRANCHRULE and defining the pure virtual function for the corresponding modification of the subproblem.

In order to simplify changing the branching strategy, ABACUS implements the generation of branching rules in a hierarchy of virtual functions of the class ABA_SUB. By default, the branching rules are generated by branching on a single variable. If a different branching strategy is implemented a virtual function must be redefined in a class derived from the class ABA_SUB.

6 Exercise: Implementation of a Simple TSP Solver

The exercise that we gave at the end of our teaching unit at the school consisted of the implementation of an ABACUS program that solves the TSP exactly, given a routine for reading the problem data in TSPLIB format [61] and the minimum capacity cut solver of Jünger et al. [42]. Much to our satisfaction the participants took up the challenge with enthusiasm, and, finally, we had 12 correct TSP solvers. A sample solution by Stefan Thienel is available as a technical report [65] that can be found on the web at:

```
http://www.informatik.uni-koeln.de/ls_juenger/
            projects/abacus/abacus_tutorial.ps.gz
```

References

1. ABACUS 2.3: User's guide and reference manual. Oreas GmbH, 1999.
2. Applegate, D., Bixby, R., Chvátal, V., Cook, W.: TSP-solver "Concorde". at http://www.keck.caam.rice.edu/concorde.html, 1999.
3. Applegate, D., Cook, W.: Solving large-scale matching problems. Network Flows and Matching, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 12, Johnson, D.S., McGeoch, C.C., eds., pp. 557–576, 1993.
4. Balas, E., Ceria, S., Cornuéjols, G., Natraj, N.R.: Gomory cuts revisited. Operations Research Letters **19**, 1–10, 1996.
5. Balas, E., Toth, P.: Branch and bound methods, in E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys (eds.), The traveling salesman Problem. John Wiley & Sons, Chichester, 361–401, 1985.
6. Balinski, M. L.: Mathematical Programming: Journal, Society, Recollections. in: Lenstra et al. (eds.) History of mathematical programming, CWI North-Holland, 5–18, 1991.

7. Barahona, F.: The max-cut problem on graphs not contractible to K_5. Operations Research Letters **2**, 107–111, 1983.
8. Barahona, F.: On cuts and matchings in planar graphs. Mathematical Programming **70**, 53–68, 1993.
9. Barahona, F., Grötschel, M., Jünger, M., Reinelt, G.: An application of combinatorial optimization to statistical physics and circuit layout design. Operations Research **36**, 493–513, 1988.
10. Barahona, F., Mahjoub, A.R.: On the cut polytope. Mathematical Programming **36**, 157–173, 1986.
11. Boros, E., Hammer, P.L. Cut-polytopes, boolean quadric polytopes and nonnegative quadratic pseudo-boolean functions. Mathematics of Operations Research **18**, 245–253, 1993.
12. Caprara, A., Fischetti, M.: Branch-and-cut algorithms. in: Dell'Amico, M. et al. (eds.), Annotated bibliographies in combinatorial optimization, Wiley, 45–64, 1997.
13. Charnes, A., Cooper, W.W., Mellon, B.: Blending aviation gasoline – A study of programming interdependent activities in an integrated oil company. Econometrica **20**, 1952.
14. Clarke, G., Wright, J.W.: Scheduling of vehicles from a central depot to a number of delivery points. Operations Research **12**, 568–581, 1964.
15. Christof, T., Reinelt, G.: Algorithmic aspects of using small instance relaxations in parallel branch and cut. Technical Report, to appear in Algorithmica, 2001.
16. Christof, T.: Low-Dimensional 0/1-Polytopes and Branch-and-Cut in Combinatorial Optimization. Doctoral Thesis, Universität Heidelberg, 1997.
17. Cplex 6.6, Using the Cplex callable library and mixed integer library. Cplex Optimization Inc., 2000.
18. Clochard, J.M., Naddef, D.: Using path inequalities in an branch and cut code for the symmetric traveling salesman problem. in: G. Rinaldi and L. Wolsey (eds.), Proceedings of the Third IPCO Conference, 291–311, 1993.
19. Cordier, C., Marchand, H., Laundy, R., Wolsey, L.A.: bc-opt: a branch-and-cut code for mixed integer programs. Discussion Paper CORE-9778, Université Catholique de Louvain, 1997.
20. Dantzig, G.B.: Linear Programming and Extensions. Princeton University Press, 1963.
21. Dantzig, G.B., Fulkerson, D.R., Johnson, S.M.: Solution of a large scale traveling salesman problem. Operations Research **2**, 393–410, 1954.
22. Dantzig, G.B., Wolfe, P.: The Decomposition Algorithm for Linear Programs. Econometrica, 29:767–778, 1961.
23. Deza, M., Laurent, M.: Cut Polyhedra and Metrics. LIENS - Ecole Normale Supérieure, 1996.
24. De Simone, C., Rinaldi, G.: A cutting plane algorithm for the max-cut problem. Optimization Methods and Software **3**, 195–214, 1994.
25. De Simone, C., Diehl, M., Jünger, M., Mutzel, P., Reinelt, G., Rinaldi, G.: Exact ground states in spin glasses: New experimental results with a branch-and-cut algorithm. Journal of Statistical Physics **80**, 487–496, 1995.
26. De Simone, C., Diehl, M., Jünger, M., Mutzel, P., Reinelt, G., Rinaldi, G.: Exact ground states of 2-dimensional $\pm J$ Ising spin glasses. Journal of Statistical Physics **84**, 1363–1371, 1996.
27. Dijkstra, E.W.: A note on two problems in connection with graphs. Numerische Mathematik **1**, 269–271, 1959.

28. Edmonds, J.: Paths, trees and flowers. Canadian Journal of Mathematics **17**, 449–467, 1965.
29. Elf, M., Jünger, M., Rinaldi, G.: Minimizing Breaks by Maximizing Cuts. Technical Report (zpr2001-409), Universität zu Köln, 2001
30. Felici, G., Gentile, C., Rinaldi, G.: Solving Large MIP Models in Supply Chain Management by Branch & Cut. IASI-CNR Research Report n. 522, 2000.
31. Gilmore, P.C., Gomory, R.E.: A linear programming approach to the cutting stock problem. Operations Research **9**, 849–859, 1961.
32. Gomory, R.E.: Outline of an algorithm for integer solutions to linear programs. Bulletin of the American Mathematical Society **64**, 275–278, 1958.
33. Gomory, R.E., Hu, T.C.: Multiterminal network flows. SIAM Journal **9**, 551–570, 1961.
34. Grötschel, M., Jünger, M., Reinelt, G.: A cutting plane algorithm for the linear ordering problem. Operations Research **32**, 1195–1220, 1984.
35. Grötschel, M., Martin, A., Weismantel, R.: Packing Steiner trees: A cutting plane algorithm and computational results. Mathematical Programming **72**, 125-145, 1996.
36. Grötschel, M., Holland, O.: Solving matching problems with linear Programming. Mathematical Programming **33**, 243–259, 1985
37. Grötschel, M., Lovász, L., Schrijver, A.: The ellipsoid method and its consequences in combinatorial optimization. Combinatorica **4**, 169–197, 1981.
38. Henz, M.: Scheduling a major college basketball conference—revisited. Technical Note, School of Computing, National University of Singapore, 1999
39. Jünger, M., Reinelt, G., Rinaldi, G.: The Traveling Salesman Problem. in: Ball, M. et al. (eds.) Network Models, Handbook on operations research and management sciences, Vol. 7, North Holland, Amsterdam, pp. 225–330, 1995.
40. Jünger, M., Reinelt, G., Rinaldi, G.: Lifting and separation procedures for the cut polytope. Technical Report, Universität zu Köln, in preparation.
41. Jünger, M., Reinelt, G., Thienel, S.: Practical problem solving with cutting plane algorithms in combinatorial optimization. in: Cook, W. et al. (eds.) Combinatorial Optimization, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, pp. 111–152, 1995.
42. Jünger, G., Rinaldi, G., Thienel, S.: Practical performance of efficient minimum cut algorithms. Algorithmica **26**, 172–195, 2000.
43. Jünger, M., Reinelt, G., Thienel, S.: Provably good solutions for the traveling salesman problem. Zeitschrift für Operations Research **40**, 183–217, 1994.
44. Jünger, M., Thienel, S.: The ABACUS System for Branch and Cut and Price Algorithms in Integer Programming and Combinatorial Optimization. Software Practice and Experience **30**, 1325–1352, 2000.
45. Karp, R. M., Papadimitriou, C. H.: On linear characterizations of combinatorial optimization problems. SIAM Journal on Computing **11**, 620–632, 1982.
46. Kruskal, J. B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society **7**, 48–50, 1956.
47. Land, A. H., Doig, A. G.: An automatic method for solving discrete programming problems. Econometrica **28**, 493–520, 1960.
48. Lin, S., Kernighan, B.W.: An effective heuristic algorithm for the traveling salesman problem. Operations Research **21**, 498–516, 1973.
49. Naddef, D., Rinaldi, G.: The graphical relaxation: A new framework for the symmetric traveling salesman polytope. Mathematical Programming **58**, 53–88, 1993.

50. Naddef, D., Thienel, S.: Efficient Separation Routines for the Symmetric Traveling Salesman – Problem I: General Tools and Comb Separation. Technical Report 99-376, Institut für Informatik, Universität zu Köln, 1999, to appear in Mathematical Programming.
51. Naddef, D., Thienel, S.: Efficient Separation Routines for the Symmetric Traveling Salesman – Problem II: Separating multi Handle Inequalities. Technical Report 99-377, Institut für Informatik, Universität zu Köln, 1999, to appear in Mathematical Programming.
52. Naddef, D.: The Traveling Salesman Problem and its Variations, Chapter 2, Gutin, G. and Punnen, A. (eds.), Kluwer Academic Press, to appear.
53. Nemhauser, G.L., Savelsbergh, M.W.P., Sigismondi, G.C.: MINTO, a Mixed Integer Optimizer. Operations Research Letters **15**, 47–58, 1994.
54. Nemhauser, G. L., Trick, M. A.: Scheduling a major college basketball conference. Operations Research **46**, 1–8, 1998.
55. Padberg, M. W., Rinaldi, G.: Optimization of a 532 City Symmetric Traveling Salesman Problem by Branch and Cut. Operations Research Letters **6**, 1–7, 1987.
56. Padberg, M. W., Rinaldi, G.: A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. SIAM Review **33**, 60–100, 1991.
57. Padberg, M.W., Rinaldi, G.: An efficient algorithm for the minimum capacity cut problem. Mathematical Programming **47**, 19–36, 1990.
58. Padberg, M.W., Rinaldi, G.: Facet Identification for the Symmetric Traveling Salesman Polytope. Mathematical Programming **47**, 219–257, 1990.
59. Poljak, S., Tuza, Z.: The max-cut problem—A survey, Institute of Mathematics, Academia Sinica, 1994.
60. Reinelt, G., Fast heuristics for large geometric traveling salesman problems. ORSA Journal on Computing **4**, 206-217, 1992.
61. Reinelt, G.: TSPLIB — A Traveling Salesman Problem Library. ORSA Journal On Computing **3**, 376–384, 1991.
62. Régin, J.-C.: Minimization of the number of breaks in sports scheduling problems using constraint programming. Talk presented at DIMACS Workshop on Constraint Programming and Large Scale Discrete Optimization, Sep. 14–19, 1998.
63. Régin, J.-C.: Modelling with constraint programming. Talk presented at Dagstuhl Seminar on Constraint Programming and Integer Programming, Jan. 17–21, 2000.
64. Thienel, S.: ABACUS – A Branch And CUt System. Doctoral Thesis, Universität zu Köln, 1995.
65. Thienel, S.: A Simple TSP-Solver: An ABACUS Tutorial. Technical Report 96.245, Universität zu Köln, 1996.
66. Schreuder, J. A. M.: Combinatorial aspects of construction of competition Dutch Professional Football Leagues. Discrete Applied Mathematics **35**, 301–312, 1992.
67. Trick, M. A.: A schedule-then-break approach to sports timetabling. Proceedings of PATAT 2000, to appear.
68. Vance, P.H., Barnhart, C., Johnson, E.L., Nemhauser, G.L.: Solving Binary Cutting Stock Problems by Column Generation and Branch-and-Bound. Computational Optimization and Applications **3**, 111—130, 1994.
69. XPRESS 12, Reference Manual: XPRESS-MP Optimizer Subroutine Library XOSL, Dash Associates, 2000.

Branch, Cut, and Price: Sequential and Parallel*

Laszlo Ladányi[1], Ted K. Ralphs[2], and Leslie E. Trotter, Jr.[3]

[1] Department of Mathematical Sciences, IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, ladanyi@us.ibm.com
[2] Department of Industrial and Manufacturing Systems Engineering,
Lehigh University, Bethlehem, PA 18017,
tkralphs@lehigh.edu, http://www.lehigh.edu/~tkr2
[3] School of OR&IE, Cornell University, Ithaca, NY 14853,
ltrotter@cs.cornell.edu

Abstract. Branch, cut, and price (BCP) is an LP-based branch and bound technique for solving large-scale discrete optimization problems (DOPs). In BCP, both cuts and variables can be generated dynamically throughout the search tree. The ability to handle constantly changing sets of cuts and variables allows these algorithms to undertake the solution of very large-scale DOPs; however, it also leads to interesting implementational challenges. These lecture notes, based on our experience over the last six years with implementing a generic framework for BCP called SYMPHONY (Single- or Multi-Process Optimization over Networks), address these challenges. They are an attempt to summarize some of what we and others have learned about implementing BCP, both sequential and parallel, and to provide a useful reference for those who wish to use BCP techniques in their own research.
SYMPHONY, the software from which we have drawn most of our experience, is a powerful, state-of-the-art library that implements the generic framework of a BCP algorithm. The library's modular design makes it easy to use in a variety of problem settings and on a variety of hardware platforms. All library subroutines are generic—their implementation does not depend on the problem-setting. To develop a full-scale BCP algorithm, the user has only to specify a few problem-specific methods such as cut generation. The vast majority of the computation takes place within a "black box," of which the user need have no knowledge. Within the black box, SYMPHONY performs all the normal functions of branch and cut—tree management, LP solution, cut pool management, as well as inter-process communication (if parallelism is used). Source code and documentation for SYMPHONY are available at http://branchandcut.org/SYMPHONY.

1 Introduction

Since the inception of optimization as a recognized field of study in mathematics, researchers have been both intrigued and stymied by the difficulty of solving

* This research was partially supported by NSF Grant DMS-9527124 and Texas ATP Grant 97-3604-010

M. Jünger and D. Naddef (Eds.): Computat. Comb. Optimization, LNCS 2241, pp. 223–260, 2001.
© Springer-Verlag Berlin Heidelberg 2001

many of the most interesting classes of discrete optimization problems. Even combinatorial problems, though conceptually easy to model as integer programs, have long remained challenging to solve in practice. The last two decades have seen tremendous progress in our ability to solve large-scale discrete optimization problems. These advances have culminated in the approach that we now call *branch and cut*, a technique (see [45,72,50]) which brings the computational tools of branch and bound algorithms together with the theoretical tools of polyhedral combinatorics. In 1998, Applegate, Bixby, Chvátal, and Cook used this technique to solve a *Traveling Salesman Problem* instance with 13,509 cities, a full order of magnitude larger than what had been possible just a decade earlier [2] and two orders of magnitude larger than the largest problem that had been solved up until 1978. This feat becomes even more impressive when one realizes that the number of variables in the standard formulation for this problem is approximately the *square* of the number of cities. Hence, we are talking about solving a problem with roughly *100 million variables*.

There are several reasons for this impressive progress. Perhaps the most important is the dramatic increase in available computing power over the last decade, both in terms of processor speed and memory. This increase in the power of hardware has subsequently facilitated the development of increasingly sophisticated software for optimization, built on a wealth of theoretical results. As software development has become a central theme of optimization research efforts, many theoretical results have been "re-discovered" in light of their new-found computational importance. Finally, the use of parallel computing has allowed researchers to further leverage their gains.

Because of the rapidly increasing sophistication of computational techniques, one of the main difficulties faced by researchers who wish to apply these techniques is the level of effort required to develop an efficient implementation. The inherent need for incorporating problem-dependent methods (most notably for dynamic generation of variables and constraints) has typically required the time-consuming development of custom implementations. Around 1993, this led to the development by two independent research groups of software libraries aimed at providing a generic framework that users could easily customize for use in a particular problem setting. One of these groups, headed by Jünger and Thienel, eventually produced ABACUS (A Branch And CUt System) [52] [⟶ Elf/Gutwenger/Jünger/Rinaldi], while the other, headed by the authors and Ladányi, produced what was then known as COMPSys (Combinatorial Optimization Multi-processing System). After several revisions to enable more broad functionality, COMPSys became SYMPHONY (Single- or Multi-Process Optimization over Networks) [78,76]. A version of SYMPHONY, which we will call COIN/BCP, has also been produced at IBM under the COIN-OR project [27]. The COIN/BCP package takes substantially the same approach and has the same functionality as SYMPHONY, but has extended SYMPHONY's capabilities in some areas, as we will point out.

These lecture notes are based on our experience over the last six years with implementing the SYMPHONY framework and using it to solve several classical combinatorial optimization problems. At times, we will also draw on our

experience with the COIN/BCP framework mentioned earlier. What follows is intended to summarize some of what we and others have learned about implementing BCP algorithms and to provide a concise reference for those who wish to use branch and cut in their own research.

2 Related Work

The past decade has witnessed development of numerous software packages for discrete optimization, most of them based on the techniques of branch, cut, and price. The packages fell into two main categories—those based on general-purpose algorithms for solving mixed integer programs (MIPs) without the use of special structure [⟶ Martin] and those facilitating the use of special structure by interfacing with user-supplied, problem-specific subroutines. We will call packages in this second category *frameworks*. There have also been numerous special-purpose codes developed for use in particular problem settings.

Of the two categories, MIP solvers are the most common. Among the many offerings in this category are MINTO [70], MIPO [10], bc-opt [26], and SIP [⟶ Martin] [67]. Generic frameworks, on the other hand, are far less numerous. The three frameworks we have already mentioned (SYMPHONY, ABACUS, and COIN/BCP) are the most full-featured packages available. Several others, such as MINTO, originated as MIP solvers but have the capability of utilizing problem-specific subroutines. CONCORDE [2,3], a package for solving the *Traveling Salesman Problem* (TSP), also deserves mention as the most sophisticated special-purpose code developed to date [⟶ Applegate/Bixby/Chvátal/Cook].

Other related software includes several frameworks for implementing parallel branch and bound. Frameworks for general parallel branch and bound include PUBB [85], BoB [16], PPBB-Lib [87], and PICO [35]. PARINO [65] and FATCOP [21] are parallel MIP solvers.

3 Organization of the Manuscript

In Sect. 4, we briefly describe branch, cut, and price for those readers requiring a review of the basic methodology. In Sect. 5, we describe the overall design of SYMPHONY without reference to implementational details and with only passing reference to parallelism. In Sect. 6, we then move on to discuss details of the implementation. In Sect. 7, we touch on issues involved in parallelizing BCP. Finally, in Sect. 8 and Sect. 9, we discuss our computational experience, with both sequential and parallel versions of the code. In these sections, we describe the implementation of solvers for two combinatorial optimization models, the *Vehicle Routing Problem* and the *Set Partitioning Problem*. We point out and explain those features and parameters that have been the most important. We also address the effectiveness of parallelism.

4 Introduction to Branch, Cut, and Price

In the remainder of this document, we discuss the application of BCP algorithms to the solution of *discrete optimization problems*. A discrete optimization problem (DOP) can be broadly defined as that of choosing from a finite set S an *optimal* element s^* that minimizes some given *objective function* $f : S \to \mathbf{R}$ ($\mathbf{R}$ will denote the set of all real numbers, and $\mathbf{Z}$ the set of all integers). DOPs arise in many important applications such as planning, scheduling, logistics, telecommunications, bioengineering, robotics, and design of intelligent agents, among others. Most DOPs are in the complexity class $\mathcal{NP}$-complete, so there is little hope of finding provably efficient algorithms [40]. Nevertheless, intelligent search algorithms, such as LP-based branch and bound (to be described below), have been tremendously successful at tackling these difficult problems [50].

4.1 Branch and Bound

Branch and bound is the broad class of algorithms from which branch, cut, and price has evolved. A branch and bound algorithm uses a divide and conquer strategy to partition the solution space into *subproblems* and then optimizes individually over each subproblem. For instance, let S be the set of solutions to a given DOP, and let $c \in \mathbf{R}^S$ be a vector of costs associated with members of S. Suppose we wish to determine a least cost member of S and we are given $\hat{s} \in S$, a "good" solution determined heuristically. Using branch and bound, we initially examine the entire solution space S. In the *processing* or *bounding* phase, we relax the problem. In so doing, we admit solutions that are not in the feasible set S. Solving this relaxation yields a lower bound on the value of an optimal solution. If the solution to this relaxation is a member of S or has cost equal to that of $\hat{s}$, then we are done—either the new solution or $\hat{s}$, respectively, is optimal. Otherwise, we identify n subsets $S_1, \ldots, S_n$ of S, such that $\bigcup_{i=1}^{n} S_i = S$. Each of these subsets is called a *subproblem*; $S_1, \ldots, S_n$ are sometimes called the *children* of S. We add the children of S to the list of *candidate subproblems* (those which await processing). This is called *branching*.

To continue the algorithm, we select one of the candidate subproblems and process it. There are four possible results. If we find a feasible solution better than $\hat{s}$, then we replace $\hat{s}$ with the new solution and continue. We may also find that the subproblem has no solutions, in which case we discard (*prune*) it. Otherwise, we compare the lower bound for the subproblem to our global upper bound, given by the value of the best feasible solution encountered thus far. If it is greater than or equal to our current upper bound, then we may again prune the subproblem. Finally, if we cannot prune the subproblem, we are forced to branch and add the children of this subproblem to the list of active candidates. We continue in this way until the list of candidate subproblems is empty, at which point our current best solution is, in fact, optimal.

4.2 Branch, Cut, and Price

In many applications, the bounding operation is accomplished using the tools of linear programming (LP), a technique described in full generality, e.g., by Hoffman and Padberg [50]. This general class of algorithms is known as *LP-based branch and bound*. Typically, the integrality constraints of an integer programming formulation of the problem are relaxed to obtain an *LP relaxation*, which is then solved to obtain a lower bound for the problem. In [72], Padberg and Rinaldi improved on this basic idea by describing a method of using globally valid inequalities (i.e., inequalities valid for the convex hull of integer solutions) to strengthen the LP relaxation. This technique was called *branch and cut*. Since then, many implementations (including ours) have been fashioned after the ideas they described for solving the Traveling Salesman Problem.

As an example, let a combinatorial optimization problem $CP = (E, \mathcal{F})$ with *ground set* E and *feasible set* $\mathcal{F} \subseteq 2^E$ be given along with a cost function $c \in \mathbf{R}^E$. The incidence vectors corresponding to the members of $\mathcal{F}$ are sometimes specified as the set of all incidence vectors obeying a (relatively) small set of inequalities. These inequalities are typically the ones used in the initial LP relaxation. Now let $\mathcal{P}$ be the convex hull of incidence vectors of members of $\mathcal{F}$. Then we know by Weyl's Theorem (see [71]) that there exists a finite set $\mathcal{L}$ of inequalities valid for $\mathcal{P}$ such that

$$\mathcal{P} = \{x \in \mathbf{R}^n : ax \leq \beta \ \ \forall\, (a, \beta) \in \mathcal{L}\} = \{x \in \mathbf{R}^n : Ax \leq b\}. \tag{1}$$

The inequalities in $\mathcal{L}$ are the potential *constraints*, or *cutting planes*, to be added to the relaxation as needed. Unfortunately, it is usually difficult, if not impossible, to enumerate all of the inequalities in $\mathcal{L}$, else we could simply solve the problem using linear programming. Instead, they are defined implicitly and we use separation algorithms and heuristics to generate these inequalities when they are violated. In Fig. 1, we describe more precisely how the bounding operation is carried out in a branch and cut algorithm for combinatorial optimization.

Once we have failed to either prune the current subproblem or separate the current *relaxed solution* from $\mathcal{P}$, we are forced to branch. The branching operation is usually accomplished by specifying a set of hyperplanes which divide the current subproblem in such a way that the current solution is not feasible for the LP relaxation of any of the new subproblems. For example, in a combinatorial optimization problem, branching could be accomplished simply by fixing a variable whose current value is fractional to 0 in one branch and 1 in the other. The procedure is described more formally in Fig. 2. Figure 3 gives a high level description of the generic branch and cut algorithm.

As with constraints, the columns of A can also be defined implicitly if n is large. If column i is not present in the current matrix, then variable x_i is implicitly taken to have value zero. The process of dynamically generating variables is called *pricing* in the jargon of linear programming, but can also be viewed as that of generating constraints for the dual of the current LP relaxation. Hence, LP-based branch and bound algorithms in which the variables are generated dynamically are known as *branch and price* algorithms. In [15], Barnhart et al. provide a thorough review of these methods.

Bounding Operation
Input: A subproblem $\mathcal{S}$, described in terms of a "small" set of inequalities $\mathcal{L}'$ such that $\mathcal{S} = \{x^s : s \in \mathcal{F} \text{ and } ax^s \leq \beta \ \forall \ (a, \beta) \in \mathcal{L}'\}$ and α, an upper bound on the global optimal value.
Output: Either (1) an optimal solution $s^* \in \mathcal{S}$ to the subproblem, (2) a lower bound on the optimal value of the subproblem and the corresponding relaxed solution $\hat{x}$, or (3) a message `pruned` indicating that the subproblem should not be considered further.
Step 1. Set $\mathcal{C} \leftarrow \mathcal{L}'$.
Step 2. If the LP $\min\{cx : ax \leq \beta \ \forall \ (a, \beta) \in \mathcal{C}\}$ is infeasible, then STOP and output `pruned`. This subproblem has no feasible solutions.
Step 3. Otherwise, consider the LP solution $\hat{x}$. If $c\hat{x} < \alpha$, then go to Step 4. Otherwise, STOP and output `pruned`. This subproblem cannot produce a solution of value better than α.
Step 4. If $\hat{x}$ is the incidence vector of some $\hat{s} \in \mathcal{S}$, then $\hat{s}$ is the optimal solution to this subproblem. STOP and output $\hat{s}$ as s^*.
Step 5. Otherwise, apply separation algorithms and heuristics to $\hat{x}$ to obtain a set of violated inequalities $\mathcal{C}'$. If $\mathcal{C}' = \emptyset$, then $c\hat{x}$ is a lower bound on the value of an optimal element of $\mathcal{S}$. STOP and return $\hat{x}$ and the lower bound $c\hat{x}$.
Step 6. Otherwise, set $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$ and go to Step 2.

Fig. 1. Bounding in the branch and cut algorithm for combinatorial optimization

Branching Operation
Input: A subproblem $\mathcal{S}$ and $\hat{x}$, the LP solution yielding the lower bound.
Output: $S_1, \ldots, S_p$ such that $\mathcal{S} = \cup_{i=1}^{p} S_i$.
Step 1. Determine sets $\mathcal{L}_1, \ldots, \mathcal{L}_p$ of inequalities such that $\mathcal{S} = \cup_{i=1}^{n}\{x \in \mathcal{S} : ax \leq \beta \ \forall \ (a, \beta) \in \mathcal{L}_i\}$ and $\hat{x} \notin \cup_{i=1}^{n} S_i$.
Step 2. Set $S_i = \{x \in \mathcal{S} : ax \leq \beta \ \ \forall \ (a, \beta) \in \mathcal{L}_i \cup \mathcal{L}'\}$ where $\mathcal{L}'$ is the set of inequalities used to describe $\mathcal{S}$.

Fig. 2. Branching in the branch and cut algorithm

When both variables and constraints are generated dynamically during LP-based branch and bound, the technique is known as *branch, cut, and price* (BCP). In such a scheme, there is a pleasing symmetry between the treatment of constraints and that of variables. We further examine this symmetry later in these notes. For now, however, it is important to note that while branch, cut, and price does combine ideas from both branch and cut and branch and price (which themselves have many commonalities), combining the two techniques requires much more sophisticated methods than either requires alone. This is an important theme in what follows.

In our descriptions, we will often use the term *search tree*. This term derives from the representation of the list of subproblems as the nodes of a graph in which each subproblem is connected only to its parent and its children. Storing the

Generic Branch and Cut Algorithm
Input: A data array specifying the problem instance.
Output: A global optimal solution s^* to the problem instance.
Step 1. Generate a "good" feasible solution $\hat{s}$ using heuristics. Set $\alpha \leftarrow c(\hat{s})$.
Step 2. Generate the first subproblem $\mathcal{S}^I$ by constructing a small set $\mathcal{L}'$ of inequalities valid for $\mathcal{P}$. Set $B \leftarrow \{\mathcal{S}^I\}$.
Step 3. If $B = \emptyset$, STOP and output $\hat{s}$ as the global optimum s^*. Otherwise, choose some $\mathcal{S} \in B$. Set $B \leftarrow B \setminus \{\mathcal{S}\}$. Apply the bounding procedure to $\mathcal{S}$ (see Fig. 1).
Step 4. If the result of Step 3 is a feasible solution $\overline{s}$, then $c\overline{s} < c\hat{s}$. Set $\hat{s} \leftarrow \overline{s}$ and $\alpha \leftarrow c(\overline{s})$ and go to Step 3. If the subproblem was pruned, go to Step 3. Otherwise, go to Step 5.
Step 5. Perform the branching operation. Add the set of subproblems generated to A and go to Step 3.

Fig. 3. Description of the generic branch and cut algorithm

subproblems in such a form is an important aspect of our global data structures. Since the subproblems correspond to the nodes of this graph, they will sometimes be referred to as *nodes in the search tree* or simply as *nodes*. The *root node* or *root* of the tree is the node representing the initial subproblem.

5 Design of SYMPHONY

In the remainder of these notes, we will illustrate general principles applicable to implementing BCP by drawing on our experience with SYMPHONY. We thus begin with a high-level description of the framework. SYMPHONY was designed with two major goals in mind—ease of use and portability. With respect to ease of use, we aimed for a "black box" design, whereby the user would not be required to know anything about the implementation of the library, but only about the user interface. With respect to portability, we aimed not only for it to be *possible* to use the framework in a wide variety of settings and on a wide variety of hardware, but also for the framework to perform *effectively* in all these settings. Our primary measure of effectiveness was how well the framework would perform in comparison with a problem-specific (or hardware-specific) implementation written "from scratch."

The reader should be mindful of the fact that achieving such design goals involves a number of difficult tradeoffs, which we highlight throughout the rest of this text. For instance, ease of use is quite often at odds with efficiency. In many instances, we had to sacrifice some efficiency in order to make the code easy to work with and to maintain a true "black box" implementation. Maintaining portability across a wide variety of hardware, both sequential and parallel, also required some difficult choices. Sequential and shared-memory platforms demand memory-efficient data structures in order to maintain the very large search trees that can be generated. When moving to distributed platforms, these storage schemes do not scale well to large numbers of processors. This is further discussed in Sect. 7.1.

5.1 An Object-Oriented Approach

As we have already remarked, applying BCP to large-scale problems presents several difficult challenges. First and foremost is designing methods and data structures capable of handling the potentially huge numbers of constraints and variables that need to be accounted for during the solution process. The dynamic nature of the algorithm requires that we must also be able to efficiently move constraints and variables in and out of the *active set* of each search node at any time. A second, closely-related challenge is that of effectively dealing with the very large search trees that can be generated for difficult problem instances. This involves not only the important question of how to store the data, but also how to move it between modules during parallel execution. A final challenge in developing a generic framework, such as SYMPHONY, is to deal with these issues using a problem-independent approach.

Describing a node in the search tree consists of, among other things, specifying which constraints and variables are initially *active* in the subproblem. In fact, the vast majority of the methods in BCP that depend on the model are related to generating, manipulating, and storing the constraints and variables. Hence, SYMPHONY can be considered an object-oriented framework with the central "objects" being the constraints and variables. From the user's perspective, implementing a BCP algorithm using SYMPHONY consists primarily of specifying various properties of objects, such as how they are generated, how they are represented, and how they should be realized within the context of a particular subproblem.

With this approach, we achieved the "black box" structure by separating these problem-specific functions from the rest of the implementation. The internal library interfaces with the user's subroutines through a well-defined Application Program Interface (API) and independently performs all the normal functions of BCP—tree management, LP solution, and cut pool management, as well as inter-process communication (when parallelism is employed). Although there are default options for many of the operations, the user can also assert control over the behavior of the algorithm by overriding the default methods or by manipulating the parameters.

Although we have described our approach as being "object-oriented," we would like to point out that SYMPHONY is implemented in C, not C++. To avoid inefficiencies and enhance the modularity of the code (allowing for easy parallelization), we used a more "function-oriented" approach for the implementation of certain aspects of the framework. For instance, methods used for communicating data between modules are not naturally "object-oriented" because the type of data being communicated is usually not known by the message-passing interface. It is also common that efficiency considerations require that a particular method be performed on a whole set of objects at once rather than on just a single object. Simply invoking the same method sequentially on each of the members of the set can be inefficient. In these cases, it is far better to define a method which operates on the whole set at once. In order to overcome these problems, we have also defined a set of *interface functions*, which are associated with the computational modules (Sect. 5.3).

5.2 Data Structures and Storage

Both the memory required to store the search tree and the time required to process a node are largely dependent on the number of objects (constraints and variables) active in each subproblem. Keeping this active set as small as possible is one of the keys to efficiently implementing BCP. For this reason, we chose data structures that enhance our ability to efficiently move objects in and out of the active set. Allowing sets of constraints and variables to move in and out of the linear programs simultaneously is one of the most significant challenges of BCP. We do this by maintaining an abstract *representation* of each global object that contains information about how to add it to a particular LP relaxation.

In the literature on linear and integer programming, the terms *constraint* and *row* are often used interchangeably. Similarly, *variable* and *column* are often used with the same meaning. In many situations, this is appropriate and does not cause confusion. However, in object-oriented BCP frameworks, such as SYMPHONY or ABACUS [52], a *constraint* and a *row* are *fundamentally different objects*. A *constraint* (also referred to as a *cut*) is a user-defined representation of an abstract object which can only be realized as a row in an LP matrix *with respect to a particular set of active variables*. Similarly, a *variable* is a representation which can only be realized as a column of an LP matrix with respect to a *particular set of constraints*. This distinction between the *representation* and the *realization* of objects is a crucial design element that allows us to effectively address some of the challenges inherent in BCP. In the remainder of this section, we further discuss this distinction and its implications.

Variables. In SYMPHONY, problem variables are *represented* by a unique global index assigned to each variable by the user. This index indicates each variable's position in a "virtual" global list known only to the user. The main requirement of this indexing scheme is that, given an index and a list of active constraints, the user must be able to generate the corresponding column to be added to the matrix. As an example, in problems where the variables correspond to the edges of an underlying graph, the index could be derived from a lexicographic ordering of the edges (when viewed as ordered pairs of nodes).

This indexing scheme provides a very compact representation, as well as a simple and effective means of moving variables in and out of the active set. However, it means that the user must have a priori knowledge of all problem variables and a method for indexing them. For combinatorial models such as the *Traveling Salesman Problem*, this does not present a problem. However, for other models such as airline crew scheduling (discussed below), for instance, the number of columns may not be known in advance. Even if the number of columns is known in advance, a viable indexing scheme may not be evident. Eliminating the indexing requirement by allowing variables to have abstract, user-defined representations (such as we do for constraints, as described in the next section), would allow for more generality, but would also sacrifice some efficiency. A hybrid scheme, allowing the user to have both indexed and *algorithmic* variables (vari-

ables with user-defined representations) has been implemented in COIN/BCP and is also planned for a future version of SYMPHONY.

For efficiency, the problem variables can be divided into two sets, the *core variables* and the *extra variables*. The core variables are active in all subproblems, whereas the extra variables can be freely added and removed. There is no theoretical difference between core variables and extra variables; however, designating a well-chosen set of core variables can significantly increase efficiency. Because they can move in and out of the problem, maintaining extra variables requires additional bookkeeping and computation. If the user has reason to believe a priori that a variable has a high probability of having a non-zero value in some optimal solution to the problem, then that variable should be designated as a core variable. Core variables selection in the case of the Vehicle Routing Problem will be illustrated in Sect. 8.1. For a detailed description of core variable selection in the case of the Traveling Salesman Problem, see [⟶ Elf/Gutwenger/Jünger/Rinaldi]. In addition to the core variables, the user can also designate other variables that should be active in the root subproblem. Often, it is useful to activate these variables in the root, as it is likely they will be priced out quickly anyway. When not using column generation, all variables must be active in the root node.

Constraints. Because the global list of potential constraints is not usually known a priori or is extremely large, constraints cannot generally be represented simply by a user-assigned index. Instead, each constraint is assigned a global index only after it becomes active in some subproblem. It is up to the user, if desired, to designate a compact *representation* for each class of constraints that is to be generated and to implement subroutines for converting from this compact representation to a matrix row, given the list of active variables. For instance, suppose that the set of variables with nonzero coefficients in a particular class of constraints corresponds to the set of edges across a cut in a graph. Instead of storing the index of each variable and its corresponding coefficient explicitly, one can simply store the set of nodes on one side ("shore") of the cut as a bit array. The constraint can then be constructed easily for any particular set of active variables (see Sect. 8.1 for more on this example).

Just as with variables, the constraints are divided into *core constraints* and *extra constraints*. The core constraints are those that are active in every subproblem, whereas the extra constraints can be generated dynamically and are free to enter and leave as appropriate. The set of core constraints must be known and constructed explicitly by the user. Extra constraints, on the other hand, are generated dynamically by the cut generator as they are violated. As with variables, a good set of core constraints can have a significant effect on efficiency.

Note that the user is not *required* to designate a compact representation scheme. Constraints can simply be represented explicitly as matrix rows with respect to the global set of variables. However, designating a compact form can result in large reductions in memory use if the number of variables in the problem is large.

Search Tree. Having described the basics of how objects are represented, we now describe the representation of search tree nodes. Since the core constraints and variables are present in every subproblem, only the indices of the extra constraints and variables are stored in each node's description. A critical aspect of implementing BCP is the maintenance of a complete description of the current basis (assuming a simplex-based LP solver) for each node to allow a warm start to the computation. This basis is either inherited from the parent, computed during strong branching (see Sect. 6.2), or comes from earlier partial processing of the node itself (see Sect. 6.3). Along with the set of active objects, we must also store the identity of the object which was branched upon to generate the node. The branching operation is described in Sect. 6.2.

Because the set of active objects and the status of the basis do not tend to change much from parent to child, all of these data are stored as differences with respect to the parent when that description is smaller than the explicit one. This method of storing the entire tree is highly memory-efficient. The list of nodes that are candidates for processing is stored in a heap ordered by a comparison function defined by the search strategy (see 6.3). This allows efficient generation of the next node to be processed.

5.3 Modular Implementation

SYMPHONY's functions are grouped into five independent computational modules. This modular implementation not only facilitates code maintenance, but also allows easy and highly configurable parallelization. Depending on the computational setting, the modules can be compiled as either (1) a single sequential code, (2) a multi-threaded shared-memory parallel code, or (3) separate processes running over a distributed network. The modules pass data to each other either through shared memory (in the case of sequential computation or shared-memory parallelism) or through a message-passing protocol defined in a separate communications API (in the case of distributed execution). A schematic overview of the modules is presented in Fig. 4. In the remainder of the section, we describe the modularization scheme and the implementation of each module in a sequential environment. We defer serious discussion of issues involved in parallel execution of the code until Sect. 7.

The Master Module. The *master module* includes functions that perform problem initialization and I/O. These functions implement the following tasks:

- Read in the parameters from a data file.
- Read in the data for the problem instance.
- Compute an initial upper bound using heuristics.
- Perform problem preprocessing.
- Initialize the BCP algorithm by sending data for the root node to the *tree manager*.
- Initialize output devices and act as a central repository for output.
- Process requests for problem data.

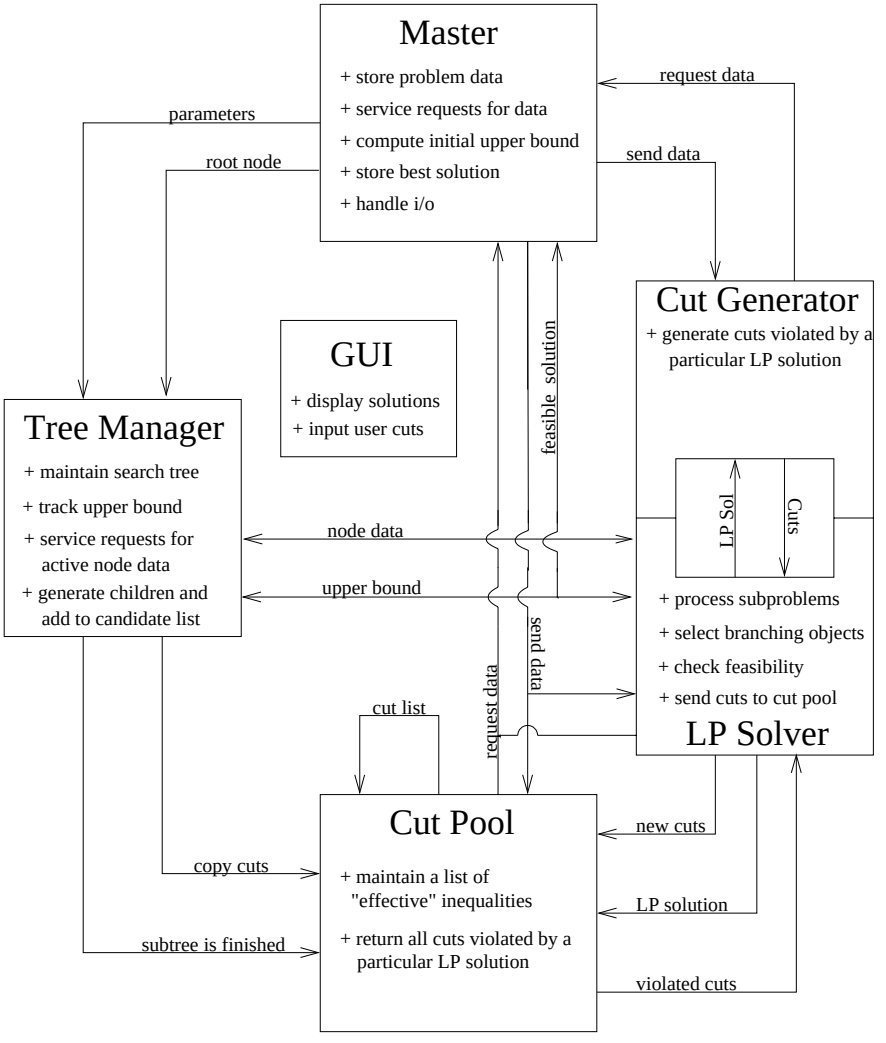


Fig. 4. Schematic overview of the branch, cut, and price algorithm

– Receive new solutions and store the best one.
– Receive the message that the algorithm has finished and print out data.
– Ensure that all modules are still functioning.

The Tree Manager Module. The *tree manager* controls the overall execution of the algorithm. It tracks the status of all modules, as well as that of the search tree, and distributes the subproblems to be processed to the LP module(s). Functions performed by the tree manager module are:

- Receive data for the root node and place it on the list of candidates for processing.
- Receive data for subproblems to be held for later processing.
- Handle requests from linear programming modules to release a subproblem for processing.
- Receive branching object information, set up data structures for the children, and add them to the list of candidate subproblems.
- Keep track of the global upper bound and notify all LP modules when it changes.
- Write current state information out to disk periodically to allow a restart in the event of a system crash.
- Keep track of run data and send it to the master program at termination.

The Linear Programming Module. The *linear programming* (LP) module is the most complex and computationally intensive of the five modules. Its job is to use linear programming to perform the bounding and branching operations. These operations are, of course, central to the performance of the algorithm. Functions performed by the LP module are:

- Inform the tree manager when a new subproblem is needed.
- Receive a subproblem and process it in conjunction with the cut generator and the cut pool.
- Decide which cuts should be sent to the global pool to be made available to other LP modules.
- If necessary, choose a branching object and send its description back to the tree manager.
- Perform the fathoming operation, including generating variables.

The Cut Generator Module. The *cut generator* performs only one function—generating valid inequalities violated by the current LP solution and sending them back to the requesting LP module. Here are the functions performed by the cut generator module:

- Receive an LP solution and attempt to separate it from the convex hull of all solutions.
- Send generated valid inequalities back to the LP solver.
- When finished processing a solution vector, inform the LP not to expect any more cuts in case it is still waiting.

The Cut Pool Module. The concept of a *cut pool* was first suggested by Padberg and Rinaldi [72], and is based on the observation that in BCP, the inequalities which are generated while processing a particular node in the search tree are generally globally valid and potentially useful at other nodes. Since generating these cuts is sometimes a relatively expensive operation, the cut pool maintains a list of the "best" or "strongest" cuts found in the tree thus far for use in processing future subproblems. Hence, the cut pool functions as an auxiliary cut generator. More explicitly, the functions of the cut pool module are:

- Receive cuts generated by other modules and store them.
- Receive an LP solution and return a set of cuts which this solution violates.
- Periodically purge "ineffective" and duplicate cuts to control its size.

5.4 SYMPHONY Overview

Currently, SYMPHONY is a *single-pool* BCP algorithm. The term single-pool refers to the fact that there is a single central list of candidate subproblems to be processed, which is maintained by the tree manager. Most sequential implementations use such a single-pool scheme. However, other schemes may be used in parallel implementations. For a description of various types of parallel branch and bound, see [43].

The master module begins by reading in the parameters and problem data. After initial I/O is completed, subroutines for finding an initial upper bound and constructing the root node are executed. During construction of the root node, the user must designate the initial set of active cuts and variables, after which the data for the root node are sent to the tree manager to initialize the list of candidate nodes. The tree manager in turn sets up the cut pool module(s), the linear programming module(s), and the cut generator module(s). All LP modules are marked as idle. The algorithm is now ready for execution.

In the steady state, the tree manager controls the execution by maintaining the list of candidate subproblems and sending them to the LP modules as they become idle. The LP modules receive nodes from the tree manager, process them, branch (if required), and send back the identity of the chosen branching object to the tree manager, which in turn generates the children and places them on the list of candidates to be processed (see Sect. 6.2 for a description of the branching operation). The preference ordering for processing nodes is a run-time parameter. Typically, the node with the smallest lower bound is chosen to be processed next, since this "best-first" strategy minimizes the overall size of the search tree. However, at times it will be advantageous to *dive* down in the tree. The concepts of *diving* and *search chains*, introduced in Sect. 6.3, extend the basic best-first approach.

We mentioned earlier that cuts and variables can be treated in a somewhat symmetric fashion. However, it should be clear by now that our current implementation favors branch and cut algorithms, where the computational effort spent generating cuts dominates that of generating variables. Our methods of representation also clearly favor such problems. In a future version of the software, we plan to eliminate this bias by adding additional functionality for

handling variable generation and storage. This is the approach already taken in COIN/BCP [27]. For more discussion of the reasons for this bias and the differences between the treatment of cuts and variables, see Sect. 6.2.

6 Details of the Implementation

6.1 The Master Module

The primary functions performed by the master module were listed in Sect. 5.3. If needed, the user must provide a routine to read problem-specific parameters in from a parameter file. Also suggested is a subroutine for upper bounding, though upper bounds can also be provided explicitly. A good initial upper bound can dramatically decrease the solution time by allowing more variable-fixing (see Sect. 6.2 and also [⟶ Elf/Gutwenger/Jünger/Rinaldi]) and earlier pruning of search tree nodes. If no upper bounding subroutine is available, then the two-phase algorithm, in which a good upper bound is found quickly in the first phase using a reduced set of variables, can be useful (see Sect. 6.3 for details). The user's only unavoidable obligation during preprocessing is to specify the list of core variables and, if desired, the list of extra variables that are to be active in the root node. Again, we point out that selecting a good set of core variables can make a marked difference in solution speed, especially when using the two-phase algorithm.

6.2 The Linear Programming Module

The LP module is at the core of the algorithm, as it performs the computationally intensive bounding operations for each subproblem. A schematic diagram of the LP solver loop is presented in Fig. 5. The details of the implementation are discussed in the following sections.

The Linear Programming Engine. SYMPHONY requires the use of a third-party callable library (referred to as the *LP engine* or *LP library*) to solve the LP relaxations once they are formulated. As with the user functions, SYMPHONY communicates with the LP engine through an API that converts SYMPHONY's internal data structures into those of the LP engine. Currently, the framework will only work with advanced, simplex-based LP engines, such as CPLEX [49], since the LP engine must be able to accept an advanced basis, and provide a variety of data to the framework during the solution process. The internal data structures used for maintaining the LP relaxations are similar to those of CPLEX and matrices are stored in the standard column-ordered format.

Managing the LP Relaxation. The majority of the computational effort of BCP is spent solving LPs and hence a major emphasis in the development was to make this process as efficient as possible. Besides using a good LP engine, the

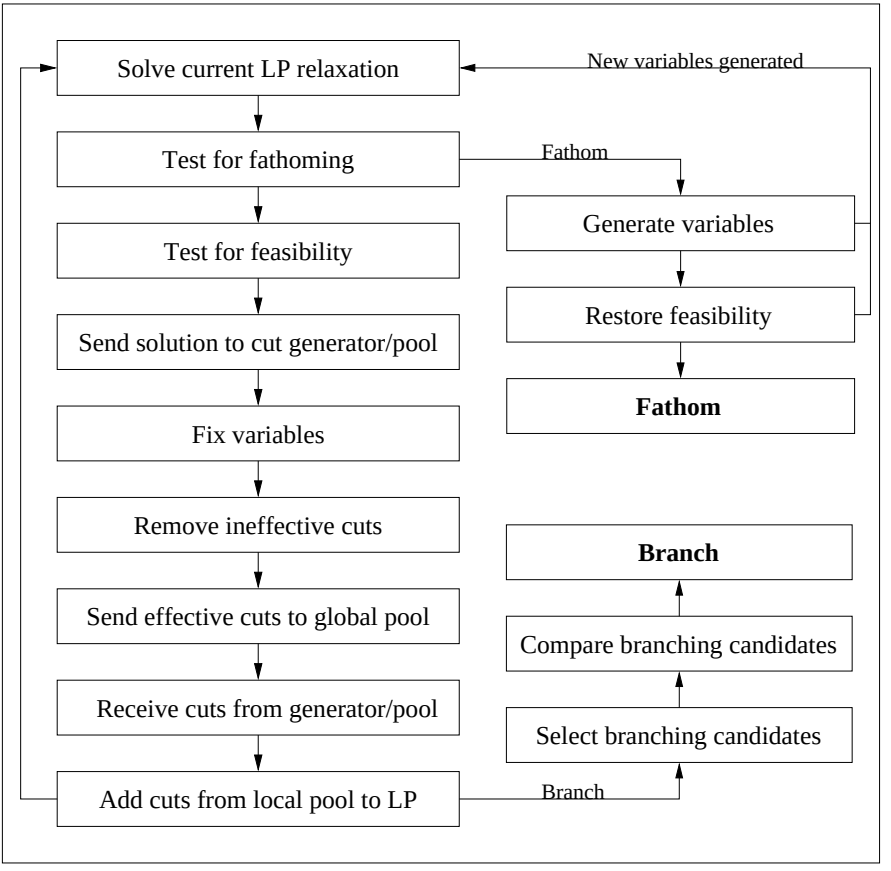


Fig. 5. Overview of the LP solver loop

primary way in which this is done is by controlling the size of each relaxation, both in terms of number of active variables and number of active constraints.

The number of constraints is controlled through use of a local pool and through purging of ineffective constraints. When a cut is generated by the cut generator, it is first sent to the local cut pool. In each iteration, up to a specified number of the strongest cuts (measured by degree of violation) from the local pool are added to the problem. Cuts that are not strong enough to be added to the relaxation are eventually purged from the list. In addition, cuts are purged from the LP itself when they have been deemed ineffective for more than a specified number of iterations, where ineffective is defined as either (1) the corresponding slack variable is positive, (2) the corresponding slack variable is basic, or (3) the dual value corresponding to the row is zero (or very small). Cuts that have remained effective in the LP for a specified number of iterations are

sent to the global pool where they can be used in later search nodes. Cuts that have been purged from the LP can be made active again if they later become violated.

The number of variables (columns) in the relaxation is controlled through *reduced cost fixing* and *dynamic column generation*. Periodically, each active variable is *priced* to see if it can be fixed by reduced cost. That is, the LP reduced cost is examined in an effort to determine whether fixing that variable at one of its bounds would remove improving solutions; if not, the variable is fixed and removed from consideration. For a more detailed description of the conditions for fixing and setting by reduced cost, see [⟶ Elf/Gutwenger/Jünger/Rinaldi]. If the matrix is *full* at the time of the fixing, meaning that all unfixed variables are active, then the fixing is permanent for that subtree. Otherwise, it is temporary and only remains in force until the next time that columns are dynamically generated.

Because SYMPHONY was originally designed for combinatorial problems with relatively small numbers of variables, techniques for performing dynamic column generation are somewhat unrefined. Currently, variables are priced out sequentially by index, which can be costly. To improve the process of pricing variables, we plan to increase the symmetry between our methods for handling variables and those for handling cuts. This includes (1) allowing user-defined, abstract representations for variables, (2) allowing the use of "variable generators" analogous to cut generators, (3) implementing both global and local pools for variables, (4) implementing heuristics that help determine the order in which the indexed variables should be priced, and (5) allowing for methods of simultaneously pricing out large groups of variables. Much of this is already implemented in COIN/BCP.

Because pricing is computationally burdensome, it currently takes place only either (1) before branching (optional), or (2) when a node is about to be pruned (depending on the phase—see the description of the two-phase algorithm in Sect. 6.3). To use dynamic column generation, the user must supply a subroutine which generates the column corresponding to a particular user index, given the list of active constraints in the current relaxation. When column generation occurs, each column not currently active that has not been previously fixed by reduced cost is either priced out immediately, or becomes active in the current relaxation. Only a specified number of columns may enter the problem at a time, so when that limit is reached, column generation ceases. For further discussion of column generation, see Sect. 6.3, where the two-phase algorithm is described.

Since the matrix is stored in compressed form, considerable computation may be needed to add and remove rows and columns. Hence, rows and columns are only physically removed from the problem when there are sufficiently many to make it "worthwhile." Otherwise, deleted rows and columns remain in the matrix but are simply ignored by the computation. Note that because ineffective rows left in the matrix increase the size of the basis unnecessarily, it is usually advisable to adopt an aggressive strategy for row removal.

Branching. Branching takes place whenever either (1) both cut generation and column generation (if performed) have failed; (2) "tailing off" in the objective function value has been detected (see [⟶ Elf/Gutwenger/Jünger/Rinaldi] for a description of tailing off); or (3) the user chooses to force branching. Branching can take place on cuts or variables and can be fully automated or fully controlled by the user, as desired. Branching can result in as many children as the user desires, though two is typical. Once it is decided that branching will occur, the user must either select the list of candidates for *strong branching* (see below for the procedure) or allow SYMPHONY to do so automatically by using one of several built-in strategies, such as branching on the variable whose value is farthest from being integral. The number of candidates may depend on the level of the current node in the tree—it is usually best to expend more effort on branching near the top of the tree.

After the list of candidates is selected, each candidate is *pre-solved*, by performing a specified number of iterations of the dual simplex algorithm in each of the resulting subproblems. Based on the objective function values obtained in each of the potential children, the final branching object is selected, again either by the user or by built-in rule. This procedure of using exploratory LP information in this manner to select a branching candidate is commonly referred to as *strong branching*. When the branching object has been selected, the LP module sends a description of that object to the tree manager, which then creates the children and adds them to the list of candidate nodes. It is then up to the tree manager to specify which node the now-idle LP module should process next. This issue is further discussed below.

6.3 The Tree Manager Module

The tree manager's primary job is to control the execution of the algorithm by deciding which candidate node should be chosen as the next to be processed. This is done using either one of several built-in rules or a user-defined rule. Usually, the goal of the search strategy is to minimize overall running time, but it is sometimes also important to find good feasible solutions early in the search process. In general, there are two ways to decrease running time—either by decreasing the size of the search tree or by decreasing the time needed to process each search tree node.

To minimize the size of the search tree, the strategy is to select consistently that candidate node with the smallest associated lower bound. In theory, this strategy, sometimes called *best-first*, will lead the smallest possible search tree. However, we need to consider the time required to process each search tree node as well. This is affected by both the quality of the current upper bound and by such factors as communication overhead and node set-up costs. When considering these additional factors, it will sometimes be more effective to deviate from the best-first search order. We discuss the importance of such strategies below.

Search Chains and Diving. One reason for not strictly enforcing the search order is because it is somewhat expensive to construct a search node, send it to the LP solver, and set it up for processing. If, after branching, we choose to continue processing one of the children of the current subproblem, we avoid the set-up cost, as well as the cost of communicating the node description of the retained child subproblem back to the tree manager. This is called *diving* and the resulting chain of nodes is called a *search chain*. There are a number of rules for deciding when an LP module should be allowed to dive. One such rule is to look at the number of variables in the current LP solution that have fractional values. When this number is low, there may be a good chance of finding a feasible integer solution quickly by diving. This rule has the advantage of not requiring any global information. We also dive if one of the children is "close" to being the best node, where "close" is defined by a chosen parameter.

In addition to the time saved by avoiding reconstruction of the LP in the child, diving has the advantage of often leading quickly to the discovery of feasible solutions, as discussed above. Good upper bounds not only allow earlier pruning of unpromising search chains, but also should decrease the time needed to process each search tree node by allowing variables to be fixed by reduced cost.

The Two-Phase Algorithm. If no heuristic subroutine is available for generating feasible solutions quickly, then a unique two-phase algorithm can also be invoked. In the two-phase method, the algorithm is first run to completion on a specified set of core variables. Any node that would have been pruned in the first phase is instead sent to a pool of candidates for the second phase. If the set of core variables is small, but well-chosen, this first phase should be finished quickly and should result in a near-optimal solution. In addition, the first phase will produce a list of useful cuts. Using the upper bound and the list of cuts from the first phase, the root node is *repriced*—that is, it is reprocessed with the full set of variables and cuts. The hope is that most or all of the variables not included in the first phase will be priced out of the problem in the new root node. Any variable thus priced out can be eliminated from the problem globally. If we are successful at pricing out all of the inactive variables, we have shown that the solution from the first phase was, in fact, optimal. If not, we must go back and price out the (reduced) set of extra variables in each leaf of the search tree produced during the first phase. We then continue processing any node in which we fail to price out all the variables.

In order to avoid pricing variables in every leaf of the tree, we can *trim the tree* before the start of the second phase. Trimming the tree consists of eliminating the children of any node for which each child has lower bound above the current upper bound. We then reprocess the parent node itself. This is typically more efficient, since there is a high probability that, given the new upper bound and cuts, we will be able to prune the parent node and avoid the task of processing each child individually.

6.4 The Cut Generator Module

To implement the cut generator module, the user must provide a function that accepts an LP solution and returns cuts violated by that solution to the LP module. In parallel configurations, each cut is returned immediately to the LP module, rather than being returned within a group of cuts when the function terminates. This allows the LP to begin adding cuts and re-solving the current relaxation before the cut generator is finished, if desired. Parameters controlling if and when the LP should begin solving the new relaxation before the cut generator is finished can be set by the user.

6.5 The Cut Pool Module

Maintaining and Scanning the Pool. The cut pool's primary job is to receive a solution from an LP module and return cuts from the pool that are violated by it. The cuts are stored along with two pieces of information—the level of the tree on which the cut was generated, known simply as the *level* of the cut, and the number of times it has been checked for violation since the last time it was actually found to be violated, known as the number of *touches*. The number of touches can be used as a simplistic measure of its effectiveness. Since the pool can get quite large, the user can choose to scan only cuts whose number of touches is below a specified threshold and/or cuts that were generated on a level at or above the current one in the tree. The idea behind this second criterion is to try to avoid checking cuts that were not generated "nearby" in the tree, as they are less likely to be effective. Any cut generated at a level in the tree below the level of the current node must have been generated in a different part of the tree. Although this is admittedly a naive method, it has proven reasonably effective in practice.

On the other hand, the user may define a specific measure of quality for each cut to be used instead. For example, the degree of violation is an obvious candidate. This measure of quality must be computed by the user, since the cut pool module has no knowledge of the cut data structures. The quality is recomputed every time the user checks the cut for violation and a running average is used as the global quality measure. The cuts in the pool are periodically sorted by this measure and only the highest quality cuts are checked each time. All duplicate cuts, as well as all cuts whose number of touches exceeds or whose quality falls below specified thresholds, are periodically purged from the pool in order to limit computational effort.

Using Multiple Pools. For several reasons, it may be desirable to have multiple cut pools. When there are multiple cut pools, each pool is initially assigned to a particular node in the search tree. After being assigned to that node, the pool services requests for cuts from that node and all of its descendants until such time as one of its descendants is assigned to another cut pool. After that, it continues to serve all the descendants of its assigned node that are not assigned to other pools.

Initially, the first pool is assigned to the root node. All other pools are unassigned. During execution, when a new node is selected for processing, the tree manager must determine which pool will service the node. The default is to assign the same pool as that of its parent. However, if there is currently an idle pool (either it has never been assigned to any node or all the descendants of its assigned node have been processed or reassigned), then that cut pool can be assigned to the new node. The new pool is initialized with all the cuts currently in the cut pool of the parent node, after which the two pools operate independently on their respective subtrees. When generating cuts, the LP module sends the new cuts to the cut pool assigned to service the node during whose processing the cuts were generated.

The primary motivation behind the idea of multiple cut pools is as follows. First, we want simply to limit the size of each pool as much as possible. By limiting the number of nodes that a cut pool has to service, the number of cuts in the pool will be similarly limited. This not only allows cut storage to be spread over multiple machines, and hence increases the available memory, but at the same time, the efficiency with which the cut pool can be scanned for violated cuts is also increased. A secondary reason for maintaining multiple cut pools is that it allows us to limit the scanning of cuts to only those that were generated in the same subtree. As described above, this helps focus the search and should increase the efficiency and effectiveness of the search. This idea also allows us to generate locally valid cuts, such as the classical Gomory cuts (see [71]).

7 Parallelizing BCP

Because of the clear partitioning of work that occurs when the branching operation generates new subproblems, branch and bound algorithms lend themselves well to parallelization. As a result, there is already a significant body of research on performing branch and bound in parallel environments. We again refer the reader to the survey of parallel branch and bound algorithms by Gendron and Crainic [43], as well as other references such as [35,46,80,57].

In parallel BCP, as in general branch and bound, there are two major sources of parallelism. First, it is clear that any group of subproblems on the current candidate list can be processed simultaneously. Once a subproblem has been added to the list, it can be properly processed before, during, or after the processing of any other subproblem. This is not to say that processing a particular node at a different point in the algorithm won't produce different results—it most certainly will—but the algorithm will terminate correctly in any case. The second major source of parallelism is to parallelize the processing of individual subproblems. For instance, by allowing separation to be performed in parallel with the solution of the linear programs, we can theoretically process a node in little more than the amount of time it takes to perform the more expensive of these two operations. Alternatively, it is also possible to separate over several classes of cuts simultaneously. However, computational experience has shown that savings from parallelizing cut generation are difficult to achieve at best.

Nonetheless, both of these sources of parallelism can be easily exploited using the SYMPHONY framework.

The most straightforward parallel implementation, the one we currently employ, is a master-slave model, in which there is a central manager responsible for partitioning the work and parceling it out to the various slave processes that perform the actual computation. This approach was adopted because it allows memory-efficient data structures for sequential computation and yet is conceptually easy to parallelize. Unfortunately, this approach has limited scalability. We discuss design tradeoffs involving scalability in the next section.

7.1 Scalability

Overview of Scalability. We now digress slightly to discuss the importance of *scalability* in parallel algorithm development. Generally speaking, the scalability of a parallel system (the combination of a parallel algorithm and a parallel architecture) is the degree to which it is capable of efficiently utilizing increased computing resources (usually additional processors). To assess this capability, we compare the speed with which we can solve a particular problem instance in parallel to that with which we could solve it on a single processor. The *sequential running time* (T_0) is used as the basis for comparison and is usually taken to be the running time of the best available sequential algorithm. The *parallel running time* (T_p) is the running time of the parallel algorithm in question and depends on p, the number of processors available. The *speedup* (S_p) is the simply the ratio T_0/T_p and hence also depends on p. Finally, the efficiency (E_p) is the ratio S_p/p of speedup to number of processors.

In general, if the problem size is kept constant, efficiency drops as the number of processors increases—this is a product of the fact that there is a fixed fraction of work that is inherently sequential in nature (reading in the problem data, for example). This *sequential fraction* limits the theoretical maximum speedup (see [1]). However, if the number of processors is kept constant, then efficiency generally *increases* as problem size increases [58,46,47]. This is because the sequential fraction becomes smaller as problem size increases. Thus, we generally define scalability in terms of the rate at which the problem size must be increased with respect to the number of processors in order to maintain a fixed efficiency. For more details, see [57].

Scalability for BCP. In order to maintain high parallel efficiency, it is critical not only to keep each processor busy, but to keep each processor busy with *useful work*. Hence, as in [46], we differentiate between two different notions of load balancing—quantitative load balancing and qualitative load balancing. Quantitative load balancing consists of ensuring that the amount of work allocated to each processor is approximately equal. Qualitative load balancing, on the other hand, consists of ensuring not only that each processor has enough work to do, but also that each processor has *high-quality* work to do.

The use of a single central tree manager has the advantage of making load balancing easy. Whenever a processor runs out of work, the tree manager can simply

issue more. Furthermore, it can easily issue the "best" work that is available at that time, usually the subproblem with the least lower bound. Unfortunately, the central tree manager becomes a computational bottleneck when large numbers of slave processes are employed. The degree to which this occurs is highly dependent on the problem setting. If each search tree node requires significant processing time (and hence the tree is not growing too quickly), then scalability may not be much of an issue. For problems in which quick enumeration of a large search tree is the primary computational approach, scalability will suffer.

This problem has been studied extensively for general branch and bound and various approaches to "decentralization" have been suggested to relieve the bottleneck at the tree manager. However, while these approaches are more scalable, they appear to be inefficient when the numbers of processors is small, at least for our purposes. Moreover, they do not allow the use of our differencing scheme for storing the entire tree efficiently at a single processor. The straightforward implementation of such a scheme may, therefore, sacrifice our ability to solve large problems sequentially. Furthermore, fault tolerance could also be decreased (see Sect. 7.2). It's in view of these considerations that we employ the master-slave model. See Sect. 10.1 for a discussion of future improvements to scalability.

7.2 Details of the Parallel Implementation

Parallel Configurations. SYMPHONY supports numerous configurations, ranging from completely sequential to fully parallel, allowing efficient execution in many different computational settings. As described in the previous section, there are five modules in the standard distributed configuration. Various subsets of these modules can be combined to form separate executables capable of communicating with each other across a network. When two or more modules are combined, they simply communicate through shared-memory instead of through message-passing. However, they are also forced to run in sequential fashion in this case, unless the user chooses to enable threading using an OpenMP compliant compiler (see next section).

As an example, the default distributed configuration includes a separate executable for each module type, allowing full parallelism. However, if cut generation is fast and not memory-intensive, it may not be worthwhile to have the LP solver and its associated cut generator work independently, as this increases communication overhead without much potential benefit. In this case, the cut generator functions can be called directly from the LP solver, creating a single, more efficient executable.

Inter-process Communication. SYMPHONY can utilize any third-party communication protocol supporting basic message-passing functions. All communication subroutines interface with SYMPHONY through a separate communications API. Currently, PVM is the only message-passing protocol supported, but interfacing with another protocol is a straightforward exercise.

Additionally, it is possible to configure the code to run in parallel using threading to process multiple search tree nodes simultaneously. Currently, this

is implemented using OpenMP compiler directives to specify the parallel regions of the code and perform memory locking functions. Compiling the code with an OpenMP compliant compiler will result in a shared-memory parallel executable. For a list of OpenMP compliant compilers and other resources, visit `http://www.openmp.org`.

Fault Tolerance. Fault tolerance is an important consideration for solving large problems on computing networks whose nodes may fail unpredictably. The tree manager tracks the status of all processes and can restart them as necessary. Since the state of the entire tree is known at all times, the most that will be lost if an LP process or cut generator process is killed is the work that had been completed on that particular search node. To protect against the tree manager itself or a cut pool being killed, full logging capabilities have been implemented. If desired, the tree manager can write out the entire state of the tree to disk periodically, allowing a restart if a fault occurs. Similarly, the cut pool process can be restarted from a log file. This not only allows for fault tolerance but also for full reconfiguration in the middle of solving a long-running problem. Such reconfiguration could consist of anything from adding more processors to moving the entire solution process to another network.

8 Applications

To make the ideas discussed thus far more concrete, we now introduce two practical applications from combinatorial optimization with which many readers will already be familiar. Graph-based problems, especially those involving *packing* and *routing* constraints, lend themselves particularly well to implementation in this type of framework. This is because many of the constraints, such as those dealing with connectivity of the solution, can be represented compactly using bit vectors, as described previously. Also, the one-to-one correspondence between variables and edges in the underlying graph yields a simple variable indexing scheme based on a lexicographic ordering of the edges. We therefore begin by describing the use of SYMPHONY to implement a basic solver for the Vehicle Routing Problem [75,74], and then move on to describe a Set Partitioning solver [36]. Summary computational results will be given later in Sect. 9.

8.1 The Vehicle Routing Problem

The *Vehicle Routing Problem* (VRP) was introduced by Dantzig and Ramser [32] in 1959. In this graph-based problem, a central depot $\{0\}$ uses k independent delivery vehicles, each of identical capacity C, to service integral demands d_i for a single commodity from customers $i \in N = \{1, \cdots, n\}$. Delivery is to be accomplished at minimum total cost, with c_{ij} denoting the transit cost from i to j, for $0 \leq i, j \leq n$. The cost structure is assumed *symmetric*, i.e., $c_{ij} = c_{ji}$ and $c_{ii} = 0$.

A solution for this problem consists of a partition $\{R_1, \ldots, R_k\}$ of N into k *routes*, each satisfying $\sum_{j \in R_i} d_j \leq C$, and a corresponding permutation, or *tour*, σ_i, of each route specifying the service ordering. This problem is naturally associated with the complete undirected graph consisting of nodes $N \cup \{0\}$, edges E, and edge-traversal costs $c_{ij}, \{i,j\} \in E$. A solution is a cloverleaf pattern whose k petals correspond to the routes serviced by the k vehicles. An integer programming formulation can be given as follows:

$$
\begin{aligned}
min \quad & \sum_{e \in E} c_e x_e \\
& \sum_{e=\{0,j\} \in E} x_e = 2k && (2) \\
& \sum_{e=\{i,j\} \in E} x_e = 2 \quad \forall i \in N && (3) \\
& \sum_{\substack{e=\{i,j\} \in E \\ i \in S, j \notin S}} x_e \geq 2b(S) \quad \forall S \subset N,\ |S| > 1 && (4) \\
& 0 \leq x_e \leq 1 \quad \forall e = \{i,j\} \in E,\ i,j \neq 0 && (5) \\
& 0 \leq x_e \leq 2 \quad \forall e = \{0,j\} \in E && (6) \\
& x_e \quad integral \quad \forall e \in E. && (7)
\end{aligned}
$$

Here, $b(S)$ can be any lower bound on the number of trucks needed to service the customers in set S, but for ease of computation, we define $b(S) = \lceil (\sum_{i \in S} d_i)/C \rceil$. The constraint (2) stipulates that there should be exactly k routes, while the constraints (3) require that each customer be visited by exactly one vehicle. The constraints (4) ensure connectivity of the solution while also implicitly ensuring that no route services total demand in excess of the capacity C.

Solver Implementation. Implementing a BCP algorithm based on the above formulation is straightforward using the framework. As discussed in Sect. 5.2, our main concern is with the treatment of constraints and variables. The number of variables is small enough for practical instances of this problem that we don't need to concern ourselves with column generation. We tried using column generation but did not find it to be advantageous. Our indexing scheme for the variables is based on a lexicographic ordering of the edges in the complete graph. This enables an easily calculable one-to-one mapping of edges to indices. To construct the core of the problem, we select the variables corresponding to the k cheapest edges adjacent to each node in the graph, as these are the variables most likely to have a positive value in some optimal solution. The remainder of the variables are added as *extra* variables in the root node. The hope is that most of these will be priced out of the problem quickly.

Constraints present different challenges, however. The number of constraints in the LP relaxation is exponential. Furthermore, the separation problem for the constraints (4) is known to be $\mathcal{NP}$-complete [9]. We therefore rely on heuristic separation routines to generate the constraints (4) dynamically during the search

process. Because the number of degree constraints (2) and (3) is small and we want them to appear in all relaxations (our heuristics depend on this fact), they are placed in the core. Initially, these are the only active constraints in the root node. In the basic solver presented here, dynamic constraint generation takes place only for the capacity constraints (4). Since the sets of variables with nonzero indices in these constraints correspond to the sets of edges across cuts in the graph, these constraints can be represented as a bit array indicating the nodes included in one shore of the cut (the one not containing the depot). To construct the row corresponding to a particular constraint, it suffices to check the edge corresponding to each active variable in the current relaxation and determine if its endpoints are on opposite shores of the cut. If so, the variable has a coefficient of one in the row. Otherwise, its coefficient is zero.

Besides cut generation, only a few other problem-specific routines are needed to implement the basic solver. Strong branching candidates are selected using a built-in default rule—select those variables nearest to .5 in value. Some logical fixing can also be done. For instance, if two edges adjacent to a particular node are already fixed to one, then all other edges adjacent to that node can be fixed to zero.

8.2 Set Partitioning Problem

In [36], Eső used an early version of SYMPHONY to implement a solver for the Set Partitioning Problem (SPP). Here, we review her work. Combinatorially, the Set Partitioning Problem can be stated as follows. We are given a ground set S of m objects and a collection C of subsets $S_1, \cdots, S_n$ of S, each with a given cost $c_j = c(S_j)$. We wish to select the minimum weight subfamily of C that forms a partition of S. This problem is well-studied and describes many important applications, including airline crew scheduling, vehicle routing, and political districting (see [41,11,51,19]).

To describe an integer programming formulation of the SPP, we construct matrix A, whose rows correspond to the members of S and whose columns correspond to the members of C. Entry a_{ij} is 1 if the i^{th} element of S is included in subset S_j; otherwise, we set a_{ij} to zero. Then the problem can simply be stated as

$$min \sum_{j=1}^{n} c_j x_j \tag{8}$$

$$s.t. \sum_{j=1}^{n} a_{ij} x_j = 1 \ , \quad 1 \leq i \leq m \tag{9}$$

$$x_j \in \{0,1\} \ , \quad 1 \leq j \leq n. \tag{10}$$

Each row of this formulation expresses the constraint that there must be exactly one member of the partition containing each element of the set S.

Solver Implementation. In crew scheduling applications, the matrix A can be extremely large and generating it can be difficult. Furthermore, solving the LP relaxation itself can also be very difficult. For the work in [36], the matrix A was assumed given. Even so, preprocessing A, in order to reduce its size without eliminating any optimal solutions, was found to be vitally important. Since finding feasible solutions to this problem within the search tree can prove quite difficult, heuristic solution procedures were used to find good upper bounds. Both the preprocessing of the matrix and the execution of heuristic procedures were performed before the branch and cut procedure was invoked. It is important to note that performing intensive computation prior to beginning the branch and cut procedure can potentially decrease parallel speedup by increasing the computation's sequential fraction (see 7.1).

Unlike many combinatorial problems, for crew scheduling models, it was difficult to judge a priori the relative importance of each column and its likelihood of participating in some optimal solution. This is because the magnitude of the objective function coefficient corresponding to a particular column is not necessarily a good indicator of its usefulness. Large objective function coefficients may simply correspond to columns representing large subsets of the set S. Because of this, the set of core variables was taken to be empty in order to allow removal by reduced cost and logical implications in the lower levels of the search tree. On the other hand, all constraints remaining after preprocessing were taken to be core constraints.

Part of the difficulty inherent in crew scheduling models stems from the extensive computation often required for solving the LP relaxations. In particular, the simplex algorithm sometimes has trouble optimizing these linear programs. Therefore, the barrier method with dual crossover was used to solve the initial LP relaxation and derive a feasible basis for the dual simplex algorithm, which was then used for the remaining calculations. The same problem reduction procedures that were so important during preprocessing were also employed throughout the tree to further reduce the matrix after branching or otherwise fixing variables. In addition, a primal heuristic was employed to derive new feasible solutions and hence improve the current upper bound. Candidates for branching were taken from among the variables, existing cuts that had become slack, and cuts produced specifically for branching. The algorithm employed detection of tailing off and forced branching whenever such a condition was detected.

Many known classes of cutting planes can be used to strengthen the LP relaxations for this problem. Examples are clique inequalities, odd holes, packing and cover odd holes, odd antiholes, and other lifted versions of these classes. Because it was not clear which of these classes would produce the strongest cuts for a particular instance, the cut generator was itself parallelized in order to find cuts in several classes simultaneously.

9 Computational Experience

In this section, we describe our experience using the framework to solve the two classical combinatorial optimization problems we have already described (VRP

and SPP). Although we report some running times in Sect. 9.2 below, what follows should be considered anecdotal evidence based on our observations and experience with the development of SYMPHONY.

9.1 Sequential Performance

Performance of our code when running sequentially has improved dramatically since the first published results in [75]. Although the same fundamental design has been maintained since the inception of the project, the implementation has been streamlined and improved to the point that the running time for a standard set of Vehicle Routing Problems, *even after adjusting for increased processing speed*, has improved by more than two orders of magnitude since 1995. This underscores the fact that, first and foremost, *implementational details can produce a marked difference in efficiency for BCP algorithms.* In the following sections, we summarize a few of the "details" that have proven to be important. However, we emphasize that the most effective way to learn about the implementation of these algorithms is to examine the documentation and source code itself [76].

It is well-known that the vast majority of the computing time for BCP is spent in two activities—solving LP relaxations and generating new cuts and variables. In SYMPHONY, both of these activities are performed by external libraries. The result is that, in practice, very little time is spent executing instructions that actually reside within the framework itself. Hence, improvements to running times must come not through reducing the time spent within the framework, but rather through *reducing the time spent in code outside the framework.* Although we have no control over the efficiency of these external codes, we *can* control not only the input to these external subroutines, but also the number of times they need to be called. To achieve real improvements in efficiency, one must guide the solution process with this in mind.

Linear Program Management. To reduce time spent solving linear programs, we emphasize once again that the most important concept is the ability to limit the size of the LP relaxations by *allowing cuts and variables to be fluidly generated, activated, and deactivated* at various points during the (iterative) solution process. This entire process must be managed in such a way as to reduce the size of the matrix while not reducing the quality of the solutions produced. *It is also critical to maintain LP warm-start information (i.e., a description of the current basis) throughout the tree to allow efficient processing of each search tree node.*

Constraints. The most effective approach to managing the constraints in the LP relaxation has been to *be conservative with adding constraints to the relaxation while being liberal with removing them.* We have found that by deleting ineffective constraints quickly, we can significantly reduce LP solution time. Of course, it would be better not to add these ineffective constraints in the first place. The local cut pools, which allow only the "best" constraints to be added

in each iteration, have been instrumental in reducing the number of cuts that eventually do become ineffective. This combination approach has worked extremely well.

With respect to reducing the time spent *generating* constraints, *the global cut pool is effective for constraints whose generation is relatively expensive.* For constraints that are inexpensive to generate in comparison to the time spent solving LPs, the cost of maintaining the cut pool does not always pay off. Our approach to management of the cut pool has been similar to that of managing the constraints of the linear programs, but here it is less clear how to effectively keep its size under control. Our conservative policy with respect to adding constraints to the pool has produced good results. However, the question of how to determine which constraints should be removed from the cut pool needs further consideration.

Variables. Although cuts and variables can be handled symmetrically in many respects, there are some major differences. While generating cuts helps tighten the formulation and increase the lower bound, generating variables has the opposite effect. Therefore, *one must be somewhat careful about when variable generation is invoked*, as it destroys monotonicity of the objective function, upon which algorithmic performance sometimes depends. Furthermore, before a node can be properly fathomed in BCP, it is necessary to ensure that there are no columns whose addition to the problem could eliminate the conditions necessary to fathom the node in question, i.e., by either lowering the objective function value back below the current upper bound or by restoring feasibility. Thus, the user must be mindful of whether the node is about to be fathomed before performing column generation.

In many problem settings, particularly those involving combinatorial optimization, it is much easier to judge *a priori* the importance of a particular variable (based on the problem structure and the structure of the objective function) than it is to judge the importance of a constraint. *It is important to take advantage of this information.* We have mentioned two different ways in which we can do this. By declaring some "unimportant" variables inactive in the root node, the user can delay including them in any LP relaxation until the column generation step. In the two-phase method with repricing in the root node (see Sect. 6.3), it is possible, even probable, that these variables would simply be priced out immediately in the second phase. In theory, this should allow much faster processing of subproblems and less time spent solving LPs.

In our experience with combinatorial problem solving, however, generating columns, unless done very efficiently, can be an expensive operation whose cost is not usually justified. This is especially true in the presence of a good initial upper bound. In this case, most "unimportant" columns end up being priced out either in the root node or relatively soon thereafter. *Therefore, if efficient explicit generation of all variables is possible in the root node and there is sufficient memory to store them, this is generally the best option.* This allows variables to be fixed by reduced cost and nodes to be fathomed without expensive pricing. However, if either (1) there is not enough memory to store all of the problem's

columns at once, (2) it is expensive to generate the variables, or (3) there is an efficient method of pricing large subsets of variables at once, then column generation might be an effective technique.

Search Tree Management. The primary way in which solution time can be reduced is by *reducing the size of the search tree.* Effective branching, on both variables and constraints, is an important tool for this reduction. *One of the most effective methods available for branching is* strong branching, discussed in Sect. 6.2. In a recent test on a set of Vehicle Routing Problems, it was found that the number of search nodes explored was reduced (in comparison with standard branching) by over 90% using strong branching with just seven candidate branching objects. For this reason, use of strong branching is highly recommended. Nonetheless, this branching should be delayed as long as possible. As long as violated cuts can be found and the relaxation solution value has not tailed off, processing of the current search tree node should continue.

Another way to affect the size of the search tree is through an effective search strategy, as we discussed in 6.3. We have found that *a hybrid of "best-first," along with controlled diving produces the best results.* Diving leads to improved upper bounds and reductions in node set-up costs while minimizing communication with the tree manager. This can be significant when employing parallelism.

9.2 Parallel Performance

Given the modular design of SYMPHONY, the transition from sequential to parallel processing is straightforward. However, the centralized management of the search process and centralized storage of the search tree, while highly effective for sequential processing, does not lead to a scalable parallel algorithm. That said, *the parallel efficiency of our approach is very high for small numbers of processors.* For many practical settings, this is all that is needed. Furthermore, as discussed earlier, this parallelism is achieved without sacrificing any of the efficiencies to be gained in the more typical sequential setting.

Parallel efficiency in BCP is achieved mainly through effective qualitative and quantitative load balancing. For small numbers of processors, our approach handles load balancing with ease. The search trees produced in parallel runs are approximately the same size as those produced sequentially, leading to *linear speedup.* See the sample computational results in the next section for an example of this.

As the number of processors increases, the tree manager eventually becomes a computational bottleneck. As indicated in Sect. 7.1, the point at which this happens is highly problem-dependent. When this point is reached, parallel efficiency can be increased by limiting communication between the tree manager and the LP solver as much as possible. For instance, increased diving can significantly enhance large-scale parallel efficiency. For more ideas on improving the scalability of these algorithms, see the discussion in Sect. 10.1.

9.3 Computational Results for Vehicle Routing

Our experience with the VRP has reinforced many of the lessons already discussed. We experimented with column generation, but found it to be more effective to simply include all variables in the root node. However, placing only the k shortest edges adjacent to each customer node in the problem core did lead to significant gains. We found strong branching to be an extremely effective tool for reducing the size of the search tree. However, we experienced diminishing returns when examining more than 7-10 candidates. We used a hybrid diving strategy that allowed us to find feasible solutions early in the search process. In many cases, this led to running times almost equaling those achieved by computing a heuristic upper bound prior to beginning the branch and cut phase of the algorithm. Although effective constraint generation is absolutely critical for this problem, use of the cut pool did not have a dramatic effect. This is probably because the cuts we applied are relatively easy to generate.

Table 1 presents summary computational results for recent testing of the basic VRP solver discussed earlier. These results are not intended to be comprehensive, but are presented for illustrative purposes. For more detailed discussion of using SYMPHONY to solve the VRP and more in-depth computational results, see [74]. Tests here were performed on a network of 3 workstations powered by 4 DEC Alpha processors each using CPLEX as the LP engine. The problems are easy- to medium-difficulty problems from VRPLIB [81] and other sources [79].

From Table 1, it is evident that the number of nodes in the search tree is largely independent of the number of LP processes being utilized. This essentially ensures linear speedup as long as parallel overhead remains low and there are no computational bottlenecks. Predictably, as the number of processors increases, the idle time also increases, indicating that the tree manager is becoming saturated with requests for data. Because not many strong cuts are known for this model, we tend to rely on quick enumeration to solve these problems. This is possible because the LP relaxations are relatively easy to solve. It is therefore common to develop large search trees in a short period of time. Our compact storage scheme allows us to deal with these large search trees. However, scalability suffers in this situation.

Table 1. Summary Computational Results for VRP instances

	Number of LP processes used			
	1	2	4	8
Number of search tree nodes	6593	6691	6504	6423
Wallclock solution time (sec)	2493	1281	666	404
Wallclock solution time per node	0.38	0.38	0.41	0.50
Idle time per node	0.00	0.01	0.03	0.08

9.4 Computational Results for Set Partitioning

In contrast to the Vehicle Routing Problem, the Set Partitioning Problem produces relatively small search trees but search nodes can be much more difficult to process. In particular, the simplex algorithm had difficulty in solving some of the LP relaxations encountered in this problem. However, as mentioned earlier, Eső does report success with using the barrier method with dual crossover to solve the initial LP relaxations. Problem size reduction techniques help to control the size of the LP relaxations, leading to reduced node processing times.

In this problem, tailing off can be a problem, so branching was invoked whenever the lower bound did not show significant improvement over a sequence of iterations. As in the VRP, strong branching was found to be an important tool in reducing the size of the search tree. However, for the SPP, choosing specially constructed cuts, in addition to variables, as branching candidates was found to be important. Dynamic cut generation was also critical to efficiency. Extensive computational results on a variety of problems (assembled from the crew scheduling literature) are reported in [36]. The implementation solved many of these problems to optimality in the root node of the enumeration tree, so there was no need in such instances for parallel processing. Of the remaining problems, some proved too difficult to solve, mainly due to difficulties in solving LP relaxations, as indicated above. Several of the more difficult models did yield solutions, however, with significant multi-processing effort. The principles observed here were similar to those for the VRP: The number of search tree nodes was essentially independent of the number of LP processes, resulting in linear speed-up for as many as 8 LP processes. For example, with model *aa04*, a well-known difficult problem taken from [50], the following results shown in Table 2 were obtained (see also Table 5.12 of [36]). The computing platform is an IBM SP2 with LP solver CPLEX.

Table 2. Sample computational results for the crew scheduling model *aa04*

	Number of LP processes used			
	1	2	4	8
Number of search tree nodes	283	268	188	234
Depth of search tree	25	22	16	17
Wallclock solution time (sec)	2405	1111	350	240

10 Future Development

Although the theory underlying BCP algorithms is well-developed, our knowledge of how to implement these algorithms continues to improve and grow. To some extent, effective implementation of these algorithms will continue to depend on problem-specific techniques, especially cut generation. However, we have already learned a great deal about how to remove certain burdens from the user by

implementing generic defaults that work well across a wide variety of problems. In this section, we offer a few ideas about where future growth will occur.

10.1 Improving Parallel Scalability

With the state of technology driving an increasing interest in parallel computation, it is likely that parallel algorithms will continue to play an important role in the field of optimization. In these notes, we have touched on some of the central issues surrounding the parallelization of BCP algorithms, but much remains to be learned. In particular, more scalable approaches to BCP need to be developed. As we have already pointed out, this clearly involves some degree of decentralization. However, the schemes that have appeared in the literature (mostly applied to parallel branch and bound) appear inadequate for the more complex challenges of BCP.

The most straightforward approach to improving scalability is simply to increase the task granularity and thereby reduce the number of decisions the tree manager has to make, as well as the amount of data it has to send and receive. To achieve this, we could simply allow each LP process to examine an entire subtree or portion of a subtree before checking back for additional work. This approach would be relatively easy to implement, but has some potentially serious drawbacks. The most serious of these is that the subtree being examined could easily turn out to contain mostly unpromising nodes that would not have been examined otherwise. Hence, this scheme seems unlikely to produce positive results in its most naive form.

Another approach is to attempt to relieve the bottleneck at the central tree manager by only storing the information required to make good load-balancing decisions (most notably, the lower bound in each search tree node) centrally. The data necessary to generate each search node could be stored either at one of a set of "subsidiary tree managers" or within the LP modules themselves. This is similar to a scheme implemented by Eckstein [33] for parallel branch and bound. Such a scheme would maintain the advantages of global decision-making while moving some of the computational burden from the tree manager to other processes. However, effective load balancing would necessarily involve an expensive increase in the amount of data being shuttled between processes. Furthermore, the differencing scheme we use for storing the search tree will not extend easily to a decentralized environment.

10.2 Other Directions

The vast majority of research on BCP has concentrated on the now well-studied technique of branch and cut. Branch and price, on the other hand, has received relatively little attention and the integration of these two methods even less. In particular, issues related to if, when, and how to effectively generate new variables, independent of the problem setting, need further investigation. Effective management of pools for both cuts and variables is another important computational issue which deserves attention. As we pointed out several times,

SYMPHONY currently has a bias towards the implementation of branch and cut algorithms. We intend to improve and generalize our implementation of variable generation in order to make the framework more flexible and efficient for branch and price.

Outside of branching on fractional variables, few generic branching rules have been developed. Most BCP implementations still rely on variable branching because it is easy to implement and relatively effective. However, there are situations in which it can be ineffective when compared to branching on a well-selected cut or on a set of objects. Automatic methods of determining which cuts will make effective branching objects have yet to be examined.

Until recently, almost all BCP algorithms have utilized simplex-based LP solvers to perform lower bounding. Currently, these solvers still offer the best performance across a wide range of problems. However, new solution techniques, such as the volume algorithm (see [14]) are showing promise in helping to solve those problems on which the simplex algorithm falters. As discussed in 8.2, we have already seen in [36] that the barrier method successfully provided an alternative to the simplex method in solving large-scale LP problems arising in airline crew scheduling models. Relaxation schemes that use techniques other than linear programming, e.g., semi-definite programming, are also coming into prominence. Interfacing with these new solvers should provide fruitful avenues for further improvement in BCP methodology.

11 Conclusion

In these notes, we have given the reader a summary of many important challenges of implementing branch, cut, and price algorithms. However, there are many more details to be explored below the surface. We encourage the interested reader to visit `http://BranchAndCut.org` for more extensive documentation and source code for SYMPHONY.

References

1. Amdahl, G.M.: Validity of the Single-processor Approach to Achieving Large-scale Computing Capabilities. In AFIPS Conference Proceedings **30** (Atlantic City, N.J., April 18-20), AFIPS Press (1967), 483
2. Applegate, D., Bixby, R., Chvátal, V., and Cook, W.: On the solution of traveling salesman problems, Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung, International Congress of Mathematicians (1998), 645
3. Applegate, D., Bixby, R., Chvátal, V., and Cook, W.: CONCORDE TSP Solver. Available at `www.keck.caam.rice.edu/concorde.html`
4. Applegate, D., Bixby, R., Chvátal, V., and Cook, W.: Finding Cuts in the TSP. DIMACS Technical Report 95-05, 1995
5. Agarwal, Y., Mathur, K., and Salkin, H.M.: Set Partitioning Approach to Vehicle Routing. Networks **7**, 731, 1989

6. Araque, J.R., Kudva, G., Morin, T.L., and Pekny, J.F.: A Branch-and-Cut Algorithm for Vehicle Routing Problems. Annals of Operations Research **50**, 37, 1994
7. Araque, J.R., Hall, L., and Magnanti, T.: Capacitated Trees, Capacitated Routing and Associated Polyhedra. Discussion paper 9061, CORE, Louvain La Nueve, 1990
8. Augerat, P., Belenguer, J.M., Benavent, E., Corberán, A., and Naddef, D.: Separating Capacity Constraints in the CVRP Using Tabu Search. European Journal of Operations Research **106**, 546, 1998
9. Augerat, P., Belenguer, J.M., Benavent, E., Corberán, A., Naddef, D., and Rinaldi, G.: Computational Results with a Branch and Cut Code for the Capacitated Vehicle Routing Problem. Research Report 949-M, Université Joseph Fourier, Grenoble, France, 1995
10. Balas, E., Ceria, S., and Cornuéjols, G.: Mixed 0-1 Programming by Lift-and-Project in a Branch-and-Cut Framework. Management Science **42**, 9, 1996
11. Balas, E., and Padberg, M.W.: Set Partitioning: A Survey. SIAM Review **18**, 710, 1976
12. Balas, E., and Toth, P.: Branch and Bound Methods. In Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G, and Shmoys, D.B., eds., The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, Wiley, New York, 361, 1985.
13. Balinski, M.L., and Quandt, R.E.: On an Integer Program for a Delivery Problem. Operations Research **12**, 300, 1964
14. Barahona, F., and Anbil, R.: The Volume Algorithm: Producing Primal Solutions with a Subgradient Method. Mathematical Programming **87**, 385, 2000
15. Barnhart, C., Johnson, E.L., Nemhauser, G.L., Savelsbergh, M.W.P., and Vance, P.H.: Branch-and-Price: Column Generation for Huge Integer Programs. Operations Research **46**, 316, 1998
16. Benchouche, M., Cung, V.-D., Dowaji, S., Le Cun, B., Mautor, T., and Roucairol, C.: Building a Parallel Branch and Bound Library. In Solving Combinatorial Optimization Problems in Parallel, Lecture Notes in Computer Science **1054**, Springer, Berlin, 201, 1996
17. Blasum, U., and Hochstättler, W.: Application of the Branch and Cut Method to the Vehicle Routing Problem. Zentrum für Angewandte Informatik Köln Technical Report zpr2000-386, 2000
18. Boehning, R.L., Butler, R.M., and Gillet, B.E.: A Parallel Integer Linear Programming Algorithm. European Journal of Operations Research **34**, 393, 1988
19. Borndörfer, R.: Aspects of Set Packing, Patitioning, and Covering. PhD. Dissertation, Technischen Universität Berlin, 1997
20. Campos, V., Corberán, A., and Mota, E.: Polyhedral Results for a Vehicle Routing Problem. European Journal of Operations Research **52**, 75, 1991
21. Chen, Q., and Ferris, M.C.: FATCOP: A Fault Tolerant Condor-PVM Mixed Integer Programming Solver. University of Wisconsin CS Department Technical Report 99-05, Madison, WI, 1999
22. Christofides, N., and Eilon, S.: An Algorithm for the Vehicle Dispatching Problem, Operational Research Quarterly **20**, 309, 1969
23. Christofides, N., Mingozzi, A., and Toth, P.: Exact Algorithms for Solving the Vehicle Routing Problem Based on Spanning Trees and Shortest Path Relaxations. Mathematical Programming **20**, 255, 1981
24. Chvátal, V.: Linear Programming. W.H. Freeman and Company, San Francisco, 1983

25. Cornuéjols, G., and Harche, F., Polyhedral Study of the Capacitated Vehicle Routing Problem. Mathematical Programming **60**, 21, 1993
26. Cordier, C., Marchand, H., Laundy, R., and Wolsey, L.A.: bc-opt: A Branch-and-Cut Code for Mixed Integer Programs. Mathematical Programming **86**, 335, 1999
27. Common Optimization INterface for Operations Research (COIN). `http:\penalty\z@//www.coin-or.org`
28. Crowder, H., and Padberg, M.: Solving Large Scale Symmetric Traveling Salesman Problems to Optimality. Management Science **26**, 495, 1980
29. Crowder, H., Johnson, E.L., and Padberg, M.: Solving Large-Scale Zero-One Linear Programming Problems. Operations Research **31** 803, 1983
30. Cullen, F.H., Jarvis, J.J., and Ratliff, H.D.: Set Partitioning Based Heuristic for Interactive Routing. Networks **11**, 125, 1981
31. Cung, V.-D., Dowaji, S., Le Cun, B., Mauthor, T., and Roucairol, C.: Concurrent Data Structures and Load Balancing Strategies for Parallel Branch and Bound/A* Algorithms. DIMACS Series in Discrete Optimization and Theoretical Computer Science **30**, 141, 1997
32. Dantzig, G.B, and Ramser, J.H., The Truck Dispatching Problem. Management Science **6**, 80, 1959
33. Eckstein, J.: Parallel Branch and Bound Algorithms for General Mixed Integer Programming on the CM-5. SIAM Journal on Optimization **4**, 794, 1994
34. Eckstein, J.: How Much Communication Does Parallel Branch and Bound Need? INFORMS Journal on Computing **9**, 15, 1997
35. Eckstein, J., Phillips, C.A., and Hart, W.E.: PICO: An Object-Oriented Framework for Parallel Branch and Bound. RUTCOR Research Report 40-2000, Rutgers University, Piscataway, NJ, 2000
36. Eső, M.: Parallel Branch and Cut for Set Partitioning. Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, NY, 1999
37. Fisher, M.L.: Optimal Solution of Vehicle Routine Problems Using Minimum k-Trees. Operations Research **42**, 141, 1988
38. Fisher, M.L., and Jaikumar, R.: A Generalized Assignment Heuristic for Solving the VRP. Networks **11**, 109, 1981
39. Foster, B.A., and Ryan, D.M.: An Integer Programming Approach to the Vehicle Scheduling Problem, Operational Research Quarterly **27**, 367, 1976
40. Garey, M.R., and Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Co., San Francisco, 1979
41. Garfinkel, R.S., and Nemhauser, G.L.: Integer Programming. Wiley, New York, 1972
42. Geist, A., et al.: PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, MA, 1994
43. Gendron, B., and Crainic, T.G.: Parallel Branch and Bound Algorithms: Survey and Synthesis. Operations Research **42**, 1042, 1994
44. Golub, G.H., and Van Loan, C.F.: Matrix Computations. The Johns Hopkins University Press, Baltimore, 1989
45. Grötschel, M., Jünger, M., and Reinelt, G.: A Cutting Plane Algorithm for the Linear Ordering Problem. Operations Research **32**, 1155, 1984
46. Grama, A., and Kumar, V.: Parallel Search Algorithms for Discrete Optimization Problems. ORSA Journal on Computing **7**, 365, 1995
47. Gustafson, J.L.: Re-evaluating Amdahl's Law. Comm. of the ACM **31**, 532, 1988

48. Held, M., and Karp, R.M.: The Traveling Salesman Problem and Minimal Spanning Trees. Operations Research **18**, 1138, 1969
49. ILOG CPLEX 6.5 Reference Manual, ILOG, 1994
50. Hoffman, K., and Padberg, M.: LP-Based Combinatorial Problem Solving. Annals of Operations Research **4**, 145, 1985/6
51. Hoffman, K., and Padberg, M.: Solving Airline Crew Scheduling Problems by Branch-and-cut. Management Science **39**, 657, 1993
52. Jünger, M., and Thienel, S.: The ABACUS System for Branch and Cut and Price Algorithms in Integer Programming and Combinatorial Optimization. Software Practice and Experience **30**, 1325, 2000
53. Jünger, M., and Thienel, S.: Introduction to ABACUS—a branch-and-cut system. Operations Research Letters **22**, 83, 1998
54. Jünger, M., Reinelt, G., and Thienel, S.: Practical Problem Solving with Cutting Plane Algorithms in Combinatorial Optimization. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 111, 1995
55. Kozen, D.C.: The Design and Analysis of Algorithms, Springer-Verlag, New York, 1992
56. Kopman, L.: A New Generic Separation Algorithm and Its Application to the Vehicle Routing Problem. Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, NY, 1999
57. Kumar, V., and Rao, V.N.: Parallel Depth-first Search. Part II. Analysis. International Journal of Parallel Programming **16**, 501, 1987
58. Kumar, V., and Gupta, A.: Analyzing Scalability of Parallel Algorithms and Architectures. Journal of Parallel and Distributed Computing **22**, 379, 1994
59. Ladányi, L.: Parallel Branch and Cut and Its Application to the Traveling Salesman Problem. Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, NY, 1996
60. Laporte, G., and Nobert, Y.: Comb Inequalities for the Vehicle Routing Problem. Methods of Operations Research **51**, 271, 1981
61. Laporte, G., Nobert, Y., and Desrouchers, M.: Optimal Routing with Capacity and Distance Restrictions. Operations Research **33**, 1050, 1985
62. Laursen, P.: Can Parallel Branch and Bound without Communication Be Effective? SIAM Journal of Optimization **4**, 288, 1994
63. Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D.B.: The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. Wiley, New York, 1985
64. Letchford, A.N., Eglese, R.W., and Lysgaard, J.: Multi-star Inequalities for the Vehicle Routing Problem. Technical Report available at `http://www.lancs.ac.uk/staff/letchfoa/pubs.htm`
65. Linderoth, J.: Topics in Parallel Integer Optimization. Ph.D. Dissertation, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA, 1998
66. Marsten, R.: The Design of The XMP Linear Programming Library. ACM Transactions on Mathematical Software **7**, 481, 1981
67. Martin, A.: Integer Programs with Block Structure. Habilitation Thesis, Technischen Universität Berlin, 1998
68. Naddef, D., and Rinaldi, G.: Branch and Cut. In Toth, P., and Vigo, D., eds., Vehicle Routing, SIAM, 2000.
69. Nagamochi, H., and Ibaraki, T.: Computing Edge Connectivity in Multigraphs and Capacitated Graphs. SIAM Journal of Discrete Mathematics **5**, 54, 1992

70. Nemhauser, G.L., Savelsbergh, M.W.P., and Sigismondi, G.S.: MINTO, a Mixed INTeger Optimizer. Operations Research Letters **15**, 47, 1994
71. Nemhauser, G.L., and Wolsey, L.A.: Integer and Combinatorial Optimization. Wiley, New York, 1988
72. Padberg, M., and Rinaldi, G.: A Branch-and-Cut Algorithm for the Resolution of Large-Scale Traveling Salesman Problems. SIAM Review **33**, 60, 1991
73. Radaramanan, R., and Choi, K.: A Branch and Bound Algorithm for Solving the Vehicle Routing Problem. Proceedings of the 8th Annual Conference on Computers and Industrial Engineering, 236.
74. Ralphs, T.K., Kopman, L., Pulleyblank, W.R., and Trotter Jr., L.E.: On the Capacitated Vehicle Routing Problem. Submitted for publication
75. Ralphs, T.K.: Parallel Branch and Cut for Vehicle Routing. Ph.D. Dissertation, Field of Operations Research, Cornell University, Ithaca, NY, 1995
76. Ralphs, T.K., SYMPHONY Version 2.8 User's Guide. Available at `http:\penalty\z@//www.branchandcut.org/SYMPHONY`
77. Ralphs, T.K., and Ladányi, L. Computational Experience with Branch, Cut, and Price. To be submitted.
78. Ralphs, T.K., and Ladányi, L.: SYMPHONY: A Parallel Framework for Branch and Cut. White paper, Rice University, 1999.
79. Ralphs, T.K.: Repository of Vehicle Routing Problem Instances. Available at `http://www.BranchAndCut.org/VRP`
80. Rao, V.N., and Kumar, V.: Parallel Depth-first Search. Part I. Implementation. International Journal of Parallel Programming **16**, 479, 1987
81. Reinelt, G.: TSPLIB—A traveling salesman problem library. ORSA Journal on Computing **3**, 376, 1991. Update available at `http://www.crpc.rice.edu/softlib/tsplib/`
82. Rushmeier, R., and Nemhauser, G.L.: Experiments with Parallel Branch and Bound Algorithms for the Set Covering Problem. Operations Research Letters **13**, 277, 1993
83. Savelsbergh, M.W.P.: A Branch-and-Price Algorithm for the Generalized Assignment Problem. Operations Research **45**, 831, 1997
84. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, New York, 1986
85. Shinano, Y., Higaki, M., and Hirabayashi, R.: Generalized Utility for Parallel Branch and Bound Algorithms. Proceedings of the 1995 Seventh Symposium on Parallel and Distributed Processing, IEEE Computer Society Press, Los Alamitos, CA, 392, 1995
86. Shinano, Y., Harada, K., and Hirabayashi, R.: Control Schemes in a Generalized Utility for Parallel Branch and Bound. Proceedings of the 1997 Eleventh International Parallel Processing Symposium, IEEE Computer Society Press, Los Alamitos, CA, 621, 1997
87. Tschöke, S., and Polzer, T.: Portable Parallel Branch and Bound Library User Manual, Library Version 2.0. Department of Computer Science, University of Paderborn

TSP Cuts Which Do Not Conform to the Template Paradigm

David Applegate[1], Robert Bixby[2], Vašek Chvátal[3], and William Cook*[4]

[1] Algorithms and Optimization Department, AT&T Labs – Research, Florham Park, NJ 07932, USA
[2] Computational and Applied Mathematics, Rice University, Houston, TX 77005, USA
[3] Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA
[4] Program in Applied and Computational Mathematics, Princeton University, Princeton, NJ 08544, USA

Abstract. The first computer implementation of the Dantzig-Fulkerson-Johnson cutting-plane method for solving the traveling salesman problem, written by Martin, used subtour inequalities as well as cutting planes of Gomory's type. The practice of looking for and using cuts that match prescribed templates in conjunction with Gomory cuts was continued in computer codes of Miliotis, Land, and Fleischmann. Grötschel, Padberg, and Hong advocated a different policy, where the template paradigm is the only source of cuts; furthermore, they argued for drawing the templates exclusively from the set of linear inequalities that induce facets of the TSP polytope. These policies were adopted in the work of Crowder and Padberg, in the work of Grötschel and Holland, and in the work of Padberg and Rinaldi; their computer codes produced the most impressive computational TSP successes of the nineteen eighties. Eventually, the template paradigm became the standard frame of reference for cutting planes in the TSP. The purpose of this paper is to describe a technique for finding cuts that disdains all understanding of the TSP polytope and bashes on regardless of all prescribed templates. Combining this technique with the traditional template approach was a crucial step in our solutions of a 13,509-city TSP instance and a 15,112-city TSP instance.

1 The Cutting-Plane Method and Its Descendants

The groundbreaking work of Dantzig, Fulkerson, and Johnson [19] on the traveling salesman problem introduced the *cutting-plane method*, which can be used to attack any problem

$$\text{minimize } c^T x \text{ subject to } x \in \mathcal{S}, \tag{1}$$

where $\mathcal{S}$ is a finite subset of some Euclidean space $\mathbb{R}^m$, provided that an efficient algorithm to recognize points of $\mathcal{S}$ is available. This method is iterative; each

* Supported by ONR Grant N00014-01-1-0058

M. Jünger and D. Naddef (Eds.): Computat. Comb. Optimization, LNCS 2241, pp. 261–303, 2001.
© Springer-Verlag Berlin Heidelberg 2001

of its iterations begins with a *linear programming relaxation* of (1), meaning a problem

$$\text{minimize } c^T x \text{ subject to } Ax \leq b, \tag{2}$$

where the polyhedron P defined as $\{x : Ax \leq b\}$ contains $\mathcal{S}$ and is bounded. Since P is bounded, we can find an optimal solution x^* of (2) which is an extreme point of P. If x^* belongs to $\mathcal{S}$, then it constitutes an optimal solution of (1); otherwise, some linear inequality *separates* x^* *from* $\mathcal{S}$ in the sense of being satisfied by all the points in $\mathcal{S}$ and violated by x^*; such an inequality is called a *cutting plane* or simply a *cut*. In the latter case, we find a family of cuts, add them to the system $Ax \leq b$, and use the resulting tighter relaxation of (1) in the next iteration of the cutting-plane method.

Each iteration of the method requires first finding x^* and then finding a family of cuts. Finding x^* presents no problem: this is what the simplex method and other LP algorithms are for. Finding cuts is the challenge that has to be answered with each new application of the cutting-plane method; we shall return to this challenge later.

Progress of the cutting-plane method towards solving a particular instance of problem (1) is often estimated by the increase in the optimal value of its LP relaxation; as more and more cuts are added, these increases tend to get smaller and smaller. When they become unbearably small, the sensible thing to do may be to *branch*: having chosen a vector α and numbers β', β'' with $\beta' < \beta''$ such that

$$\alpha^T x^* \in (\beta', \beta'') \quad \text{and} \quad \{\alpha^T x : x \in \mathcal{S}\} \subset (-\infty, \beta'] \cup [\beta'', +\infty),$$

we solve the two *subproblems*,

$$\text{minimize } c^T x \text{ subject to } x \in \mathcal{S},\ \alpha^T x \leq \beta'$$

and

$$\text{minimize } c^T x \text{ subject to } x \in \mathcal{S},\ \alpha^T x \geq \beta'',$$

separately. (If all the elements of $\mathcal{S}$ are integer vectors and some component x_e^* of x^* is not an integer, then we may choose α so that $\alpha^T x$ is identically equal to x_e and set $\beta' = \lfloor x_e^* \rfloor$, $\beta'' = \lceil x_e^* \rceil$.) At some later time, one or both of these two subproblems may be split into sub-subproblems, and so on; in the resulting binary tree of subproblems, each node has the form

$$\text{minimize } c^T x \text{ subject to } x \in \mathcal{S},\ Cx \leq d \tag{3}$$

for some system $Cx \leq d$ of linear inequalities and each leaf will have been either solved without recourse to branching or else found irrelevant since the optimal value of its LP relaxation turned out to be at least as large as $c^T x$ for some previously known element x of $\mathcal{S}$. This scheme is one of the many variants of the *branch-and-bound* method. (The term "branch-and-bound", coined by Little, Murty, Sweeney, and Karel [44], refers to a general class of algorithms that originated in the work of Bock [7], Croes [16], Eastman [20], Rossman and Twery

[65], and Land and Doig [42]; in this more general context, relaxations of (1) may come from a universe far wider than that of linear programming relaxations (2) and each subproblem may be split into more than two sub-subproblems.)

Computer codes written by Hong [39], Miliotis [48], and Grötschel, Jünger, and Reinelt [33] introduced a particular variant of this variant, where each subproblem is attacked by the cutting-plane method; in these codes, the cuts introduced in solving (3) are satisfied by all points of $\mathcal{S}$ (rather than merely by all points x of $\mathcal{S}$ which satisfy $Cx \leq d$), and so they can be passed to any other subproblem later on. Padberg and Rinaldi [60] termed this approach *branch-and-cut*.

2 Ways of Finding Cuts

The symmetric traveling salesman problem, or TSP for short, is this: given a finite number of "cities" along with the cost of travel between each pair of them, find the cheapest way of visiting all of the cities and returning to your starting point. The travel costs are symmetric in the sense that traveling from city X to city Y costs just as much as traveling from Y to X; the "way of visiting all the cities" is simply the order in which the cities are visited. This problem is a special case of (1) with $m = n(n-1)/2$, where n is the number of the cities and $\mathcal{S}$ consists of the set of incidence vectors of all the hamiltonian cycles through the set V of the n cities; in this context, hamiltonian cycles are commonly called *tours*. Dantzig, Fulkerson, and Johnson illustrated the power of their cutting-plane method by solving a 49-city instance of the TSP, an impressive size at the time. They let the initial polyhedron P consist of all vectors x, with components subscripted by edges of the complete graph on V, that satisfy

$$0 \leq x_e \leq 1 \quad \text{for all edges } e \tag{4}$$

and

$$\sum(x_e : v \in e) = 2 \quad \text{for all cities } v. \tag{5}$$

(Throughout this paper, we treat the edges of a graph as two-element subsets of its vertex-set: $v \in e$ means that vertex v is an endpoint of edge e; $e \cap Q \neq \emptyset$ means that edge e has an endpoint in set Q; $e - Q \neq \emptyset$ means that edge e has an endpoint outside set Q; and so on.) All but two of their cuts have the form $\sum(x_e : e \cap Q \neq \emptyset, e - Q \neq \emptyset) \geq 2$, where Q is a nonempty proper subset of V; they are satisfied by all tours through V because every such tour has to move from Q to $V - Q$ at least once and it has to move back to Q after each such crossing. Dantzig, Fulkerson, and Johnson called such inequalities "loop constraints"; nowadays, they are commonly referred to as "subtour elimination inequalities"; we are going to call them simply *subtour inequalities*. (As for the two exceptional cuts, Dantzig, Fulkerson, and Johnson give ad hoc combinatorial arguments to show that these inequalities are satisfied by incidence vectors of all tours through the 49 cities and, in a footnote, they say "We are indebted to I. Glicksberg of Rand for pointing out relations of this kind to us".)

An important class of problems (1) are the *integer linear programming* problems, where $\mathcal{S}$ is specified as the set of all integer solutions of some explicitly recorded system of linear constraints. For this class, Gomory [26,27,28] designed fast procedures for generating cuts from the optimal simplex basis (and proved that systematic use of these cuts makes the cutting-plane method terminate); cuts generated by these procedures are called *Gomory cuts.*

If an LP relaxation of a TSP instance includes all constraints (4), (5), then a nonempty set of cuts can be found routinely whenever $x^* \notin \mathcal{S}$: on the one hand, if x^* is not an integer vector, then it violates a Gomory cut; on the other hand, if x^* is an integer vector, then it is the incidence vector of the edge-set of a disconnected graph and each connected component of this graph yields a subtour cut. The first computer code for solving the TSP by the cutting-plane method, written by Martin [46], adopts this policy: some of its cuts are subtour inequalities and others are generated by a variation on Gomory's theme described in Martin [45]. In subsequent TSP codes, subtour inequalities became a stock item, but Gomory cuts fell into disuse when a different paradigm for finding cuts took over.

By a *template,* we mean a set of linear inequalities; we say that a cut *matches the template* if it belongs to the set. By the *template paradigm*, we mean the following two-part procedure used in the design of branch-and-cut algorithms:

(i) describe one or more templates of linear inequalities that are satisfied by all the points of $\mathcal{S}$,
(ii) for each template described in part (i), design an efficient *separation algorithm* that, given an x^*, attempts to find a cut that matches the template.

The separation algorithms in (ii) may be *exact* in the sense of finding a cut that separates x^* from $\mathcal{S}$ and matches the template whenever one exists and they may be *heuristic* in the sense of sometimes failing to find such a cut even though one exists.

The primordial template of TSP cuts is the set of subtour inequalities; an exact separation algorithm for this template has been pointed out by Hong [39]. Next came the template of "blossom inequalities", introduced by Edmonds [21] in the context of 2-matchings and used in the branch-and-cut TSP computer code written by Hong [39]; then came the more general template of "comb inequalities", first used by Grötschel [30,31] in his solution of a 120-city TSP instance by the cutting-plane method. To describe these templates, let us define, for every vector x with components subscripted by edges of the complete graph on V and for every pair A, B of disjoint subsets of V,

$$x(A,B) = \sum(x_e : \ e \cap A \neq \emptyset,\ e \cap B \neq \emptyset).$$

In this notation, subtour inequalities are recorded as $x(Q, V-Q) \geq 2$; a *comb inequality* is any inequality

$$x(H, V-H) + \sum_{i=1}^{k} x(T_i, V-T_i) \geq 3k+1, \tag{6}$$

where

- $H, T_1, T_2, \ldots T_k$ are subsets of V,
- $T_1, T_2, \ldots T_k$ are pairwise disjoint,
- $T_i \cap H \neq \emptyset$ and $T_i - H \neq \emptyset$ for all $i = 1, 2, \ldots k$,
- k is odd. To see that every comb inequality is satisfied by all characteristic vectors x of tours through V, let j denote the number of sets T_i that satisfy $x(T_i, V - T_i) = 2$. In this notation, $x(H, V - H) \geq j$; furthermore, if $j = k$, then $x(H, V-H) \geq k+1$ as $x(H, V-H)$ is even and k is odd; since each $x(T_i, V-T_i)$ is a positive even integer, we have $\sum_{i=1}^{k} x(T_i, V-T_i) \geq 2j+4(k-j)$ and (6) follows. A *blossom inequality* is a comb inequality with $|T_i| = 2$ for all $i = 1, 2, \ldots k$.

Just like the subtour inequalities, the blossom inequalities, and the comb inequalities, more complicated templates of linear inequalities that are satisfied by all characteristic vectors x of tours through V are often described as *hypergraph inequalities*

$$\sum(\lambda_Q x(Q, V - Q) : Q \in \mathcal{H}) \geq \mu$$

where $\mathcal{H}$ is a collection of subsets of V (also known as a *hypergraph* on V) and λ_Q $(Q \in \mathcal{H})$ and μ are integers. Naddef [51] reviews a number of templates of hypergraph inequalities satisfied by all characteristic vectors x of tours through V.

TSP codes of Miliotis [49], Land [41], and Fleischmann [23] used the template paradigm as their preferred source of cuts; whenever this source dried up, they provisionally switched to Gomory cuts. Grötschel and Padberg [35,36] and Padberg and Hong [58] advocated a different policy, where the template paradigm is the only source of cuts; furthermore, they argued for drawing the templates exclusively from the set of linear inequalities that induce facets of the *traveling salesman polytope*, meaning the convex hull of $\mathcal{S}$. These policies were adopted in TSP codes of Crowder and Padberg [18], Grötschel and Holland [32], and Padberg and Rinaldi [60,63], which produced the most impressive computational TSP successes of the nineteen eighties. The template paradigm is also the frame of reference for other papers related to TSP codes, such as Carr [10], Christof and Reinelt [11], Clochard and Naddef [13], Cornuéjols, Fonlupt, and Naddef [14], Fleischer and Tardos [22], Grötschel and Pulleyblank [37], Letchford [43], Naddef and Thienel [54,55], Padberg and Rao [59], and Padberg and Rinaldi [61, 62].

We have written a branch-and-cut computer code for solving the TSP. Its initial version was written in 1994; we presented some of its aspects at the 15th International Symposium on Mathematical Programming held at the University of Michigan in 1994, described them in Applegate et al. [1], and eventually distributed the code as Concorde 97.08.27 at the 16th ISMP held at the Ecole polytechnique fédérale de Lausanne in 1997. A later version was written in 1997; we presented some of its aspects at the 16th ISMP, outlined them in Applegate et al. [2], and eventually made the code available on the internet, as Concorde 99.12.15, at Applegate et al. [3]. In the initial 1994 version, cuts were found partly by following the template paradigm (with a couple of our own separation heuristics for comb inequalities thrown in) and partly by a couple of new tech-

niques with the common theme of innovating obsolete cuts. In the later 1997 version, we incorporated a technique that disdains all understanding of the traveling salesman polytope and bashes on regardless of all prescribed templates. This departure from the template paradigm is the subject of the present paper.

3 Cuts, Tours, and Shrinking

Let x^* denote the optimal solution of the current LP relaxation that has been returned by the LP solver. In computer codes that search for TSP cuts, it is common practice – Land [41], Padberg and Hong [58], Crowder and Padberg [18], Padberg and Rinaldi [61,62], Grötschel and Holland [32], Naddef and Thienel [54,55] – to first reduce the size of the problem by shrinking pairwise disjoint subsets of the set V of all cities into singletons and then to look for cuts in the shrunk version of x^*.

Shrinking is an intuitive concept; to illustrate it on a toy-size example, let us begin with $V = \{0, 1, 2, \ldots, 9\}$. Shrinking each of the sets $\{0, 6, 7\}$, $\{1, 8\}$, and $\{2, 9\}$ into a single vertex – $\{0, 6, 7\}$ into 0, $\{1, 8\}$ into 1, $\{2, 9\}$ into 2 – reduces each vector x with components $x_{01}, x_{02}, \ldots, x_{89}$ (edges are two-point sets, but we prefer writing x_{ij} to writing $x_{\{i,j\}}$) to the vector $\overline{x}$ with components $\overline{x}_{01}, \overline{x}_{02}, \ldots, \overline{x}_{45}$ defined by

$$\begin{aligned}
\overline{x}_{01} &= x_{01} + x_{08} + x_{16} + x_{17} + x_{68} + x_{78},\\
\overline{x}_{02} &= x_{02} + x_{09} + x_{26} + x_{27} + x_{69} + x_{79},\\
\overline{x}_{03} &= x_{03} + x_{36} + x_{37},\\
\overline{x}_{04} &= x_{04} + x_{46} + x_{47},\\
\overline{x}_{05} &= x_{05} + x_{56} + x_{57},\\
\overline{x}_{12} &= x_{12} + x_{19} + x_{28} + x_{89},\\
\overline{x}_{13} &= x_{13} + x_{38},\\
\overline{x}_{14} &= x_{14} + x_{48},\\
\overline{x}_{15} &= x_{15} + x_{58},\\
\overline{x}_{23} &= x_{23} + x_{39},\\
\overline{x}_{24} &= x_{24} + x_{49},\\
\overline{x}_{25} &= x_{25} + x_{59},\\
\overline{x}_{34} &= x_{34},\\
\overline{x}_{35} &= x_{35},\\
\overline{x}_{45} &= x_{45}.
\end{aligned}$$

In particular, if x^* is defined by

$$\begin{aligned}
&x^*_{01} = 0.3,\ x^*_{02} = 0.3,\ x^*_{06} = 0.7,\ x^*_{07} = 0.7,\ x^*_{14} = 0.5,\ x^*_{16} = 0.2,\ x^*_{18} = 1.0,\\
&x^*_{25} = 0.5,\ x^*_{27} = 0.2,\ x^*_{29} = 1.0,\ x^*_{34} = 0.5,\ x^*_{35} = 0.5,\ x^*_{36} = 0.5,\ x^*_{37} = 0.5,\\
&\qquad x^*_{45} = 0.5,\ x^*_{48} = 0.5,\ x^*_{59} = 0.5,\ x^*_{67} = 0.6,\ x^*_{89} = 0.5,
\end{aligned}$$

and $x^*_{ij} = 0$ for all the remaining i and j such that $0 \le i < j \le 9$, then its shrunk version $\overline{x}^*$ is defined by

$$\begin{array}{c} \overline{x}^*_{01} = 0.5,\ \overline{x}^*_{02} = 0.5,\ \overline{x}^*_{12} = 0.5, \\ \overline{x}^*_{03} = 1.0,\ \overline{x}^*_{14} = 1.0,\ \overline{x}^*_{25} = 1.0, \\ \overline{x}^*_{34} = 0.5,\ \overline{x}^*_{35} = 0.5,\ \overline{x}^*_{45} = 0.5, \end{array}$$

and $\overline{x}^*_{ij} = 0$ for all the remaining i and j such that $0 \le i < j \le 5$.

Looking for cuts in the shrunk version $\overline{x}^*$ of x^* does not mean looking for linear inequalities satisfied by all tours through the shrunk version $\overline{V}$ of V and violated by $\overline{x}^*$. In our toy example, the inequality

$$\overline{x}_{01} + \overline{x}_{02} + \overline{x}_{12} + \overline{x}_{03} + \overline{x}_{14} + \overline{x}_{25} \le 4$$

is satisfied by all tours through $\overline{V}$ – which is $\{0, 1, 2, \ldots, 5\}$ – and violated by $\overline{x}^*$; through substitution from the definitions of $\overline{x}_{ij}$, it yields the inequality

$$\begin{array}{rl} & x_{01} + x_{08} + x_{16} + x_{17} + x_{68} + x_{78} \\ + & x_{02} + x_{09} + x_{26} + x_{27} + x_{69} + x_{79} \\ + & x_{12} + x_{19} + x_{28} + x_{89} \\ + & x_{03} + x_{36} + x_{37} \\ + & x_{14} + x_{48} \\ + & x_{25} + x_{59} \le 4, \end{array}$$

which is not satisfied by the tour 0-2-5-9-1-4-8-7-6-3-0.

In this example, shrinking $\{0, 6, 7\}$ into 0, shrinking $\{1, 8\}$ into 1, and shrinking $\{2, 9\}$ into 2 reduces the tour 0-2-5-9-1-4-8-7-6-3-0 to the spanning closed walk 0-2-5-2-1-4-1-0-3-0; it reduces the incidence vector x of the tour to the vector $\overline{x}$ such that

$$\overline{x}_{01} = 1,\ \overline{x}_{02} = 1,\ \overline{x}_{03} = 2,\ \overline{x}_{12} = 1,\ \overline{x}_{14} = 2,\ \overline{x}_{25} = 2,$$

and $\overline{x}_{ij} = 0$ for all the remaining i and j such that $0 \le i < j \le 5$. In general, shrinking V into $\overline{V}$ reduces a tour through V to a spanning closed walk through $\overline{V}$; it reduces the incidence vector x of the tour to a vector $\overline{x}$ such that

- each $\overline{x}_e$ is a nonnegative integer,
- the graph with vertex-set $\overline{V}$ and edge-set $\{e : \overline{x}_e > 0\}$ is connected,
- $\sum(\overline{x}_e : v \in e)$ is even whenever $v \in \overline{V}$;

we will refer to the set of all the vectors $\overline{x}$ with these properties as *tangled tours* through $\overline{V}$. This notion, but not the term, was introduced by Cornuéjols, Fonlupt, and Naddef [14]; they refer to the convex hull of the set of all tangled tours through a prescribed set as the *graphical traveling salesman polytope*. Every inequality $\sum \overline{a}_{ij}\overline{x}_{ij} \le b$ yields, through substitution from the definitions of $\overline{x}_{ij}$, an inequality $\sum a_{ij}x_{ij} \le b$; if the original inequality is satisfied by all tangled tours through $\overline{V}$, then the new inequality is satisfied by all tours through V; if

the original inequality is violated by the shrunk version $\overline{x}^*$ of x^*, then the new inequality is violated by x^*. In our toy example, the blossom inequality

$$\begin{aligned}&\overline{x}(\{0,1,2\},\{3,4,5\})\\+&\overline{x}(\{0,3\},\{1,2,4,5\})\\+&\overline{x}(\{1,4\},\{0,2,3,5\})\\+&\overline{x}(\{2,5\},\{0,1,3,4\})\geq 10\end{aligned}$$

is satisfied by all tangled tours through $\overline{V}$ and violated by $\overline{x}^*$; it yields, through substitution from the definitions of $\overline{x}_{ij}$, the comb inequality

$$\begin{aligned}&x(\{0,1,2,6,7,8,9\},\{3,4,5\})\\+&x(\{0,3,6,7\},\{1,2,4,5,8,9\})\\+&x(\{1,4,8\},\{0,2,3,5,6,7,9\})\\+&x(\{2,5,9\},\{0,1,3,4,6,7,8\})\geq 10\,,\end{aligned}$$

which is satisfied by all tours through V and violated by x^*.

As this example suggests, the change of variable from $\overline{x}$ to x is particularly easy to implement in hypergraph inequalities: substitution from the definitions of $\overline{x}_{ij}$ converts each linear function $\overline{x}(\overline{Q},\overline{V}-\overline{Q})$ to the linear function $x(Q,V-Q)$ where Q is the set of all cities that are mapped into $\overline{Q}$ by the function that shrinks V onto $\overline{V}$.

The subject of the present paper is a technique where, rather than first shrinking x^* once and for all onto a $\overline{V}$ that may be large and then attempting to find many cuts in the single $\overline{x}^*$, we shrink x^* many times onto sets $\overline{V}$ that are small (their typical size is at most thirty or so) and we find precisely one cut in each $\overline{x}^*$ that lies outside the graphical traveling salesman polytope on $\overline{V}$: see Algorithm 31.

The forthcoming Sect. 4 takes up about half of the paper and describes the collection of commonplace techniques that we have chosen to implement the body of the **for** loop in Algorithm 31 in Concorde, our computer code (a more detailed description will be presented in Applegate et al. [4]); the very short Sect. 5 describes Concorde's implementation of the control of the **for** loop. Section 6 attempts to give an idea of how useful the inclusion of Algorithm 31 in Concorde turned out to be. Section 7 places Algorithm 31 in a broader context and points out the potential for more general uses of the machinery assembled in Sect. 4.

Algorithm 31. *A scheme for collecting TSP cuts*

initialize an empty list $\mathcal{L}$ of cuts;
for selected small integers k and
partitions of V into nonempty sets $V_0, V_1, \ldots, V_k$
do $\overline{x}^*$ = the vector obtained from x^* by shrinking each V_i into singleton i;
$\overline{V} = \{0, 1, \ldots, k\}$;

if $\overline{x}^*$ lies outside the graphical traveling salesman polytope on $\overline{V}$
then find a hypergraph inequality that is
satisfied by all tangled tours through $\overline{V}$ and violated by $\overline{x}^*$;
change its variable from $\overline{x}$ to x, and
add the resulting hypergraph inequality to $\mathcal{L}$;
end
end
return $\mathcal{L}$;

4 Processing $\overline{x}^*$

We will illustrate Concorde's implementation of the body of the **for** loop in Algorithm 31 on the example where $\overline{V} = \{0, 1, \ldots, 7\}$ and

$$\begin{array}{l}\overline{x}^*_{01} = 0.7, \overline{x}^*_{02} = 0.1, \overline{x}^*_{04} = 0.9, \overline{x}^*_{07} = 0.9, \overline{x}^*_{12} = 0.7,\\ \overline{x}^*_{15} = 0.6, \overline{x}^*_{23} = 0.8, \overline{x}^*_{25} = 0.4, \overline{x}^*_{34} = 1.0, \overline{x}^*_{35} = 0.1,\\ \overline{x}^*_{36} = 0.1, \overline{x}^*_{47} = 0.1, \overline{x}^*_{56} = 0.9, \overline{x}^*_{67} = 1.0,\end{array}$$

and $\overline{x}^*_{ij} = 0$ for all the remaining i and j such that $0 \le i < j \le 7$.

4.1 Does $\overline{x}^*$ Lie Outside the Graphical Traveling Salesman Polytope?

Rephrasing the Question. In our example, every tangled tour $\overline{x}$ through $\overline{V}$ satisfies the fourteen inequalities

$$\begin{array}{l}\overline{x}_{03} \ge 0, \overline{x}_{05} \ge 0, \overline{x}_{06} \ge 0, \overline{x}_{13} \ge 0, \overline{x}_{14} \ge 0,\\ \overline{x}_{16} \ge 0, \overline{x}_{17} \ge 0, \overline{x}_{24} \ge 0, \overline{x}_{26} \ge 0, \overline{x}_{27} \ge 0,\\ \overline{x}_{37} \ge 0, \overline{x}_{45} \ge 0, \overline{x}_{46} \ge 0, \overline{x}_{57} \ge 0\end{array}$$

and the seven inequalities

$$\begin{array}{l}\overline{x}_{01} + \overline{x}_{12} + \overline{x}_{13} + \overline{x}_{14} + \overline{x}_{15} + \overline{x}_{16} + \overline{x}_{17} \ge 2,\\ \overline{x}_{02} + \overline{x}_{12} + \overline{x}_{23} + \overline{x}_{24} + \overline{x}_{25} + \overline{x}_{26} + \overline{x}_{27} \ge 2,\\ \overline{x}_{03} + \overline{x}_{13} + \overline{x}_{23} + \overline{x}_{34} + \overline{x}_{35} + \overline{x}_{36} + \overline{x}_{37} \ge 2,\\ \overline{x}_{04} + \overline{x}_{14} + \overline{x}_{24} + \overline{x}_{34} + \overline{x}_{45} + \overline{x}_{46} + \overline{x}_{47} \ge 2,\\ \overline{x}_{05} + \overline{x}_{15} + \overline{x}_{25} + \overline{x}_{35} + \overline{x}_{45} + \overline{x}_{56} + \overline{x}_{57} \ge 2,\\ \overline{x}_{06} + \overline{x}_{16} + \overline{x}_{26} + \overline{x}_{36} + \overline{x}_{46} + \overline{x}_{56} + \overline{x}_{67} \ge 2,\\ \overline{x}_{07} + \overline{x}_{17} + \overline{x}_{27} + \overline{x}_{37} + \overline{x}_{47} + \overline{x}_{57} + \overline{x}_{67} \ge 2\end{array}$$

and the two inequalities

$$\begin{array}{ll}\overline{x}_{03} + \overline{x}_{13} + \overline{x}_{23} + \overline{x}_{35} + \overline{x}_{36} + \overline{x}_{37} + & \\ \overline{x}_{04} + \overline{x}_{14} + \overline{x}_{24} + \overline{x}_{45} + \overline{x}_{46} + \overline{x}_{47} & \ge 2,\\ \overline{x}_{06} + \overline{x}_{16} + \overline{x}_{26} + \overline{x}_{36} + \overline{x}_{46} + \overline{x}_{56} + & \\ \overline{x}_{07} + \overline{x}_{17} + \overline{x}_{27} + \overline{x}_{37} + \overline{x}_{47} + \overline{x}_{57} & \ge 2;\end{array}$$

since $\overline{x}^*$ satisfies each of these twenty-three inequalities as an equation, it is a convex combination of tangled tours through $\overline{V}$ if only if it is a convex combination of those tangled tours through $\overline{V}$ that satisfy each of the twenty-three inequalities as an equation.

In general, every tangled tour $\overline{x}$ through $\overline{V}$ satisfies the inequalities

$$\begin{aligned} \overline{x}_e &\geq 0 \quad \text{for all } e, \\ \sum(\overline{x}_e : u \in e) &\geq 2 \quad \text{for all } u, \\ \overline{x}(e, \overline{V} - e\} &\geq 2 \quad \text{for all } e, \end{aligned}$$

and so $\overline{x}^*$ is a convex combination of tangled tours through $\overline{V}$ if only if it is a convex combination of tangled tours through $\overline{V}$ that satisfy

$$\overline{x}_e = 0 \quad \text{for all } e \text{ such that } \overline{x}^*_e = 0, \tag{7}$$

$$\sum(\overline{x}_e : u \in e) = 2 \quad \text{for all } u \text{ such that } \sum(\overline{x}^*_e : u \in e) = 2, \tag{8}$$

$$\overline{x}(e, \overline{V} - e) = 2 \quad \text{for all } e \text{ such that } \overline{x}^*(e, \overline{V} - e\} = 2. \tag{9}$$

Concorde chooses $V_0, V_1, \ldots, V_k$ so that $x(V_i, V - V_i) = 2$ for all $i = 1, 2, \ldots, k$ (but not necessarily for $i = 0$), and so

$$\sum(\overline{x}^*_e : i \in e) = 2 \text{ for all } i \text{ in } \{1, 2, \ldots, k\}.$$

(Our machinery for processing $\overline{x}^*$ is predicated on this property of $\overline{x}^*$, but could be easily modified to handle all $\overline{x}^*$: see Sect. 7.) In this case, every solution $\overline{x}$ of (8), (9) satisfies

$$\sum(\overline{x}_e : u \in e) = 2 \quad \text{for all } u \text{ in } \{1, 2, \ldots, k\}, \tag{10}$$

$$\overline{x}_e = 1 \quad \text{for all } e \text{ such that } e \subset \{1, 2, \ldots, k\} \text{ and } \overline{x}^*_e = 1, \tag{11}$$

and so

> $\overline{x}^*$ is a convex combination of tangled tours through $\overline{V}$
> if only if it is a convex combination of tangled tours through $\overline{V}$
> that satisfy (7), (10), (11).

We will refer to tangled tours that satisfy (7), (10), (11) as *strongly constrained*.

Delayed Generation of Strongly Constrained Tangled Tours. Unfortunately, the strongly constrained tangled tours through $\overline{V}$ may be too numerous to be listed one by one; fortunately, $\overline{x}^*$ can be tested for membership in the convex hull of the set of strongly constrained tangled tours through $\overline{V}$ without listing these tangled tours one by one. The trick, introduced by Ford and Fulkerson [25] and by Jewell [40], is known as *delayed column generation*; we use it in Algorithm 41. This algorithm returns either the message "$\overline{x}^*$ is in the graphical traveling salesman polytope on $\overline{V}$" or a vector a and a scalar b such that the inequality $a^T\overline{x} \leq b$ is satisfied by all strongly constrained tangled tours $\overline{x}$ through $\overline{V}$ and violated by $\overline{x}^*$.

Algorithm 41. *Testing the* **if** *condition in Algorithm 31:*

if there is a strongly constrained tangled tour $\overline{x}$ through $\overline{V}$
then make $\overline{x}$ the only specimen in a collection of
strongly constrained tangled tours through $\overline{V}$;
repeat if some linear inequality $a^T\overline{x} \leq b$ is
satisfied by all $\overline{x}$ in the collection and violated by $\overline{x}^*$
then find a strongly constrained tangled tour $\overline{x}$ through $\overline{V}$
that maximizes $a^T\overline{x}$;
if $a^T\overline{x} \leq b$
then return a and b;
else add $\overline{x}$ to the collection;
end
else return the message
"$\overline{x}^*$ is in the graphical traveling salesman polytope on $\overline{V}$";
end
end
else return 0 and -1;
end

In our illustrative example, Algorithm 41 may initialize the collection by the
• strongly constrained tangled tour 0-4-3-2-1-5-6-7-0
and then proceed through the following five iterations:

Iteration 1: Inequality

$$-\overline{x}_{15} \leq -1$$

is satisfied by all $\overline{x}$ in the collection and violated by $\overline{x}^*$. The
• strongly constrained tangled tour 0-1-2-0-4-3-5-6-7-0
maximixes $-\overline{x}_{15}$. We add this tangled tour to the collection.

Iteration 2: Inequality

$$\overline{x}_{25} \leq 0$$

is satisfied by all $\overline{x}$ in the collection and violated by $\overline{x}^*$. The
• strongly constrained tangled tour 0-1-5-2-0-4-3-6-7-0
maximixes $\overline{x}_{25}$. We add this tangled tour to the collection.

Iteration 3: Inequality

$$-\overline{x}_{15} + \overline{x}_{23} + \overline{x}_{25} \leq 0$$

is satisfied by all $\overline{x}$ in the collection and violated by $\overline{x}^*$. The
• strongly constrained tangled tour 0-1-0-4-3-2-5-6-7-0
maximixes $-\overline{x}_{15} + \overline{x}_{23} + \overline{x}_{25}$. We add this tangled tour to the collection.

Iteration 4: Inequality

$$\overline{x}_{47} \leq 0$$

is satisfied by all $\overline{x}$ in the collection and violated by $\overline{x}^*$. The

• strongly constrained tangled tour 0-1-5-6-7-4-3-2-0

maximixes $\overline{x}_{47}$. We add this tangled tour to the collection.

ITERATION 5: Inequality

$$\overline{x}_{12} + \overline{x}_{25} + \overline{x}_{47} \leq 1 \tag{12}$$

is satisfied by all $\overline{x}$ in the collection and violated by $\overline{x}^*$. The

• strongly constrained tangled tour 0-4-3-2-1-5-6-7-0

maximixes $\overline{x}_{12} + \overline{x}_{25} + \overline{x}_{47}$. We conclude that (12) is satisfied by all strongly constrained tangled tours and violated by $\overline{x}^*$.

Implementing Algorithm 41. Let

$\overline{E}_{1/2}$ denote the set of all the edges e such that
$e \subset \{1, 2, \ldots, k\}$, $\overline{x}^*_e \neq 0$, $\overline{x}^*_e \neq 1$.

The significance of $\overline{E}_{1/2}$ comes from the fact that every strongly constrained tangled tour $\overline{x}$ satisfies

$$\begin{aligned}
\overline{x}_{0u} &= 2 - \textstyle\sum(\overline{x}_e : e \subset \{1, 2, \ldots, k\}, u \in e) \quad \text{for all } u \text{ in } \{1, 2, \ldots, k\},\\
\overline{x}_e &= 0 \quad \text{for all } e \text{ such that } e \subset \{1, 2, \ldots, k\} \text{ and } \overline{x}^*_e = 0,\\
\overline{x}_e &= 1 \quad \text{for all } e \text{ such that } e \subset \{1, 2, \ldots, k\} \text{ and } \overline{x}^*_e = 1,
\end{aligned}$$

and so the condition

some linear inequality $a^T\overline{x} \leq b$ is
satisfied by all $\overline{x}$ in the collection and violated by $\overline{x}^*$

in Algorithm 41 is equivalent to the condition

some linear inequality $a^T\overline{x} \leq b$ with $a_e = 0$ whenever $e \notin \overline{E}_{1/2}$ is
satisfied by all $\overline{x}$ in the collection and violated by $\overline{x}^*$.

To test this condition, Concorde solves a linear programming problem. With

$\psi(\overline{x})$ standing for the restriction of $\overline{x}$
on its components indexed by elements of $\overline{E}_{1/2}$,

with A the matrix whose columns $\psi(\overline{x})$ come from specimens $\overline{x}$ in the collection, and with e standing – as usual – for the vector $[1, 1, \ldots, 1]^T$ whose number of components is determined by the context, this problem in variables s, λ, w reads

$$\begin{aligned}
&\text{maximize } s\\
&\text{subject to } \quad s\psi(\overline{x}^*) - A\lambda + w \quad = 0,\\
&\qquad\qquad\quad -s + e^T\lambda \qquad\quad = 0,\\
&\qquad\qquad\quad \lambda \geq 0, \quad -e \leq w \leq e.
\end{aligned} \tag{13}$$

Since its constraints can be satisfied by setting $s = 0$, $\lambda = 0$, $w = 0$, problem (13) either has an optimal solution or else it is unbounded. In the former case, simplex method applied to (13) finds also an optimal solution of its dual,

$$\begin{aligned} \text{minimize} \quad & e^T(u+v) \\ \text{subject to} \quad & -a^T A + be^T \geq 0, \\ & a^T \psi(\overline{x}^*) - b = 1, \\ & a + u - v = 0, \\ & u \geq 0, \quad v \geq 0, \end{aligned} \tag{14}$$

and this optimal solution provides the a and b of Algorithm 41; in fact, it

$$\text{maximizes} \quad \frac{a^T \overline{x}^* - b}{\|a\|_1}$$

subject to the constraints that $a^T \overline{x} \leq b$ for all $\overline{x}$ in the collection and that $a_e = 0$ whenever $e \notin \overline{E}_{1/2}$. In the latter case, (14) is infeasible, and so no linear inequality is satisfied by all $\overline{x}$ in the collection and violated by $\overline{x}^*$.

To find specimens $\overline{x}$ for the collection, we use a function ORACLE that, given an integer vector c, returns either a strongly constrained tangled tour $\overline{x}$ through $\overline{V}$ that maximizes $c^T \overline{x}$ or the message "infeasible" indicating that no tangled tour through $\overline{V}$ is strongly constrained. Concorde implements ORACLE as two algorithms in tandem: if a primitive branch-and-bound algorithm fails to solve the instance within a prescribed number of recursive calls of itself, then we switch to a more sophisticated branch-and-cut algorithm. To reconcile

- the integer arithmetic of ORACLE, which uses a and b,

with

- the floating-point arithmetic of the simplex method, which finds a and b,

Concorde uses the continued fraction method (see, for instance, Schrijver [66]); since ORACLE uses integer arithmetic, the cut $a^T \overline{x} \leq b$ that separates $\overline{x}^*$ from all strongly constrained tangled tours through $\overline{V}$ has an integer a and an integer right-hand side b.

4.2 Separating $\overline{x}^*$ from the Graphical Traveling Salesman Polytope: The Three Phases

If $\overline{x}^*$ lies outside the graphical traveling salesman polytope, then Algorithm 41 returns a linear inequality that separates $\overline{x}^*$ from all strongly constrained tangled tours. To convert this inequality to a cut that separates $\overline{x}^*$ from all tangled tours, we proceed in in three phases:

in PHASE 1, we find a linear inequality that
separates $\overline{x}^*$ from all *moderately constrained* tangled tours and
induces a facet of the convex hull of these tangled tours,
in PHASE 2, we find a linear inequality that
separates $\overline{x}^*$ from all *weakly constrained* tangled tours and
induces a facet of the convex hull of these tangled tours,

in PHASE 3, we find a linear inequality that
separates $\overline{x}^*$ from all tangled tours and
induces a facet of the convex hull of weakly constrained tangled tours.

(In PHASE 3, we could easily find a linear inequality that separates $\overline{x}^*$ from all tangled tours and induces a facet of the convex hull of all tangled tours; however, such an inequality might not be acceptable to Concorde as a constraint of an LP relaxation of the TSP instance. We will discuss this point in Sect. 4.5.)

The intermediate classes of moderately constrained tangled tours and weakly constrained tangled tours are defined by reference to the sets of constraints

(a) $\sum(\overline{x}_e : u \in e) = 2$ for all u in $\{1, 2, \ldots, k\}$;
(b) $\overline{x}_e = 0$ for all e such that $e \subset \{1, 2, \ldots, k\}$ and $\overline{x}^*_e = 0$,
$\overline{x}_e = 1$ for all e such that $e \subset \{1, 2, \ldots, k\}$ and $\overline{x}^*_e = 1$;
(c) $\overline{x}_{0u} = 0$ for all u in $\{1, 2, \ldots, k\}$ such that $\overline{x}^*_{0u} = 0$.

Specifically,

strongly constrained tangled tours are those that satisfy (a), (b), (c);
moderately constrained tangled tours are those that satisfy (a), (b);
weakly constrained tangled tours are those that satisfy (a).

The principal tools used in PHASE 1 are a constructive proof of a a classic result of Minkowski (Theorem 1) and the Dinkelbach method of fractional programming; PHASE 2 is sequential lifting; PHASE 3 is nearly effortless. In PHASE 1 and PHASE 2, we use a function ORACLE that,

given integer vectors c, ℓ, u and a threshold t (either an integer or $-\infty$),
returns either
a weakly constrained tangled tour $\overline{x}$ that maximizes $c^T\overline{x}$
subject to $\ell \le \overline{x} \le u$, $c^T\overline{x} > t$
or
the message "infeasible" indicating that
no weakly constrained tangled tour $\overline{x}$ satisfies
$\ell \le \overline{x} \le u$, $c^T\overline{x} > t$.

This is the same function that is used, with a fixed ℓ and a fixed u, to find items $\overline{x}$ for the collection in Algorithm 41.

4.3 PHASE 1: From Strongly Constrained to Moderately Constrained Tangled Tours

Separating $\overline{x}^*$ from Moderately Constrained Tangled Tours. It is easy to convert any linear inequality that separates $\overline{x}^*$ from all strongly constrained tangled tours to a linear inequality that separates $\overline{x}^*$ from all moderately constrained tangled tours. In our example, inequality

$$\overline{x}_{12} + \overline{x}_{25} + \overline{x}_{47} \le 1$$

is satisfied by all strongly constrained tangled tours $\overline{x}$ and violated by $\overline{x}^*$; if $\overline{x}$ is a moderately constrained tangled tour, then $\psi(\overline{x})$ is a zero-one vector, and so

$$\overline{x}_{12} + \overline{x}_{25} + \overline{x}_{47} \leq 3;$$

a moderately constrained tangled tour $\overline{x}$ is strongly constrained if and only if

$$\overline{x}_{03} = 0,\ \overline{x}_{05} = 0,\ \overline{x}_{06} = 0.$$

It follows that the inequality

$$\overline{x}_{12} + \overline{x}_{25} + \overline{x}_{47} - 2(\overline{x}_{03} + \overline{x}_{05} + \overline{x}_{06}) \leq 1$$

is satisfied by all moderately constrained tangled tours $\overline{x}$ and violated by $\overline{x}^*$. In general, a moderately constrained tangled tour $\overline{x}$ is strongly constrained if and only if

$$\overline{x}_{0u} = 0 \text{ for all } u \text{ in } \{1, 2, \ldots, k\} \text{ such that } \overline{x}^*_{0u} = 0;$$

if

$$p^T \psi(\overline{x}) \leq r$$

is satisfied by all strongly constrained tangled tours $\overline{x}$ and violated by $\overline{x}^*$, then

$$p^T \psi(\overline{x}) - (||p||_1 - r) \sum(\overline{x}_{0u} : \overline{x}^*_{0u} = 0) \leq r \tag{15}$$

is satisfied by all moderately constrained tangled tours $\overline{x}$ and violated by $\overline{x}^*$ (to see that $||p||_1 - r$ is always positive, note that $r < p^T \psi(\overline{x}^*) \leq ||p||_1$); the bulk of PHASE 1 is taken up by transforming this cut into a cut that induces a facet of the convex hull of all moderately constrained tangled tours.

Dimension of the Set of Moderately Constrained Tangled Tours. In Sect. 4.1, we defined

$$\overline{E}_{1/2} = \{e :\ e \subset \{1, 2, \ldots, k\},\ \overline{x}^*_e \neq 0,\ \overline{x}^*_e \neq 1\}.$$

In fact, we may assume that

$$\overline{E}_{1/2} = \{e : e \subset \{1, 2, \ldots, k\},\ 0 < \overline{x}^*_e < 1\} :$$

otherwise $\overline{x}^*_e > 1$ for some e such that $e \subset \{1, 2, \ldots, k\}$, and so $\overline{x}^*$ is separated from all tangled tours by the subtour inequality $\overline{x}(e, \overline{V} - e) \geq 2$. (As noted by Cornuéjols, Fonlupt, and Naddef [14], subtour inequalities induce facets of the graphical traveling salesman polytope.) In addition,

with $\psi(\overline{x})$ standing for the restriction of $\overline{x}$ on $\overline{E}_{1/2}$ just as in Sect. 4.1,

we may assume that each of the vectors

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

with components indexed by elements of $\overline{E}_{1/2}$ equals $\psi(\overline{x})$ for some moderately constrained tangled tour $\overline{x}$ (else $\overline{x}^*$ is separated from $\mathcal{T}$ by another readily available subtour inequality), and so

$\{\psi(\overline{x}) : \overline{x}$ is a moderately constrained tangled tour$\}$ has full dimension.

Since every moderately constrained tangled tour $\overline{x}$ satisfies

$$\begin{aligned}
\overline{x}_{0u} &= 2 - \textstyle\sum(\overline{x}_e : e \subset \{1, 2, \ldots, k\}, u \in e) \quad \text{for all } u \text{ in } \{1, 2, \ldots, k\}, \\
\overline{x}_e &= 0 \quad \text{for all } e \text{ such that } e \subset \{1, 2, \ldots, k\} \text{ and } \overline{x}^*_e = 0, \\
\overline{x}_e &= 1 \quad \text{for all } e \text{ such that } e \subset \{1, 2, \ldots, k\} \text{ and } \overline{x}^*_e = 1,
\end{aligned}$$

we conclude that

the set of moderately constrained tangled tours has dimension $|\overline{E}_{1/2}|$.

From a Cut to a Facet-Inducing Cut: An Overview. The optimal basis of problem (13) provides an affinely independent set $\mathcal{I}$ of strongly constrained tangled tours $\overline{x}$ that satisfy (15) with the sign of equality. In our illustrative example, $\mathcal{I}$ consists of

- the strongly constrained tangled tour 0-4-3-2-1-5-6-7-0,
- the strongly constrained tangled tour 0-1-2-0-4-3-5-6-7-0,
- the strongly constrained tangled tour 0-1-5-2-0-4-3-6-7-0,
- the strongly constrained tangled tour 0-1-0-4-3-2-5-6-7-0,
- the strongly constrained tangled tour 0-1-5-6-7-4-3-2-0.

In general, it may turn out that $\mathcal{I} = \emptyset$; this happens if and only if no tangled tour through $\overline{V}$ is strongly constrained, in which case (15) reads

$$-\textstyle\sum(\overline{x}_{0u} : \overline{x}^*_{0u} = 0) \leq -1.$$

Writing $a^T\overline{x} \leq b$ for (15), we have an integer vector a, an integer b, and a (possibly empty) set $\mathcal{I}$ such that

(Inv1) all moderately constrained tangled tours $\overline{x}$ have $a^T\overline{x} \leq b$,
(Inv2) $\mathcal{I}$ is an affinely independent set
of moderately constrained tangled tours,
(Inv3) $a^T\overline{x} = b$ whenever $\overline{x} \in \mathcal{I}$,
(Inv4) $a^T\overline{x}^* > b$.

Concorde maintains these four invariants while adding new elements to $\mathcal{I}$ and adjusting a and b if necessary: when $|\mathcal{I}|$ reaches $|\overline{E}_{1/2}|$, the current cut $a^T\overline{x} \leq b$ induces a facet of the convex hull of all moderately constrained tangled tours. An outline of this process is provided in Algorithm 42.

Algorithm 42. *From a cut to a facet-inducing cut in Phase 1*:

while $|\mathcal{I}| < |\overline{E}_{1/2}|$
do find an integer vector v, an integer w, and
a moderately constrained tangled tour $\overline{x}^0$ such that
(A1) $v^T\overline{x} = w$ whenever $\overline{x} \in \mathcal{I}$,
(A2) some moderately constrained tangled tour $\overline{x}$ has $v^T\overline{x} \neq w$,
and either
(A3.1) $v^T\overline{x}^* \geq w$ and $v^T\overline{x} \geq w$ for all moderately constrained tangled tours
or else
(A3.2) $a^T\overline{x}^0 < b$ and $v^T\overline{x}^0 = w$;
find an integer vector a', an integer b', and
a moderately constrained tangled tour $\overline{x}'$ such that
(B1) all moderately constrained tangled tours $\overline{x}$ have ${a'}^T\overline{x} \leq b'$,
(B2) equation ${a'}^T\overline{x} = b'$ is a linear combination of $a^T\overline{x} = b$ and $v^T\overline{x} = w$,
(B3) ${a'}^T\overline{x}' = b'$ and $(a^T\overline{x}', v^T\overline{x}') \neq (b, w)$,
(B4) ${a'}^T\overline{x}^* > b'$;
$a = a'$, $b = b'$, $\mathcal{I} = \mathcal{I} \cup \{\overline{x}'\}$;
end
return a and b;

In our illustrative example, the initial $a^T x \leq b$ reads

$$-2\overline{x}_{03} - 2\overline{x}_{05} - 2\overline{x}_{06} + \overline{x}_{12} + \overline{x}_{25} + \overline{x}_{47} \leq 1$$

and Algorithm 42 may proceed as follows.

Iteration 1: $v^T\overline{x} = \overline{x}_{03}$, $w = 0$, and $\overline{x}^0$ is an arbitrary moderately constrained tangled tour. We leave $a^T\overline{x} \leq b$ unchanged and we add to $\mathcal{I}$

- the moderately constrained tangled tour 0-1-2-5-6-7-4-3-0.

Iteration 2: $v^T\overline{x} = \overline{x}_{05}$, $w = 0$, and $\overline{x}^0$ is an arbitrary moderately constrained tangled tour. We replace $a^T\overline{x} \leq b$ by

$$-2\overline{x}_{03} - \overline{x}_{05} - 2\overline{x}_{06} + \overline{x}_{12} + \overline{x}_{25} + \overline{x}_{47} \leq 1 \tag{16}$$

and we add to $\mathcal{I}$

- the moderately constrained tangled tour 0-1-2-5-0-4-3-6-7-0.

Iteration 3: $v^T\overline{x} = \overline{x}_{06}$, $w = 0$, and $\overline{x}^0$ is an arbitrary moderately constrained tangled tour. We leave $a^T\overline{x} \leq b$ unchanged and we add to $\mathcal{I}$

- the moderately constrained tangled tour 0-1-2-5-3-4-7-6-0.

Now $|\mathcal{I}|$ has reached $|\overline{E}_{1/2}|$, and so (16) induces a facet of the convex hull of moderately constrained tangled tours.

Finding v, w, and $\overline{x}^0$ in Algorithm 42. In early iterations of the **while** loop in Algorithm 42, Concorde draws v and w from a catalog of inequalities $v^T\overline{x} \geq w$ that satisfy (A3.1) and (A2); any of these inequalities that happens to satisfy (A1) can provide the v and the w for use, with an arbitrary moderately constrained tangled tour $\overline{x}^0$, in the next iteration of the **while** loop. The catalog consists of

$$\begin{array}{lll} \text{all inequalities} & \overline{x}_e \geq 0 & \text{such that } e \in \overline{E}_{1/2}, \\ \text{all inequalities} & -\overline{x}_e \geq -1 & \text{such that } e \in \overline{E}_{1/2}, \\ \text{all inequalities} & \overline{x}_{0u} \geq 0 & \text{such that } u \in \{1, 2, \ldots, k\}; \end{array}$$

for all u in $\{1, 2, \ldots, k\}$, Concorde's way of shrinking x^* into $\overline{x}^*$ – whose description is deferred to Sect. 5 – guarantees the existence of a moderately constrained tangled tour $\overline{x}$ such that $\overline{x}_{0u} > 0$.

If no inequality $v^T\overline{x} \geq w$ in the catalog satisfies (A1) and yet $|\mathcal{I}| < |\overline{E}_{1/2}|$, then Concorde finds v as a nonzero solution of the system

$$v^T\overline{x} = 0 \text{ for all } \overline{x} \text{ in } \mathcal{I}, \tag{17}$$

$$v_e = 0 \text{ for all } e \text{ outside } \overline{E}_{1/2}, \tag{18}$$

it sets $w = 0$, and it lets $\overline{x}^0$ be the moderately constrained tangled tour such that $\overline{x}^0_e = 0$ for all e in $\overline{E}_{1/2}$.

For each e in $\overline{E}_{1/2}$, let $\overline{x}^e$ denote the moderately constrained tangled tour such that $\overline{x}^e_e = 1$ and $\overline{x}^e_f = 0$ for all other f in $\overline{E}_{1/2}$. Property (18) guarantees that $v^T\overline{x}^e = v_e$ for all e in $\overline{E}_{1/2}$; since v is a nonzero vector with property (18), at least one e in $\overline{E}_{1/2}$ has $v_e \neq 0$; it follows that (A2) holds.

To see that $a^T\overline{x}^0 < b$, observe that

$$\sum(\overline{x}^*_e\overline{x}^e : e \in \overline{E}_{1/2}) + \big(1 - \sum(\overline{x}^*_e : e \in \overline{E}_{1/2})\big)\,\overline{x}^0 = \overline{x}^*,$$

and so

$$\begin{array}{rcl} \big(1 - \sum(\overline{x}^*_e : e \in \overline{E}_{1/2})\big)\,a^T\overline{x}^0 & = & a^T\overline{x}^* - \sum(\overline{x}^*_e a^T\overline{x}^e : e \in \overline{E}_{1/2}) \\ & > & b\,\big(1 - \sum(\overline{x}^*_e : e \in \overline{E}_{1/2})\big). \end{array}$$

To see that $v^T\overline{x}^0 = 0$, note that $v_e\overline{x}^0_e = 0$ for all e.

Finding a', b', and $\overline{x}'$ in Algorithm 42. To add new elements to $\mathcal{I}$ and to adjust a and b if necessary, Concorde's implementation of Algorithm 42 uses a function TILT, which, given integer vectors a, v, integers b, w, and a moderately constrained tangled tour $\overline{x}^0$ with the property

> if all moderately constrained tangled tours $\overline{x}$ have $v^T\overline{x} \leq w$,
> then $a^T\overline{x}^0 < b$ and $v^T\overline{x}^0 = w$,

returns a nonzero integer vector a', an integer b', and a moderately constrained tangled tour $\overline{x}'$ with properties (B1),

(B2$^+$) inequality ${a'}^T\overline{x} \le b'$ is a nonnegative linear combination of $a^T\overline{x} \le b$ and $v^T\overline{x} \le w$,

and (B3).

In the iterations of the **while** loop in Algorithm 42 where v and w are drawn from the catalog, Concorde calls TILT $(a, b, v, w, \overline{x}^0)$ for $(a', b', \overline{x}')$. To see that ${a'}^T\overline{x}^* > b'$, note that there are nonnegative λ and μ such that $a' = \lambda a + \mu v$, $b' = \lambda b + \mu w$; that $v^T\overline{x}^* \ge w$; and that $\lambda > 0$ since ${a'}^T\overline{x} \le b'$ is satisfied by all moderately constrained tangled tours but $v^T\overline{x} \le w$ is not.

In the iterations of the **while** loop in Algorithm 42 where $w = 0$ and v is computed by solving the system (17), (18), Concorde computes

$$(a^+, b^+, \overline{x}^+) = \text{TILT } (a, b, v, 0, \overline{x}^0),$$
$$(a^-, b^-, \overline{x}^-) = \text{TILT } (a, b, -v, 0, \overline{x}^0)$$

and then it sets

$$\begin{array}{ll} a' = a^+,\ b' = b^+,\ \overline{x}' = \overline{x}^+ & \text{if } {a^+}^T\overline{x}^* - b^+ \ge {a^-}^T\overline{x}^* - b^-, \\ a' = a^-,\ b' = b^-,\ \overline{x}' = \overline{x}^- & \text{otherwise.} \end{array}$$

To show that ${a'}^T\overline{x}^* > b'$, we are going to prove that

inequality $a^T\overline{x} \le b$ is a nonnegative linear combination of ${a^+}^T\overline{x} \le b^+$ and ${a^-}^T\overline{x} \le b^-$.

There are nonnegative numbers λ^+ and μ^+ such that

$$a^+ = \lambda^+ a + \mu^+ v, \quad b^+ = \lambda^+ b$$

and there are nonnegative numbers λ^- and μ^- such that

$$a^- = \lambda^- a - \mu^- v, \quad b^- = \lambda^- b.$$

Since ${a^+}^T\overline{x} \le b^+$ and ${a^-}^T\overline{x} \le b^-$ for all moderately constrained tangled tours $\overline{x}$ and since some moderately constrained tangled tour $\overline{x}$ has $v^T\overline{x} \ne 0$,

$$\text{at least one of } \lambda^+, \lambda^- \text{ is positive.}$$

If $\mu^+ = 0$, then $\lambda^+ > 0$, and so $a^T\overline{x} \le b$ is a positive multiple of ${a^+}^T\overline{x} \le b^+$; if $\mu^- = 0$, then $\lambda^- > 0$, and so $a^T\overline{x} \le b$ is a positive multiple of ${a^-}^T\overline{x} \le b^-$; if $\mu^+ > 0$ and $\mu^- > 0$, then $\lambda^+\mu^- + \lambda^-\mu^+ > 0$ and

$$a = \frac{\mu^+}{\lambda^+\mu^- + \lambda^-\mu^+}\, a^- + \frac{\mu^-}{\lambda^+\mu^- + \lambda^-\mu^+}\, a^+,$$
$$b = \frac{\mu^+}{\lambda^+\mu^- + \lambda^-\mu^+}\, b^- + \frac{\mu^-}{\lambda^+\mu^- + \lambda^-\mu^+}\, b^+.$$

Implementation of TILT**.** TILT could be implemented by the *Dinkelbach method* of fractional programming (see, for instance, Sect. 5.6 of Craven [15] or Sect. 4.5 of Stancu-Minasian [67]) as in Algorithm 43.

Algorithm 43. TILT $(a, b, v, w, \overline{x}^0)$*:*

$\overline{x}$ = moderately constrained tangled tour that maximizes $v^T\overline{x}$;
$\lambda = v^T\overline{x} - w$, $\mu = b - a^T\overline{x}$;
if $\lambda = 0$
then return $(v, w, \overline{x}^0)$;
else **if** $\mu = 0$
then return $(a, b, \overline{x})$;
else return TILT $(a,\ b,\ \lambda a + \mu v,\ \lambda b + \mu w,\ \overline{x})$;
end
end

For illustration, consider ITERATION 2 in our example. Here,

$$a_{03} = -2,\ a_{05} = -2,\ a_{06} = -2,\ a_{12} = 1,\ a_{25} = 1,\ a_{47} = 1,$$

and $a_{ij} = 0$ for all other i and j such that $0 \le i < j \le 7$; we have $b = 1$;

$$v_{05} = 1$$

and $v_{ij} = 0$ for all other i and j such that $0 \le i < j \le 7$; we have $w = 0$; vector x^0 may be any moderately constrained tangled tour.

In the nested recursive calls of TILT $(a, b, v, w, \overline{x})$, values of a and b do not change, but values of v and w do; in our illustration, we may specify each new v as

$$[\, v_{03},\ v_{05},\ v_{06},\ v_{12},\ v_{25},\ v_{47}\,]^T$$

since $v_{ij} = 0$ for all other i and j such that $0 \le i < j \le 7$. In this notation, a record of the computations may go as follows.

TILT $(a, b, [0,1,0,0,0,0]^T, 0, \overline{x}^0)$:
the $\overline{x}$ returned by ORACLE is 0-1-0-2-0-3-4-0-5-0-6-7-0;
$\lambda = 2$, $\mu = 9$;
TILT $(a, b, [-4,5,-4,2,2,2]^T, 2, \overline{x})$:
the $\overline{x}$ returned by ORACLE is 0-1-2-0-4-3-6-7-0-5-0;
$\lambda = 10$, $\mu = 4$;
TILT $(a, b, [-36,0,-36,18,18,18]^T, 18, \overline{x})$:
the $\overline{x}$ returned by ORACLE is 0-1-2-5-0-4-3-6-7-0;
$\lambda = 18$, $\mu = 1$;
TILT $(a, b, [-72,-36,-72,36,36,36]^T, 36, \overline{x})$:
the $\overline{x}$ returned by ORACLE is 0-1-2-5-0-4-3-6-7-0;
$\lambda = 0$, $\mu = 1$;
return $([-72,-36,-72,36,36,36]^T,\ 36,$ 0-1-2-5-0-4-3-6-7-0);
return $([-72,-36,-72,36,36,36]^T,\ 36,$ 0-1-2-5-0-4-3-6-7-0);

return $([-72,-36,-72,36,36,36]^T,\ 36,$ 0-1-2-5-0-4-3-6-7-0);
return $([-72,-36,-72,36,36,36]^T,\ 36,$ 0-1-2-5-0-4-3-6-7-0);

Concorde implements TILT as a modified version of Algorithm 43: before each recursive call of TILT $(a, b, v, w, \overline{x})$, it divides w and all the components of v by their greatest common divisor. This policy makes

TILT $(a, b, [0, 1, 0, 0, 0, 0]^T, 0, \overline{x}^0)$
  TILT $(a, b, [-4, 5, -4, 2, 2, 2]^T, 2, \overline{x})$
    TILT $(a, b, [-2, 0, -2, 1, 1, 1]^T, 1, \overline{x})$
      TILT $(a, b, [-2, -1, -2, 1, 1, 1]^T, 1, \overline{x})$

the four invocations of TILT in our example and it makes

$$([-2,-1,-2,1,1,1]^T,\ 1,\ \text{0-1-2-5-0-4-3-6-7-0})$$

their common return value.

4.4 PHASE 2: From Moderately Constrained to Weakly Constrained Tangled Tours

We will write

$$E_0 = \{e : e \subset \{1, 2, \ldots, k\}, \overline{x}^*_e = 0\},$$
$$E_1 = \{e : e \subset \{1, 2, \ldots, k\}, \overline{x}^*_e = 1\};$$

in this notation, a weakly constrained tangled tour $\overline{x}$ is moderately constrained if and only if

$$\overline{x}_e = 0 \text{ whenever } e \in E_0 \text{ and } \overline{x}_e = 1 \text{ whenever } e \in E_1.$$

The linear inequality $a^T\overline{x} \le b$ constructed in PHASE 1 separates $\overline{x}^*$ from all moderately constrained tangled tours and induces a a facet of their convex hull; in PHASE 2, we find integers Δ_e $(e \in \overline{E}_0 \cup \overline{E}_1)$ such that the inequality

$$a^T\overline{x} + \sum(\Delta_e\overline{x}_e : e \in \overline{E}_0 \cup \overline{E}_1) \le b + \sum(\Delta_e : e \in \overline{E}_1)$$

separates $\overline{x}^*$ from all weakly constrained tangled tours and induces a a facet of their convex hull. A way of computing the Δ_e one by one originated in the work of Gomory (1969) and was elaborated by Balas (1975), Hammer, Johnson, and Peled (1975), Padberg (1973,1975), Wolsey (1975a, 1975b), and others; it is known as *sequential lifting*; its application in our context is described in Algorithm 44. Both **while** loops in this algorithm maintain the invariant

$$\left.\begin{array}{l} a^T\overline{x} \le b \text{ induces a facet of the convex hull of} \\ \text{all weakly constrained tangled tours } \overline{x} \text{ such that} \\ \overline{x}_e = 0 \text{ whenever } e \in F_0 \text{ and } \overline{x}_e = 1 \text{ whenever } e \in F_1. \end{array}\right\} \quad (19)$$

Algorithm 44. *Sequential lifting*

$F_0 = \overline{E}_0$, $F_1 = \overline{E}_1$;
while $F_1 \neq \emptyset$
do $f =$ an edge in F_1;
find a weakly constrained tangled tour $\overline{x}^{\max}$ that
maximizes $a^T\overline{x}$ subject to
$\overline{x}_e = 0$ whenever $e \in F_0 \cup \{f\}$,
$\overline{x}_e = 1$ whenever $e \in F_1 - \{f\}$;
replace $a^T\overline{x} \leq b$ by $a^T\overline{x} + (a^T\overline{x}^{\max} - b)\overline{x}_f \leq a^T\overline{x}^{\max}$;
delete f from F_1;
end
while $F_0 \neq \emptyset$
do $f =$ an edge in F_0;
find a weakly constrained tangled tour $\overline{x}^{\max}$ that
maximizes $a^T\overline{x}$ subject to
$\overline{x}_e = 0$ whenever $e \in F_0 - \{f\}$,
$\overline{x}_f = 1$;
replace $a^T\overline{x} \leq b$ by $a^T\overline{x} + (b - a^T\overline{x}^{\max})\overline{x}_f \leq b$;
delete f from F_0;
end

Concorde enters PHASE 2 with an inequality $a^T\overline{x} \leq b$ such that

(i) $a^T\overline{x} \leq b$ induces a facet of the convex hull of all moderately constrained tangled tours,
(ii) $a^T\overline{x}^* > b$,
(iii) $a_e = 0$ for all e outside $\overline{E}_{1/2}$.

An arbitrary inequality $a^T\overline{x} \leq b$ with properties (i), (ii) can be made to satisfy (iii) as well by first substituting $2 - \sum(\overline{x}_e : 0 \notin e, u \in e)$ for all $\overline{x}_{0u}$ and then substituting 0 for all $\overline{x}_e$ such that $e \in E_0$ and substituting 1 for all $\overline{x}_e$ such that $e \in E_1$. In our illustrative example, these substitutions convert inequality

$$-2\overline{x}_{03} - \overline{x}_{05} - 2\overline{x}_{06} + \overline{x}_{12} + \overline{x}_{25} + \overline{x}_{47} \leq 1$$

with properties (i), (ii) to inequality

$$\overline{x}_{12} + \overline{x}_{15} + 2\overline{x}_{23} + 2\overline{x}_{25} + 3\overline{x}_{35} + 4\overline{x}_{36} + \overline{x}_{47} + 3\overline{x}_{56} \leq 7$$

with properties (i), (ii), and (iii).

Concorde implements PHASE 2 as a streamlined version of Algorithm 44; to describe this version, we will write

$$\overline{V}^{\text{int}} = \{1, 2, \ldots, k\} \quad \text{and} \quad \overline{E}^{\text{int}} = \{e : e \subset \overline{V}^{\text{int}}, |e| = 2\}.$$

Since every weakly constrained tangled tour $\overline{x}$ satisfies

$$\overline{x}_{0u} = 2 - \sum(\overline{x}_e : e \in \overline{E}^{\text{int}}, u \in e) \text{ for all } u \text{ in } \overline{V}^{\text{int}},$$

it is determined by its restriction on $\overline{E}^{\text{int}}$. Restrictions of weakly constrained tangled tours on $\overline{E}^{\text{int}}$ are precisely the incidence vectors of the edge-sets of *path systems* – meaning graphs whose connected components are paths – with vertex-set $\overline{V}^{\text{int}}$. The set of all path systems with vertex-set $\overline{V}^{\text{int}}$ is *monotone* in the sense that the removal of an edge from a path system yields another path system.

Monotonicity of the set of all path systems with vertex-set $\overline{V}^{\text{int}}$ implies that, for all choices of subsets R_0, R_1 of $\overline{E}^{\text{int}}$ and for all choices of objective functions $a^T x$ such that

$$a_e = 0 \text{ for all } e \text{ outside } \overline{E}^{\text{int}}$$

(this property of a is maintained by Algorithm 44), the problem of finding a weakly constrained tangled tour that

$$\begin{array}{l}\text{maximizes } a^T\overline{x} \text{ subject to}\\ \overline{x}_e = 0 \text{ whenever } e \in R_0,\\ \overline{x}_e = 1 \text{ whenever } e \in R_1\end{array}$$

either has no feasible solution at all or else has an optimal solution such that

$$\overline{x}_e = 0 \text{ whenever } e \in \overline{E}^{\text{int}} - (R_0 \cup R_1) \text{ and } a_e = 0.$$

Concorde exploits this observation in a couple of ways: it makes the job of ORACLE easier by adding constraint

$$\overline{x}_e = 0 \text{ whenever } e \in \overline{E}^{\text{int}} - (F_0 \cup F_1) \text{ and } a_e = 0$$

to the problem of finding $\overline{x}^{\max}$ and it skips certain calls of ORACLE altogether.

The second of these two tricks begins with the set $\mathcal{I}$ of $|\overline{E}_{1/2}|$ moderately constrained tangled tours produced in PHASE 1: for each element $\overline{x}'$ of $\mathcal{I}$, Concorde deletes from F_0 all the edges uv such that u and v are endpoints of distinct paths in the path system defined by $\overline{x}'$. To see that each of these deletions preserves invariant (19), consider the weakly constrained tangled tour $\overline{x}''$ that

$$\begin{array}{l}\text{maximizes } a^T\overline{x} \text{ subject to}\\ \overline{x}_e = 0 \text{ whenever } e \in F_0 - \{uv\},\\ \overline{x}_e = 1 \text{ whenever } e \in F_1 \cup \{uv\}.\end{array}$$

On the one hand, the path system defined by $\overline{x}'$ with edge uv added shows that $a^T\overline{x}'' \geq b$; on the other hand, the path system defined by $\overline{x}''$ with edge uv deleted shows that $a^T\overline{x}'' \leq b$; we conclude that $a^T\overline{x}'' = b$, and so the deletion of uv from F_0 preserves invariant (19).

This trick applies not only to the elements of $\mathcal{I}$, but also to each weakly tangled tour found by ORACLE in Phase 2: as soon as it finds a new $x^{\max}$, Concorde deletes from F_0 all the edges uv such that u and v are endpoints of distinct paths in the path system defined by $x^{\max}$.

In our illustrative example, we begin with

$$F_0 = \{13, 14, 16, 17, 24, 26, 27, 37, 45, 46, 57\}, \quad F_1 = \{34, 67\}$$

and then we examine the eight path systems defined by $\mathcal{I}$:

- the path system 4-3-2-1-5-6-7 is a single path,
- the path system with components 1-2 and 4-3-5-6-7 eliminates edges 14, 17, 24, 27 from F_0,
- the path system with components 1-5-2 and 4-3-6-7 eliminates edges 14, 17, 24, 27 from F_0,
- the path system with components 1 and 4-3-2-5-6-7 eliminates edges 14, 17 from F_0,
- the path system 1-5-6-7-4-3-2 is a single path,
- the path system 1-2-5-6-7-4-3 is a single path,
- the path system with components 1-2-5 and 4-3-6-7 eliminates edges 14, 17, 45, 57 from F_0,
- the path system 1-2-5-3-4-7-6 is a single path.

The initial $a^T\overline{x} \leq b$ reads

$$\overline{x}_{12} + \overline{x}_{15} + 2\overline{x}_{23} + 2\overline{x}_{25} + 3\overline{x}_{35} + 4\overline{x}_{36} + \overline{x}_{47} + 3\overline{x}_{56} \leq 7. \tag{20}$$

The first **while** loop of Algorithm 44 may go as follows.

ITERATION 1: $F_0 = \{13, 16, 26, 37, 46\},\ F_1 = \{34, 67\},\ f = 34.$

ORACLE, called to find a weakly constrained tangled tour that maximizes the left-hand side $a^T\overline{x}$ of (20) subject to

$\overline{x}_{13} = \overline{x}_{14} = \overline{x}_{16} = \overline{x}_{17} = \overline{x}_{24} = \overline{x}_{26} = \overline{x}_{27} = \overline{x}_{37} = \overline{x}_{45} = \overline{x}_{46} = \overline{x}_{57} = 0$, $\overline{x}_{34} = 0$, and $\overline{x}_{67} = 1$,

returns the $\overline{x}^{\max}$ represented by

- the path system with the single component 1-2-5-3-6-7-4;

since $a^T\overline{x}^{\max} = 11$, we replace (20) by

$$\overline{x}_{12} + \overline{x}_{15} + 2\overline{x}_{23} + 2\overline{x}_{25} + 4\overline{x}_{34} + 3\overline{x}_{35} + 4\overline{x}_{36} + \overline{x}_{47} + 3\overline{x}_{56} \leq 11. \tag{21}$$

ITERATION 2: $F_0 = \{13, 16, 26, 37, 46\},\ F_1 = \{67\},\ f = 67.$

ORACLE, called to find a weakly constrained tangled tour that maximizes the left-hand side $a^T\overline{x}$ of (21) subject to

$\overline{x}_{13} = \overline{x}_{14} = \overline{x}_{16} = \overline{x}_{17} = \overline{x}_{24} = \overline{x}_{26} = \overline{x}_{27} = \overline{x}_{37} = \overline{x}_{45} = \overline{x}_{46} = \overline{x}_{57} = 0$, $\overline{x}_{67} = 0$,

returns the $\overline{x}^{\max}$ represented by

- the path system with the single component 1-2-5-6-3-4-7;

since $a^T\overline{x}^{\max} = 15$, we replace (21) by

$$\overline{x}_{12} + \overline{x}_{15} + 2\overline{x}_{23} + 2\overline{x}_{25} + 4\overline{x}_{34} + 3\overline{x}_{35} + 4\overline{x}_{36} + \overline{x}_{47} + 3\overline{x}_{56} + 4\overline{x}_{67} \leq 15. \tag{22}$$

The second **while** loop of Algorithm 44 may go as follows.

ITERATION 1: $F_0 = \{13, 16, 26, 37, 46\},\ F_1 = \emptyset,\ f = 13.$

ORACLE, called to find a weakly constrained tangled tour that maximizes the left-hand side $a^T\overline{x}$ of (22) subject to

$$\overline{x}_{14} = \overline{x}_{16} = \overline{x}_{17} = \overline{x}_{24} = \overline{x}_{26} = \overline{x}_{27} = \overline{x}_{37} = \overline{x}_{45} = \overline{x}_{46} = \overline{x}_{57} = 0,$$
$$\overline{x}_{13} = 1,$$

returns the $\overline{x}^{\max}$ represented by

- the path system with the single component 1-3-4-7-6-5-2;

since $a^T\overline{x}^{\max} = 14$, we replace (22) by

$$\overline{x}_{12} + \overline{x}_{13} + \overline{x}_{15} + 2\overline{x}_{23} + 2\overline{x}_{25} + 4\overline{x}_{34} + 3\overline{x}_{35} + 4\overline{x}_{36} + \overline{x}_{47} + 3\overline{x}_{56} + 4\overline{x}_{67} \leq 15. \tag{23}$$

ITERATION 2: $F_0 = \{16, 26, 37, 46\},\ F_1 = \emptyset,\ f = 16.$

ORACLE, called to find a weakly constrained tangled tour that maximizes the left-hand side $a^T\overline{x}$ of (23) subject to

$$\overline{x}_{14} = \overline{x}_{17} = \overline{x}_{24} = \overline{x}_{26} = \overline{x}_{27} = \overline{x}_{37} = \overline{x}_{45} = \overline{x}_{46} = \overline{x}_{57} = 0,$$
$$\overline{x}_{16} = 1,$$

returns the $\overline{x}^{\max}$ represented by

- the path system with the single component 1-6-7-4-3-5-2;

since $a^T\overline{x}^{\max} = 14$, we replace (23) by

$$\overline{x}_{12} + \overline{x}_{13} + \overline{x}_{15} + \overline{x}_{16} + 2\overline{x}_{23} + 2\overline{x}_{25} + 4\overline{x}_{34} + 3\overline{x}_{35} + 4\overline{x}_{36} + \overline{x}_{47} + 3\overline{x}_{56} + 4\overline{x}_{67} \leq 15. \tag{24}$$

ITERATION 3: $F_0 = \{26, 37, 46\},\ F_1 = \emptyset,\ f = 26.$

ORACLE, called to find a weakly constrained tangled tour that maximizes the left-hand side $a^T\overline{x}$ of (24) subject to

$$\overline{x}_{14} = \overline{x}_{17} = \overline{x}_{24} = \overline{x}_{27} = \overline{x}_{37} = \overline{x}_{45} = \overline{x}_{46} = \overline{x}_{57} = 0,$$
$$\overline{x}_{26} = 1,$$

returns the $\overline{x}^{\max}$ represented by

- the path system with the single component 1-5-3-4-7-6-2;

since $a^T\overline{x}^{\max} = 13$, we replace (24) by

$$\overline{x}_{12} + \overline{x}_{13} + \overline{x}_{15} + \overline{x}_{16} + 2\overline{x}_{23} + 2\overline{x}_{25} + 2\overline{x}_{26} + 4\overline{x}_{34} + 3\overline{x}_{35} + 4\overline{x}_{36} + \overline{x}_{47} + 3\overline{x}_{56} + 4\overline{x}_{67} \leq 15. \tag{25}$$

ITERATION 4: $F_0 = \{37, 46\},\ F_1 = \emptyset,\ f = 37.$

ORACLE, called to find a weakly constrained tangled tour that maximizes the left-hand side $a^T\overline{x}$ of (25) subject to

$$\overline{x}_{14} = \overline{x}_{17} = \overline{x}_{24} = \overline{x}_{27} = \overline{x}_{45} = \overline{x}_{46} = \overline{x}_{57} = 0,$$
$$\overline{x}_{37} = 1,$$

returns the $\overline{x}^{\max}$ represented by

- the path system with the single component 1-2-5-6-7-3-4;

since $a^T\overline{x}^{\max} = 14$, we replace (25) by

$$\overline{x}_{12} + \overline{x}_{13} + \overline{x}_{15} + \overline{x}_{16} + 2\overline{x}_{23} + 2\overline{x}_{25} + 2\overline{x}_{26} + 4\overline{x}_{34} + 3\overline{x}_{35} + 4\overline{x}_{36} + \overline{x}_{37} + \overline{x}_{47} + 3\overline{x}_{56} + 4\overline{x}_{67} \leq 15. \tag{26}$$

ITERATION 5: $F_0 = \{46\},\ F_1 = \emptyset,\ f = 46$.

ORACLE, called to find a weakly constrained tangled tour that maximizes the left-hand side $a^T\overline{x}$ of (26) subject to

$$\overline{x}_{14} = \overline{x}_{17} = \overline{x}_{24} = \overline{x}_{27} = \overline{x}_{45} = \overline{x}_{57} = 0,$$
$$\overline{x}_{46} = 1,$$

returns the $\overline{x}^{\max}$ represented by

- the path system with the single component 1-2-5-3-4-6-7;

since $a^T\overline{x}^{\max} = 14$, we replace (26) by

$$\overline{x}_{12} + \overline{x}_{13} + \overline{x}_{15} + \overline{x}_{16} + 2\overline{x}_{23} + 2\overline{x}_{25} + 2\overline{x}_{26} + 4\overline{x}_{34} + 3\overline{x}_{35} + 4\overline{x}_{36} + \overline{x}_{37} + \overline{x}_{46} + +\overline{x}_{47} + 3\overline{x}_{56} + 4\overline{x}_{67} \leq 15. \tag{27}$$

After this iteration, $F_0 = F_1 = \emptyset$, and so (27) induces a facet of the convex hull of all weakly constrained tangled tours.

4.5 PHASE 3: **From Weakly Constrained to All Tangled Tours**

PHASE 2 produces an inequality $a^T\overline{x} \leq b$ such that

(i) $a^T\overline{x} \leq b$ induces a facet of the convex hull of all weakly constrained tangled tours,
(ii) $a^T\overline{x}^* > b$,
(iii) $a_e = 0$ for all e outside $\overline{E}^{\text{int}}$;

since the set of restrictions of weakly constrained tangled tours onto $\overline{E}^{\text{int}}$ is monotone, (i), (ii), and (iii) imply that

(iv) $a_e > 0$ for all e in $\overline{E}^{\text{int}}$.

A tangled tour $\overline{x}$ is weakly constrained if and only if

$$\sum(\overline{x}_e : e \in \overline{E}^{\text{int}}, w \in e) = 2 \text{ for all } w \text{ in } \overline{V}^{\text{int}};$$

if a hypergraph inequality

$$\sum(\lambda_Q x(Q, \overline{V} - Q) : Q \in \overline{\mathcal{H}}) \geq \beta \tag{28}$$

is related to $a^T\overline{x} \leq b$ in the sense that, for some numbers $\pi_w (w \in \overline{V}^{\text{int}})$, the left-hand side of (28) is identically equal to

$$\sum(\pi_w \sum(\overline{x}_e : w \in e) : w \in \overline{V}^{\text{int}}) - 2a^T\overline{x}$$

and the right-hand side of (28) equals

$$\sum(2\pi_w : w \in \overline{V}^{\text{int}}) - 2b,$$

then (28) induces a facet of the convex hull of all weakly constrained tangled tours and is violated by $\overline{x}^*$. In PHASE 3, we find a hypergraph inequality that is related to $a^T\overline{x} \leq b$ in this sense and is satisfied by all tangled tours. Algorithm 45 accomplishes this objective with a negligible amount of computations.

Algorithm 45. PHASE 3:

construct a hypergraph $\overline{\mathcal{H}}$ on $\overline{V}^{\text{int}}$ and positive integers $\lambda_Q (Q \in \overline{\mathcal{H}})$
such that the linear form
$\sum \lambda_Q (\sum(\overline{x}_e : e \subseteq Q) : Q \in \overline{\mathcal{H}})$
is identically equal to $a^T\overline{x}$;
return the inequality
$\sum(\lambda_Q x(Q, \overline{V} - Q) : Q \in \overline{\mathcal{H}}) \geq \sum(2\lambda_Q |Q| : Q \in \overline{\mathcal{H}}) - 2b$;

Arguments used by Naddef and Rinaldi [53] show that the hypergraph inequality returned by Algorithm 45 is satisfied by all tangled tours; it is a routine matter to verify that the numbers defined by $\pi_w = \sum(\lambda_Q : w \in Q, Q \in \overline{\mathcal{H}})$ have the desired properties.

A straightforward way of constructing the $\overline{\mathcal{H}}$ and the $\lambda_Q (Q \in \overline{\mathcal{H}})$ in Algorithm 45 is to let $\overline{\mathcal{H}}$ consist of all two-point subsets e such that $a_e > 0$ and to set $\lambda_e = a_e$ for all e in $\overline{\mathcal{H}}$. Instead, Concorde constructs $\overline{\mathcal{H}}$ and $\lambda_Q (Q \in \overline{\mathcal{H}})$ by an iterative greedy procedure: in each iteration, it chooses a maximal (with respect to set-inclusion) subset Q of $\overline{V}^{\text{int}}$ such that $a_e > 0$ for all edges e with both endpoints in Q, it lets λ_Q be the smallest of these positive a_e, it brings Q into $\overline{\mathcal{H}}$, and it subtracts λ_Q from all a_e such that e has both endpoints in Q.

In our illustrative example, the inequality $a^T\overline{x} \leq b$ produced in PHASE 2 reads

$$\begin{aligned}\overline{x}_{12} + \overline{x}_{13} + \overline{x}_{15} + \overline{x}_{16} + 2\overline{x}_{23} + 2\overline{x}_{25} + 2\overline{x}_{26} + 4\overline{x}_{34} \\ +3\overline{x}_{35} + 4\overline{x}_{36} + \overline{x}_{37} + \overline{x}_{46} + +\overline{x}_{47} + 3\overline{x}_{56} + 4\overline{x}_{67} \leq 15;\end{aligned}$$

we construct the $\overline{\mathcal{H}}$ and the $\lambda_Q (Q \in \overline{\mathcal{H}})$ in Algorithm 45 as follows.

1. We choose $Q = \{1, 2, 3, 5, 6\}$, which yields $\lambda_Q = 1$ and leaves us with

$$a^T x = \overline{x}_{23} + \overline{x}_{25} + \overline{x}_{26} + 4\overline{x}_{34} + 2\overline{x}_{35} + 3\overline{x}_{36} + \overline{x}_{37} + \overline{x}_{46} + \overline{x}_{47} + 2\overline{x}_{56} + 4\overline{x}_{67}.$$

2. We choose $Q = \{2, 3, 5, 6\}$, which yields $\lambda_Q = 1$ and leaves us with
$$a^T x = 4\overline{x}_{34} + \overline{x}_{35} + 2\overline{x}_{36} + \overline{x}_{37} + \overline{x}_{46} + \overline{x}_{47} + \overline{x}_{56} + 4\overline{x}_{67}.$$
3. We choose $Q = \{3, 4, 6, 7\}$, which yields $\lambda_Q = 1$ and leaves us with
$$a^T x = 3\overline{x}_{34} + \overline{x}_{35} + \overline{x}_{36} + \overline{x}_{56} + 3\overline{x}_{67}.$$
4. We choose $Q = \{3, 4\}$, which yields $\lambda_Q = 3$ and leaves us with
$$a^T x = \overline{x}_{35} + \overline{x}_{36} + \overline{x}_{56} + 3\overline{x}_{67}.$$
5. We choose $Q = \{3, 5, 6\}$, which yields $\lambda_Q = 1$ and leaves us with
$$a^T x = 3\overline{x}_{67}.$$
6. We choose $Q = \{6, 7\}$, which yields $\lambda_Q = 3$ and leaves us with
$$a^T x = 0.$$

The resulting hypergraph inequality,

$$\begin{aligned}
&\overline{x}(\{1,2,3,5,6\},\{0,4,7\})\\
&+\overline{x}(\{2,3,5,6\},\{0,1,4,7\})\\
&+\overline{x}(\{3,4,6,7\},\{0,1,2,5\})\\
&+3\overline{x}(\{3,4\},\{0,1,2,5,6,7\})\\
&+\overline{x}(\{3,5,6\},\{0,1,2,4,7\})\\
&+3\overline{x}(\{6,7\},\{0,1,2,3,4,5\}) \geq 26\,,
\end{aligned}$$

induces a facet of the convex hull of all weakly constrained tangled tours and is violated by $\overline{x}^*$. (By the way, this inequality belongs to the class of *path inequalities* of Cornuéjols, Fonlupt, and Naddef (1985).)

Every cut produced by Algorithm 45 can be easily transformed into a cut that induces a facet of the graphical traveling salesman polytope. It can be shown that, in case the $a^T\overline{x} \leq b$ produced in Phase 2 is such that

$$\text{every } a_e \text{ with } e \in \overline{E}^{\text{int}} \text{ is a positive integer,} \tag{29}$$

this $a^T\overline{x} \leq b$ reads

$$\lambda \sum(\overline{x}_e : e \in \overline{E}^{\text{int}}) \leq \lambda(k-1)$$

for some positive λ, and so Algorithm 45 returns a positive multiple of the subtour inequality

$$\overline{x}(\{0\}, \overline{V}^{\text{int}}) \geq 2;$$

as noted in Sect. 4.3, Cornuéjols, Fonlupt, and Naddef [14] pointed out that subtour inequalities induce facets of the convex hull of $\mathcal{T}$. It can also be shown that, in case (29) fails, the following procedure transforms the cut

$$\sum(\lambda_Q x(Q, \overline{V} - Q) : Q \in \overline{\mathcal{H}}) \geq \beta$$

produced by Algorithm 45 into a cut that induces a facet of the graphical traveling salesman polytope:

STEP 1. For all choices of distinct points u, v, w of $\overline{V}$, define

$$\tau(u,v,w) = \sum(\lambda_Q : Q \in \overline{\mathcal{H}}, u \in Q, v \in Q, w \notin Q) + \sum(\lambda_Q : Q \in \overline{\mathcal{H}}, u \notin Q, v \notin Q, w \in Q).$$

STEP 2. For all points w of $\overline{V}^{\text{int}}$, evaluate

$$\Delta_w = \min\{\tau(u,v,w) : uv \in \overline{E}, u \neq w, v \neq w\}.$$

STEP 3. Return the inequality

$$\sum(\lambda_Q x(Q, \overline{V} - Q) : Q \in \overline{\mathcal{H}}) - \sum(\Delta_w \overline{x}(\{w\}, \overline{V} - \{w\}) : w \in \overline{V}^{\text{int}}) \geq \beta - 2\sum(\Delta_w : w \in \overline{V}^{\text{int}}).$$

Again, the arguments come from Naddef and Rinaldi [53]; in their terminology, the inequality returned in STEP 3 is *tight triangular*.

The current version of Concorde can handle only hypergraph constraints with nonnegative coefficients; it settles for the cut produced by Algorithm 45 even when this cut does not induce a facet of the graphical traveling salesman polytope. Concorde's way of choosing $\overline{\mathcal{H}}$ and $\lambda_Q(Q \in \overline{\mathcal{H}})$ aims to mitigate the effects of this carelessness by reducing the number of points w such that $\Delta_w > 0$.

In our illustrative example, $\overline{\mathcal{H}}$ consists of

$$\{1,2,3,5,6\},\ \{2,3,5,6\},\ \{3,4,6,7\},\ \{3,4\},\ \{3,5,6\},\ \{6,7\};$$

since no member of $\overline{\mathcal{H}}$ contains both 1 and 4, assumption (29) is satisfied; since

$$\begin{aligned}
\Delta_1 &= \tau(0,3,1) = 0,\\
\Delta_2 &= \tau(0,3,2) = 0,\\
\Delta_3 &= \tau(4,5,3) = 0,\\
\Delta_4 &= \tau(0,3,4) = 0,\\
\Delta_5 &= \tau(0,3,5) = 0,\\
\Delta_6 &= \tau(3,7,6) = 0,\\
\Delta_7 &= \tau(0,6,7) = 0,
\end{aligned}$$

we have $\Delta_w = 0$ for all $w = 1, 2, \ldots, 7$. It follows that this path inequality is tight triangular and induces a facet of the graphical traveling salesman polytope.

5 Making Choices of $V_0, V_1, \ldots, V_k$

Concorde's choices of $V_0, V_1, \ldots, V_k$ in Algorithm 31 are guided by x^* in a way similar to that used by Christof and Reinelt [12] in their algorithm for finding cuts that match templates from a prescribed large catalog. First, it constructs

once and for all an equivalence relation on V in such a way that each equivalence class $V^\star$ of this relation satisfies

$$x^*(V^\star, V - V^\star) = 2;$$

then it makes many different choices of $V_0, V_1, \ldots, V_k$ in such a way that each of $V_1, \ldots, V_k$ is one of these equivalence classes and $V_0 = V - (V_1 \cup \ldots \cup V_k)$.

With W standing for the set of the equivalence classes on V, the first stage amounts to preshrinking V onto W; making each of the many different choices of $V_0, V_1, \ldots, V_k$ in the second stage means choosing a small subset of W and shrinking the entire remainder of W onto a single point. In terms of the preshrunk set W, each choice of $V_0, V_1, \ldots, V_k$ in the second stage zooms in onto a relatively small part of the problem – typically k is at most thirty or so and $|W|$ may run to hundreds or thousands – and effectively discards the rest. For this reason, we developed the habit of referring to the cuts produced by Algorithm 31 as *local cuts* and referred to them by this name in Applegate et al. [2]. In terms of the original V, each of the sets $V_1, \ldots, V_k$ could be quite large, which makes the qualifier "local" a misnomer. Still, a crisp label for the cuts produced by Algorithm 31 is convenient to have and "local cuts" seems to be as good a name as any other that we managed to think up.

The equivalence relation is constructed by iteratively shrinking two-point sets into a single point. At each stage of this process, we have a set W and a mapping $\pi : W \to 2^V$ that defines a partition of V into pairwise disjoint subsets $\pi(w)$ with $w \in W$. Initially, $W = V$ and each $\pi(w)$ is the singleton $\{w\}$; as long as there are distinct elements u, v, w of W such that

$$x^*(\pi(u), \pi(v)) = 1 \quad \text{and} \quad x^*(\pi(u), \pi(w)) + x^*(\pi(v), \pi(w)) = 1, \tag{30}$$

we keep replacing $\pi(u)$ by $\pi(u) \cup \pi(v)$ and removing v from W; when there are no u, v, w with property (30), we stop. (During this process, we may discover pairs u, v with $x^*(\pi(u), \pi(v)) > 1$, in which case x^* violates the subtour inequality $x(Q, V - Q) \geq 2$ with $Q = \pi(u) \cup \pi(v)$.)

To make the many different choices of $V_1, \ldots, V_k$, we first set the value of a parameter t that nearly determines the value of k in the sense that $t - 3 \leq k \leq t$. Then, for each w in W, we choose a subset C of W so that $w \in C$ and $t - 3 \leq |C| \leq t$; the corresponding $V_1, \ldots, V_k$ are the $\pi(v)$ with $v \in C$. The choice of C is guided by the graph with vertex-set W, where u and v are adjacent if and only if $x^*(\pi(u), \pi(v)) > \varepsilon$ for some prescribed zero tolerance ε: starting at w, we carry out a breadth-first search through this graph, until we collect a set C of $t - 3$ vertices. If there are any vertices u outside this C such that $x^*(\pi(u), \pi(v)) = 1$ for some v in C, then we keep adding these vertices u to C as long as $|C| \leq t$.

It seems plausible that such a crude way of choosing C can be improved. However, we found its performance satisfactory; none of the alternatives that we tried appeared to work better.

6 Experimental Findings

Reinelt [64] created a library named TSPLIB that contains sample instances of the TSP (and related problems) from various sources and of various types. There are 110 TSP instances in this library. Some of them arise from the task of drilling holes in printed circuit boards and others have been constructed artificially, often in the Dantzig-Fulkerson-Johnson tradition of choosing a set of actual cities and defining the cost of travel from X to Y as the distance between X and Y. None of them (with a single exception, the problem named `ts225`) is contrived to be hard and none of them is contrived to be easy; 106 of them have been solved and four have not.

The results reported in this section involve various TSP instances drawn from TSPLIB. The *default code* is Concorde 99.12.15 of Applegate et al. [3] with 99 as the random number seed (`concorde -s 99 xxx.tsp`). The *running times* are given in seconds on a Compaq XP1000 workstation with a 500 MHz EV6 Alpha processor.

6.1 The Easier TSPLIB Instances

Our default code solved 87 of the 110 TSPLIB instances in under 1000 seconds. These instances are listed in Table 1.

In Sect. 5, we described a way of producing a number of partititions of V into nonempty sets $V_0, V_1, \ldots, V_k$ with $t - 3 \leq k \leq t$ for a prescribed positive integer t. Concorde uses it with t ranging between 8 and a prescribed integer $t_{\max}$. More precisely, the search always begins with $t = 8$. Whenever a value of t is set, Concorde adds all the resulting cuts produced by Algorithm 31 to the LP relaxation of our problem and it solves the tightened LP relaxation; if the increase in the value of the relaxation is not satisfactory and $t < t_{\max}$, then the next iteration takes place with t incremented by one. The default setting, $t_{\max} = 16$, was prompted by results reported in Table 2. The trends exhibited in this table are hardly surprising: increases of $t_{\max}$ yield tighter LP relaxations, but they also require additional time to construct these tighter relaxations.

Table 2 shows also that the total time to solve the 87 TSPLIB instances in Table 1 by our default code would increase to 126.8% of its original value if local cuts were turned off. This endorsement of local cuts pales in comparison with other cutting-plane routines: if we turned off a class of comb-finding procedures proposed by Padberg and Rinaldi [62] (based on the block decomposition of the graph obtained by considering only edges e with $0 < x^*_e < 1$), then the running time over our default code would increase to 427.2% of its original value.

6.2 Three of the Harder TSPLIB Instances

Two of the harder TSPLIB instances solved by Concorde are the printed-circuit board instance `pcb3038` and the geographical instance `fnl4461`. On each of them, we have run tests similar to those of Table 2; their results are reported

Table 1. 87 instances from the TSPLIB
NT denotes the number of nodes in the branch-and-cut tree

name	NT	time	name	NT	time	name	NT	time
burma14	1	0.06	lin105	1	0.59	lin318	1	9.74
ulysses16	1	0.22	pr107	1	1.03	rd400	15	148.42
gr17	1	0.08	gr120	1	2.23	fl417	5	57.75
gr21	1	0.03	pr124	1	3.64	gr431	13	133.29
ulysses22	1	0.53	bier127	1	1.65	pr439	15	216.75
gr24	1	0.07	ch130	1	2.13	pcb442	9	49.92
fri26	1	0.07	pr136	1	3.97	d493	5	113.32
bayg29	1	0.09	gr137	1	3.42	att532	7	109.52
bays29	1	0.13	pr144	1	2.58	ali535	3	53.14
dantzig42	1	0.23	ch150	1	3.03	si535	3	43.13
swiss42	1	0.13	kroA150	1	5.00	pa561	17	246.82
att48	1	0.56	kroB150	1	4.23	u574	1	23.04
gr48	1	0.67	pr152	1	7.93	rat575	25	363.07
hk48	1	0.17	u159	1	1.00	p654	3	26.52
eil51	1	0.73	si175	3	13.09	d657	13	260.37
berlin52	1	0.29	brg180	1	1.46	gr666	3	49.86
brazil58	1	0.68	rat195	5	22.23	u724	11	225.44
st70	1	0.50	d198	3	11.82	rat783	1	37.88
eil76	1	0.30	kroA200	1	6.59	dsj1000	7	410.32
pr76	1	1.86	kroB200	1	3.91	pr1002	1	34.30
gr96	1	6.71	gr202	1	5.01	si1032	1	25.47
rat99	1	0.95	ts225	1	20.52	u1060	21	571.43
kroA100	1	1.00	tsp225	1	15.01	vm1084	11	604.78
kroB100	1	2.36	pr226	1	4.35	pcb1173	19	468.27
kroC100	1	0.96	gr229	3	38.61	rl1304	1	189.20
kroD100	1	1.00	gil262	1	13.06	nrw1379	19	578.42
kroE100	1	2.44	pr264	1	2.67	u1432	3	223.70
rd100	1	0.67	a280	3	5.37	d1655	5	263.03
eil101	1	0.74	pr299	3	17.49	pr2392	1	116.86

in Table 3. (For the sake of uniformity, we have started each of the nine runs on each of the two instances with the value of the optimal tour as the upper bound.)

As with Table 2, the trends exhibited in Table 3 are hardly surprising, although the dependence of running time on $t_{\max}$ is not quite as neat for `fnl4461` (note its increase as $t_{\max}$ moves from 0 to 8) and it is even more erratic for `pcb3038`. One striking difference between the easier instances on the one hand and `pcb3038`, `fnl4461` on the other hand is an increase in the optimal setting of $t_{\max}$, which is 28 for `pcb3038` and 32 for `fnl4461`. These experimental results agree with the intuition that the harder instances are better off with the larger values of $t_{\max}$. Another striking difference is the increased effect of local cuts on the overall running time. With `pcb3038`, local cuts with the default $t_{\max} = 16$ reduce the running time to 50.3% of its original value and setting $t_{\max} = 28$ reduces it further to 43.8%. For `fnl4461`, the figures are more impressive: local

Table 2. The effect of $t_{\max}$ on the easier TSPLIB instances

$t_{\max}$	total time to solve the 87 instances in Table 1	number of instances solved without branching
0	7424.58	42
8	6679.15	48
10	6624.02	52
12	6248.12	54
14	6133.06	59
16	5900.72	59
18	6394.10	64
20	8818.98	65
22	9591.02	65
24	16519.62	68
26	23285.19	67
28	35798.49	64
30	40732.42	66

Table 3. The effect of $t_{\max}$ on two harder TSPLIB instances

root LP = optimal value of the LP relaxation before the first branching
NT denotes the number of nodes in the branch-and-cut tree

`pcb3038` optimal value = 137694				`fnl4461` optimal value = 182566			
$t_{\max}$	root LP	NT	time	$t_{\max}$	root LP	NT	time
0	137592.61	665	145241.25	0	182471.39	14977	1990568.46
8	137595.10	659	162228.02	8	182487.75	11903	2286217.24
12	137613.44	383	105026.74	12	182506.27	2011	297019.61
16	137625.28	271	73085.72	16	182519.87	1071	194441.98
20	137637.25	271	119042.35	20	182530.12	417	78398.31
24	137640.66	155	65757.78	24	182541.98	213	53420.13
28	137644.28	107	63678.80	28	182543.60	137	49719.02
32	137643.42	127	129460.96	32	182546.60	81	44957.02
36	137651.53	101	233092.39	36	182549.72	59	56714.06

cuts with the default $t_{\max} = 16$ reduce the running time to 9.8% of its original value and setting $t_{\max} = 32$ reduces it further to 2.3%.

The hardest TSPLIB instances that we have solved are `usa13509` and `d15112`. Batoukov and Sørevik [6] refer to our solution of `usa13509` as one of "only a few heroic examples on successful computation on a network of workstations" and we do not propose to contradict them. This instance was solved by running an earlier version of Concorde in parallel on a network of 48 workstations, including Digital Alphas, Intel Pentium IIs and Pentium Pros, and Sun UltraSparcs; a very rough estimate of the total running time is about 4 years on our single Compaq XP1000.

Four years of CPU time is an exorbitant figure. We believe that without the use of local cuts, this exorbitant figure would grow further to a level which

would put solving `usa13509` out of our reach even if we had gathered many more workstations for the heroic purpose.

The experimental results presented in Table 4 support this sentiment. Its 29 rows correspond to 29 runs that tighten up the root LP relaxation. Each of these runs starts where the preceding run has stopped and it takes note of the gap g between the length of the optimal tour and the optimal value of the current LP relaxation. Then it attempts to tighten the relaxation by additional cuts and it solves this tightened relaxation; this step gets iterated as long as the optimal value of the relaxation keeps increasing by at least a prescribed fixed percentage of g.

We begin with five runs of Def C00, meaning our default code with local cuts turned off. The last three of these runs yields only imperceptible improvements and they narrow the relative gap to 0.0561%. Then we follow with four runs of All C00, meaning Def C00 with additional cutting-plane routines (other than local cuts) that we found to be a hindrance in solving the easy instances from Table 1 but that might help in solving the harder instances. (These techniques include our implementation of what we understood from Naddef and Thienel [54, 55], the dominos-and-necklaces heuristic for finding violated comb inequalities that has been described in Sect. 3 of Applegate et al. [1], and other cutting-plane routines.) The last three of these runs yields only imperceptible improvements and they narrow the relative gap to 0.0435%. This is the best we can do without the use of local cuts.

Then we bring in local cuts in nineteen runs of the default code with $t_{\max}$ progressing through the sequence 8, 10, 12, ..., 44. As a result, the relative gap is narrowed to 0.0164%, less than two fifths of the previous value. Finally, an additional run of All C00 yields only an imperceptible improvement.

7 Generalizations

The cutting-plane method and its descendants are applicable to any problem

$$\text{minimize } c^T x \text{ subject to } x \in \mathcal{S},$$

where $\mathcal{S}$ is a finite subset of some Euclidean space $\mathbb{R}^m$, provided that an efficient algorithm to recognize points of $\mathcal{S}$ is available. The corresponding general problem of finding cuts is this:

given (by means of an efficient membership-testing oracle)
a finite subset $\mathcal{S}$ of some $\mathbb{R}^m$ and
given a point x^* in $\mathbb{R}^m$ that lies outside the convex hull of $\mathcal{S}$,
find a vector a and a scalar b such that
$$\mathcal{S} \subset \{x : a^T x \le b\} \text{ and } a^T x^* > b.$$

Algorithm 31, dealing with the special case case where $m = n(n-1)/2$ and $\mathcal{S}$ is the set of the incidence vectors of all the tours through a set of n cities, generalizes to Algorithm 71. (In the special case, $d = (k+1)k/2$, $\phi(x) = \overline{x}$, and $\mathcal{T}$ is the set of all tangled tours through $\overline{V}$.)

Table 4. Local cuts and `usa13509`

`usa13509` optimal value = 19982859			
Code	root LP	gap	time
Def C00	19967864.84	0.0751%	2157.08
+Def C00	19970751.68	0.0606%	2795.47
+Def C00	19971645.49	0.0561%	1140.06
+Def C00	19971656.19	0.0561%	165.04
+Def C00	19971657.09	0.0561%	155.35
+All C00	19973413.70	0.0473%	4293.28
+All C00	19974175.97	0.0435%	9455.69
+All C00	19974177.19	0.0435%	733.44
+All C00	19974180.02	0.0435%	766.91
+Def C08	19974585.45	0.0414%	2559.30
+Def C10	19975711.27	0.0358%	5131.98
+Def C12	19976111.11	0.0338%	3931.73
+Def C14	19976520.40	0.0317%	4355.10
+Def C16	19976898.82	0.0298%	4288.43
+Def C18	19977508.51	0.0268%	8964.28
+Def C20	19978299.28	0.0228%	10901.51
+Def C22	19978577.43	0.0214%	6126.33
+Def C24	19978738.27	0.0206%	5478.57
+Def C26	19978861.69	0.0200%	5556.20
+Def C28	19979011.25	0.0193%	10791.34
+Def C30	19979133.77	0.0186%	6802.63
+Def C32	19979135.92	0.0186%	1676.74
+Def C34	19979382.66	0.0174%	32465.95
+Def C36	19979384.65	0.0174%	7018.06
+Def C38	19979507.22	0.0168%	36408.27
+Def C40	19979571.48	0.0165%	25237.98
+Def C42	19979574.20	0.0164%	9143.59
+Def C44	19979576.01	0.0164%	12623.30
+All C00	19979583.83	0.0164%	1316.49

Algorithm 71. *A very general scheme for collecting cuts*

initialize an empty list $\mathcal{L}$ of cuts;
for selected small integers d, linear mappings $\phi : \mathbb{R}^m \to \mathbb{R}^d$, and finite subsets $\mathcal{T}$ of $\mathbb{R}^d$ such that $\phi(\mathcal{S}) \subseteq \mathcal{T}$
do **if** $\phi(x^*)$ lies outside the convex hull of $\mathcal{T}$
then find a vector a and a scalar b such that
$$\mathcal{T} \subset \{\overline{x} : a^T\overline{x} \leq b\} \text{ and } a^T\phi(x^*) > b;$$
add the cut $a^T\phi(x) \leq b$ to $\mathcal{L}$;
end
end
return $\mathcal{L}$;

The trick of trying to separate x^* from $\mathcal{S}$ by separating $\phi(x^*)$ from $\mathcal{T}$ was used previously by Crowder, Johnson, and Padberg [17] in the context of integer linear programming, where $\mathcal{S}$ consists of all integer solutions of some explicitly recorded system

$$Ax = b,\ \ell \le x \le u \tag{31}$$

and x^* satisfies (31) in place of x. Crowder, Johnson, and Padberg consider systems (31) such that A is sparse and $\ell = 0, u = e$; for each equation $\alpha^T x = \beta$ in the system $Ax = b$, they consider the set $\mathcal{T}$ of all integer solutions of

$$\alpha^T x = \beta,\ \ 0 \le x \le e$$

restricted on components x_j such that $\alpha_j \neq 0$; with $\phi(x)$ standing for the restriction of x on these components, they try to separate x^* from $\mathcal{S}$ by separating $\phi(x^*)$ from $\mathcal{T}$. In the attempt to separate $\phi(x^*)$ from $\mathcal{T}$, they use exclusively inequalities that match certain prescribed templates and they use special-purpose separation algorithms to find such cuts.

Boyd [8,9] starts out with the choices of ϕ and $\mathcal{T}$ made by Crowder, Johnson, and Padberg, but then he separates $\phi(x^*)$ from $\mathcal{T}$ by a general-purpose procedure. He solves the problem

$$\begin{array}{ll} \text{maximize} & z \\ \text{subject to} & z - a^T\phi(x^*) + a^T\phi(x) \le 0 \text{ for all } x \text{ in } \mathcal{T}, \\ & ||a||_1 \le \gamma,\ ||a||_\infty \le 1 \end{array}$$

with a prescribed constant γ by a method that is essentially the simplex method (in particular, the value of z increases with each nondegenerate iteration); to access $\mathcal{T}$, he uses an oracle implemented by a dynamic programming algorithm. If the optimal value z^* turns out to be positive, then he returns the cut

$$a^T\phi(x) \le a^T\phi(x^*) - z^*,$$

which he calls a *Fenchel cutting plane.*

The technique described in Sect. 4.1 provides another implementation of the body of the **for** loop in Algorithm 71 for a fairly general class of sets $\mathcal{T}$: it requires only an efficient oracle that, given a vector a in $\mathbb{R}^d$, returns either an $\overline{x}$ in $\mathcal{T}$ that maximizes $a^T\overline{x}$ or the message "infeasible" indicating that $\mathcal{T}$ is empty. This technique – presented as Algorithm 41 in the special case where the set of all strongly constrained tangled tours is substituted for $\mathcal{T}$ – is reviewed as Algorithm 72: function SEPARATE, given (by means of an efficient maximization oracle) a finite subset $\mathcal{T}$ of some $\mathbb{R}^d$ and given a point $\overline{x}^*$ in $\mathbb{R}^d$, returns either the message "$\overline{x}^*$ is in the convex hull of $\mathcal{T}$" or a vector a and a scalar b such that

$$\mathcal{T} \subset \{\overline{x} : a^T\overline{x} \le b\} \text{ and } a^T\overline{x}^* > b.$$

Algorithm 72. SEPARATE($\overline{x}^*, \mathcal{T}$)

if $\mathcal{T} \neq \emptyset$
then $A =$ any $d \times 1$ matrix whose single column is an element of $\mathcal{T}$;
repeat if the linear programming problem

$$\begin{aligned} &\text{minimize } u^T e + v^T e \\ &\text{subject to } \quad -a^T A + b e^T \geq 0, \\ &\qquad\qquad\quad a^T \overline{x}^* - b = 1, \\ &\qquad\qquad -a^T + u^T - v^T = 0, \\ &\qquad\qquad\quad u^T, v^T \geq 0 \end{aligned}$$

has an optimal solution
then find an element $\overline{x}$ of $\mathcal{T}$ that maximizes $a^T \overline{x}$;
if $a^T \overline{x} \leq b$
then return a and b;
else add $\overline{x}$ to A as a new column;
end
else return the message
"$\overline{x}^*$ is in the convex hull of $\mathcal{T}$";
end
end
else return 0 and -1;
end

One way of solving the linear programming problem in each iteration of the **repeat** loop in Algorithm 72 is to apply the simplex method to its dual with relatively few rows,

$$\begin{aligned} &\text{maximize } s \\ &\text{subject to } \quad s\overline{x}^* - A\lambda + w \quad = 0, \\ &\qquad\qquad\quad -s + e^T \lambda \quad = 0, \\ &\qquad\qquad\quad \lambda \geq 0, \quad -e \leq w \leq e, \end{aligned} \tag{32}$$

just as Concorde does in its implementation of Algorithm 41.

Modifications of SEPARATE($\overline{x}^*, \mathcal{T}$) used by Concorde in its implementation of Algorithm 31 may also apply to implementations of Algorithm 71. In particular, if we have at our disposal a nonempty family $\mathcal{F}$ of row vectors $[v^T, w]$ such that

$$\mathcal{T} \subset \{\overline{x} : v^T \overline{x} \geq w\} \text{ and } v^T \overline{x}^* = w,$$

then we may test the condition

$$\overline{x}^* \text{ lies outside the convex hull of } \mathcal{T}$$

in Algorithm 71 by calling SEPARATE($\overline{x}^*, \mathcal{T}^*$) with

$$\mathcal{T}^* = \{\overline{x} \in \mathcal{T} : v^T \overline{x} = w \text{ for all } [v^T, w] \text{ in } \mathcal{F}\}.$$

(In Concorde, $\mathcal{T}^*$ is the set of all strongly constrained tangled tours through $\overline{V}$.) The point of this substitution is that SEPARATE$(\overline{x}^*, \mathcal{T}^*)$ tends to run faster as the dimension of $\{\overline{x} \in \mathbb{R}^d : v^T\overline{x} = w \text{ for all } [v^T, w] \text{ in } \mathcal{F}\}$ decreases; if most of our choices of d, ϕ, and $\mathcal{T}$ in Algorithm 71 place $\overline{x}^*$ in the convex hull of $\mathcal{T}$, then we may save time even if each successful call of SEPARATE$(\overline{x}^*, \mathcal{T}^*)$ is followed by a call of SEPARATE$(\overline{x}^*, \mathcal{T})$. In fact, if SEPARATE$(\overline{x}^*, \mathcal{T}^*)$ returns an inequality $a^{*T}\overline{x} \le b^*$ that separates $\overline{x}^*$ from $\mathcal{T}^*$, then there is no need to call SEPARATE$(\overline{x}^*, \mathcal{T})$: for every nonnegative M such that

$$M \ge \frac{a^{*T}\overline{x} - b^*}{\sum(v^T\overline{x} - w : [v^T, w] \in \mathcal{F})} \quad \text{for all } \overline{x} \text{ in } \mathcal{T} - \mathcal{T}^*,$$

the inequality

$$a^{*T}\overline{x} - M \cdot \sum(v^T\overline{x} - w : [v^T, w] \in \mathcal{F}) \le b^*$$

separates $\overline{x}^*$ from $\mathcal{T}$.

Furthermore, as long as $\overline{x}^*$ is in the affine hull of $\mathcal{T}$, every cut separating $\overline{x}^*$ from $\mathcal{T}$ may be converted to a cut that induces a facet of the convex hull of $\mathcal{T}$ by the techniques described in Sect. 4.3, where the set of all moderately tangled tours through $\overline{V}$ plays the role of $\mathcal{T}$. Specifically, Algorithm 73, given any cut $a^T\overline{x} \le b$, returns a cut inducing a facet of the convex hull of $\mathcal{T}$. Here, a default $\mathcal{I}$ is supplied by the optimal basis of problem (32), a default $\mathcal{C}$ is supplied by all $[v^T, w]$ in $\mathcal{F}$ such that $\mathcal{T} \not\subset \{\overline{x} : v^T\overline{x} = w\}$, and TILT $(a, b, v, w, \overline{x}^0)$ is Algorithm 43 with $\mathcal{T}$ substituted for the set of all moderately tangled tours.

Algorithm 73. *From a cut $a^T\overline{x} \le b$ to a facet-inducing cut in general*

$\mathcal{I}$ = an affinely independent subset of $\{\overline{x} \in \mathcal{T} : a^T\overline{x} = b\}$;
$\mathcal{C}$ = a catalog of vectors $[v^T, w]$ such that $\mathcal{T} \subset \{\overline{x} : v^T\overline{x} \ge w\}$, $v^T\overline{x}^* \ge w$, and $v^T\overline{x} > w$ for some $\overline{x}$ in $\mathcal{T}$;
$\overline{x}_0$ = an arbitrary element of $\mathcal{T}$;
for all $[v^T, w]$ in $\mathcal{C}$
do remove $[v^T, w]$ from $\mathcal{C}$
  if $\mathcal{I} \subset \{\overline{x} : v^T\overline{x} = w\}$
  then $(a^+, b^+, \overline{x}^+) =$ TILT $(a, b, v, w, \overline{x}^0)$;
    $a = a^+$, $b = b^+$, $\mathcal{I} = \mathcal{I} \cup \{\overline{x}^+\}$;
  end
end
while $|\mathcal{I}| < \dim \mathcal{T}$
do $\overline{x}^0$ = an element of $\mathcal{T} - \mathcal{I}$ such that $\mathcal{I} \cup \{\overline{x}^0\}$ is affinely independent;
  if $a^T\overline{x}^0 = b$
  then replace $(a, b; \mathcal{I})$ by $(a, b; \mathcal{I} \cup \{\overline{x}^0\})$;
  else find a nonzero vector v and a number w such that
    $v^T\overline{x} - w = 0$ for all $\overline{x}$ in $\mathcal{I} \cup \{\overline{x}^0\}$,
    $v^T\overline{x} - w \ne 0$ for some $\overline{x}$ in $\mathcal{T}$;

$(a^+, b^+, \overline{x}^+) =$ TILT $(a, b, v, w, \overline{x}^0)$;
$(a^-, b^-, \overline{x}^-) =$ TILT $(a, b, -v, -w, \overline{x}^0)$;
if $a^{+T}\overline{x}^* - b^+ \geq a^{-T}\overline{x}^* - b^-$
then replace $(a, b; \mathcal{I})$ by $(a^+, b^+; \mathcal{I} \cup \{\overline{x}^+\})$;
else replace $(a, b; \mathcal{I})$ by $(a^-, b^-; \mathcal{I} \cup \{\overline{x}^-\})$;
end
end
end
return a and b;

By the way, Algorithm 73 provides a constructive proof of the following corollary of a classic theorem of Minkowski ([50] Section 19):

Theorem 1. *Let $\mathcal{T}$ be a finite subset of some $\mathbb{R}^d$, let $\overline{x}^*$ be a point in the affine hull of $\mathcal{T}$, and let an inequality $a^T\overline{x} \leq b$ separate $\mathcal{T}$ from $\overline{x}^*$ in the sense that*

$$\mathcal{T} \subset \{\overline{x} : a^T\overline{x} \leq b\} \text{ and } a^T\overline{x}^* > b.$$

Then there is an inequality $\alpha^T\overline{x} \leq \beta$ that separates $\mathcal{T}$ from $\overline{x}^*$, induces a facet of the convex hull of $\mathcal{T}$, and satisfies

$$\{\overline{x} \in \mathcal{T} : a^T\overline{x} = b\} \subseteq \{\overline{x} \in \mathcal{T} : \alpha^T\overline{x} = \beta\}.$$

Success of Algorithm 71 hinges on the ability to make choices of ϕ and $\mathcal{T}$ (which may be guided by x^* and $\mathcal{S}$) in such a way that

(i) chances of $\phi(x^*)$ falling outside the convex hull of $\mathcal{T}$ are reasonable and

(ii) cuts $a^T\phi(x) \leq b$ collected in $\mathcal{L}$ are not too weak.

Our way (described in Sect. 5) of making these choices in the special case where $\mathcal{S}$ is the set of all tours through a set V meets both of these criteria. With respect to (i), it is adequate: typically, one out of fifty to a hundred of our choices of ϕ and $\mathcal{T}$ makes $\phi(x^*)$ fall outside the convex hull of $\mathcal{T}$. With respect to (ii), it could hardly be better: there is no known counterexample to the conjecture (implicit in Naddef and Rinaldi [53]) that

$a^T\phi(x) \leq b$ induces a facet of the convex hull of $\mathcal{S}$
whenever $a^T\overline{x} \leq b$ induces a facet of the convex hull of $\mathcal{T}$ and $a^T\phi(x^*) > b$.

This may be a major reason behind the success of our application of Algorithm 71 to the traveling salesman problem.

If one were to use Algorithm 71 in the context of integer linear programming, where $\mathcal{S}$ consists of all integer solutions of some explicitly recorded system

$$Ax = b,\ \ell \leq x \leq u,$$

then one would have to design a way of making choices of ϕ and $\mathcal{T}$. One option is to choose each ϕ and $\mathcal{T}$ by choosing a $d \times m$ integer matrix P, setting $\phi(x) = Px$, and letting $\mathcal{T}$ consist of all integer vectors $\overline{x}$ such that some x satisfies

$$Ax = b,\ \ell \leq x \leq u, Px = \overline{x}.$$

Maximizing a linear function over $\mathcal{T}$ amounts to solving a mixed integer linear programming problem in d integer and m non-integer variables; as long as d is small, this problem can be solved quickly. As long as at least one row r^T of P satisfies

$$\lfloor \max\{r^T x : Ax = b,\ \ell \leq x \leq u\} \rfloor < r^T x^*, \tag{33}$$

$\phi(x^*)$ falls outside the convex hull of $\mathcal{T}$; Gomory's methods mentioned in Sect. 2 provide vectors r^T with property (33) at insignificant computational cost. We have not carried out any experiments with this scheme.

Acknowledgments. We wish to thank Adrian Bondy, Martin Farach-Colton, Denis Naddef, Yves Pochet, and Günter Rote for their helpful comments on earlier versions of this manuscript.

References

1. Applegate, D., Bixby, R., Chvátal, V., Cook, W.: Finding cuts in the TSP (A preliminary report). DIMACS Technical Report 95-05, 1995. Available at `ftp://dimacs.rutgers.edu/pub/dimacs/TechnicalReports/TechReports/1995/`
2. Applegate, D., Bixby, R., Chvátal, V., Cook, W.: On the solution of traveling salesman problems. Documenta Mathematica **Extra Volume** (Proceedings of the International Congress of Mathematicians), 645–656, 1998. Also available at `http://www.mathematik.uni-bielefeld.de/documenta/xvol-icm/17/17.html`
3. Applegate, D., Bixby, R., Chvátal, V., Cook, W.: Concorde, 1999. Available at `http://www.math.princeton.edu/tsp/concorde.html`
4. Applegate, D., Bixby, R., Chvátal, V., Cook, W.: Solving Traveling Salesman Problems. To appear.
5. Balas, E.: Facets of the knapsack polytope. Mathematical Programming **8**, 146–164, 1975.
6. Batoukov, R., Sørevik, T.: A generic parallel branch and bound environment on a network of workstations. In: Proceedings of HiPer'99, pp. 474–483, 1999. Also available at `http://www.ii.uib.no/~tors/publications/`
7. Bock, F.: An algorithm for solving 'traveling-salesman' and related network optimization problems. Research Report, Armour Research Foundation. Presented at the Operations Research Society of America Fourteenth National Meeting, St. Louis, October 24, 1958.
8. Boyd, E.A.: Generating Fenchel cutting planes for knapsack polyhedra. SIAM Journal of Optimization **3**, 734–750, 1993.
9. Boyd, E.A.: Fenchel cutting planes for integer programs. Operations Research **42**, 53–64, 1994.

10. Carr, R.: Separating clique trees and bipartition inequalities having a fixed number of handles and teeth in polynomial time. Mathematics of Operations Research **22**, 257–265, 1997.
11. Christof, T., Reinelt, G.: Parallel cutting plane generation for the TSP. In: Parallel Programming and Applications (P. Fritzson and L. Finmo, eds.), IOS Press, pp. 163–169, 1995.
12. Christof, T., Reinelt, G.: Combinatorial optimization and small polytopes. Top **4**, 1–64, 1996.
13. Clochard, J.-M., Naddef, D.: Using path inequalities in a branch and cut code for the symmetric traveling salesman problem. In: Third IPCO Conference, (G. Rinaldi and L. Wolsey, eds.), pp. 291–311, 1993.
14. Cornuéjols, G., Fonlupt, J., Naddef, D.: The traveling salesman problem on a graph and some related integer polyhedra. Mathematical Programming **33**, 1–27, 1985.
15. Craven, B.D.: Fractional Programming. Heldermann, Berlin, 1988.
16. Croes, G.A.: A method for solving traveling-salesman problems. Operations Research **6**, 791–812, 1958.
17. Crowder, H., Johnson, E.L., Padberg, M.: Solving large-scale zero-one linear programming problems. Operations Research **31**, 803–834, 1983.
18. Crowder, H., Padberg, M.W.: Solving large-scale symmetric travelling salesman problems to optimality. Management Science **26**, 495–509, 1980.
19. Dantzig, G., Fulkerson, R., Johnson, S.: Solution of a large-scale traveling salesman problem. Operations Research **2**, 393–410, 1954.
20. Eastman, W.L.: Linear programming with pattern constraints. Ph.D. Thesis, Harvard University, 1958.
21. Edmonds, J.: Maximum matching and a polyhedron with 0,1-vertices. Journal of Research of the National Bureau of Standards **69B**, 125–130, 1965.
22. Fleischer, L.K., Tardos, É.: Separating Maximally Violated Combs in Planar Graphs. Mathematics of Operations Research **24**, 130–148, 1999.
23. Fleischmann, B.: A cutting plane procedure for the travelling salesman problem on road networks. European Journal of Operational Research **21**, 307–317, 1985.
24. Fleischmann, B.: A new class of cutting planes for the symmetric travelling salesman problem. Mathematical Programming **40**, 225–246, 1988.
25. Ford, L.R.Jr., Fulkerson, D.R.: A suggested computation for maximal multicommodity networks flows. Management Science **5**, 97–101, 1958.
26. Gomory, R.E.: Outline of an algorithm for integer solutions to linear programs. Bulletin of the American Mathematical Society **64**, 275–278, 1958.
27. Gomory, R.E.: Solving linear programs in integers. In: Combinatorial Analysis (R. E. Bellman and M. Hall, Jr., eds.), Proceedings of the Symposia on Applied Mathematics **X**, pp. 211–216, 1960.
28. Gomory, R.E.: An algorithm for integer solutions to linear programs. In: Recent Advances in Mathematical Programming (R. L. Graves and P. Wolfe, eds.), McGraw-Hill, New York, pp. 269–302, 1963.
29. Gomory, R.E.: Some polyhedra related to combinatorial problems. Linear Algebra and Its Applications **2**, 451–558, 1969.
30. Grötschel, M.: Polyedrische Charakterisierungen kombinatorischer Optimierungsprobleme, Anton Hain Verlag, Meisenheim/Glan, 1977.
31. Grötschel, M.: On the symmetric travelling salesman problem: solution of a 120-city problem. Mathematical Programming Study **12**, 61–77, 1980.
32. Grötschel, M., O. Holland.: Solution of large-scale symmetric travelling salesman problems. Mathematical Programming **51**, 141–202, 1991.

33. Grötschel, M., Jünger, M., Reinelt, G.: A cutting plane algorithm for the linear ordering problem. Operations Research **32**, 1195–1220, 1984.
34. Grötschel, M., Padberg, M.W.: On the Symmetric Travelling Salesman Problem, Report No.7536-OR, Institut für Ökonometrie und Operations Research, Universität Bonn, 1975.
35. Grötschel, M., Padberg, M.W.: On the symmetric travelling salesman problem I: Inequalities. Mathematical Programming **16**, 265–280, 1979.
36. Grötschel, M., Padberg, M.W.: On the symmetric travelling salesman problem II: Lifting theorems and facets. Mathematical Programming **16**, 281–302, 1979.
37. Grötschel, M., Pulleyblank, W.: Clique tree inequalities and the symmetric travelling salesman problem. Mathematics of Operations Research **11**, 537–569, 1986.
38. Hammer, P.L., Johnson, E.L., Peled, U.N.: Facets of regular 0-1 polytopes. Mathematical Programming **8**, 179–206, 1975.
39. Hong, S.: A linear programming approach for the traveling salesman problem, Ph.D. Thesis, The Johns Hopkins University, 1972.
40. Jewell, W.S.: Optimal flow through networks. Interim Technical Report No. 8, Massachusetts Institute of Technology, 1958.
41. Land, A.: The solution of some 100-city travelling salesman problems. Unpublished manuscript, 1979.
42. Land, A.H., Doig, A.G.: An automatic method of solving discrete programming problems. Econometrica **28**, 497–520, 1960.
43. Letchford, A.N.: Separating a superclass of comb inequalities in planar graphs. Mathematics of Operations Research **25**, 443–454, 2000.
44. Little, J.D.C., Murty, K.G., Sweeney, D.W., Karel, C.: An algorithm for the traveling salesman problem. Operations Research **11**, 972–989, 1963.
45. Martin, G.T.: An accelerated euclidean algorithm for integer linear programming. In: Recent advances in mathematical programming (R. L. Graves and P. Wolfe, eds.), McGraw-Hill, pp. 311–318, 1963.
46. Martin, G.T.: Solving the traveling salesman problem by integer linear programming. Operations Research **14** (Supplement 1), Abstract WA7.10, 1966.
47. Maurras, J.F.: Some results on the convex hull of Hamiltonian cycles of symmetric complete graphs. In: Combinatorial Programming: Methods and Applications (B. Roy, ed.), Reidel, Dordrecht, pp. 179–190, 1975.
48. Miliotis, P.: Integer programming approaches to the travelling salesman problem. Mathematical Programming **10**, 367–378, 1976.
49. Miliotis, P.: Using cutting planes to solve the symmetric travelling salesman problem. Mathematical Programming **15**, 177–188, 1978.
50. Minkowski, H.: Geometrie der Zahlen (Erste Lieferung). Teubner, Leipzig, 1896. Reprinted: Chelsea, New York, 1953.
51. Naddef, D.: Handles and teeth in the symmetric traveling salesman polytope. In: Polyhedral combinatorics (W. Cook and P. D. Seymour, eds.), DIMACS Series in Mathematics and Theoretical Computer Science **1**, American Mathematical Society, pp. 61–74, 1990.
52. Naddef, D., Rinaldi, G.: The symmetric traveling salesman polytope and its graphical relaxation: Composition of valid inequalities. Mathematical Programming **51**, 359–400, 1991.
53. Naddef, D., Rinaldi, G.: The graphical relaxation: A new framework for the symmetric traveling salesman polytope. Mathematical Programming **58**, 53–88, 1992.

54. Naddef, D., Thienel, S.: Efficient separation routines for the symmetric traveling salesman problem I: General tools and comb separation. Working paper, 1999. Available at
http://www-id.imag.fr/Laboratoire/Membres/Naddef_Denis/perso.html
55. Naddef, D., Thienel, S.: Efficient separation routines for the symmetric traveling salesman problem II: Separating multi handle inequalities. Working paper, 1999. Available at
http://www-id.imag.fr/Laboratoire/Membres/Naddef_Denis/perso.html
56. Padberg, M.W.: On the facial structure of set packing polyhedra. Mathematical Programming **5**, 199–215, 1973.
57. Padberg, M.W.: A note on zero-one programming. Operations Research **23**, 833–837, 1975.
58. Padberg, M.W., Hong, S.: On the symmetric travelling salesman problem: a computational study. Mathematical Programming Study **12**, 78–107, 1980.
59. Padberg, M.W., Rao, M.R.: Odd minimum cut-sets and b-matchings. Mathematics of Operations Research **7**, 67–80, 1982.
60. Padberg, M.W., Rinaldi, G.: Optimization of a 532-city symmetric traveling salesman problem by branch and cut. Operations Research Letters **6**, 1–7, 1987.
61. Padberg, M.W., Rinaldi, G.: An efficient algorithm for the minimum capacity cut problem. Mathematical Programming **47**, 1990.
62. Padberg, M.W., Rinaldi, G.: Facet identification for the symmetric traveling salesman polytope. Mathematical Programming **47**, 219–257, 1990.
63. Padberg, M.W., Rinaldi, G.: A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. SIAM Review **33**, 60–100, 1991.
64. Reinelt, G.: TSPLIB – A traveling salesman problem library. ORSA Journal on Computing **3**, 376–384, 1991. An updated version is available at
http://www.iwr.uni-heidelberg.de/iwr/comopt/software/ TSPLIB95/.
65. Rossman, M.J., Twery, R.J.: A solution to the travelling salesman problem. Operations Research **6**, p.687, Abstract E3.1.3, 1958.
66. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, Chichester, 1986.
67. Stancu-Minasian, I.M.: Fractional Programming. Kluwer, Dordrecht, 1997.
68. Wolsey, L.A.: Faces for a linear inequality in 0-1 variables. Mathematical Programming **8**, 165–178, 1975.
69. Wolsey, L.A.: Facets and strong valid inequalities for integer programs. Operations Research **24**, 367–372, 1975.