Peter Brucker
Sigrid Knust

GOR PUBLICATIONS

Complex Scheduling

Springer

GOR ■ Publications

Titles in the Series

H.-O. Günther and *P. v. Beek* (Eds.)
Advanced Planning and Scheduling
Solutions in Process Industry
VI, 426 pages. 2003. ISBN 3-540-00222-7

J. Schönberger
Operational Freight Carrier Planning
IX, 164 pages. 2005. ISBN 3-540-25318-1

C. Schwindt
Resource Allocation
in Project Management
X, 193 pages. 2005. ISBN 3-540-25410-2

Peter Brucker
Sigrid Knust

Complex Scheduling

With 135 Figures
and 3 Tables

Springer

Professor Dr. Peter Brucker
Juniorprofessor Dr. Sigrid Knust
Universität Osnabrück
Fachbereich Mathematik/Informatik
Albrechtstraße 28
49069 Osnabrück
E-mail: pbrucker@uni-osnabrueck.de
E-mail: sigrid.knust@informatik.uni-osnabrueck.de

Cataloging-in-Publication Data
Library of Congress Control Number: 2005938499

ISBN-10 3-540-29545-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-29545-7 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media
springeronline.com

Printed in Germany

Cover design: Erich Kirchner
Production: Helmut Petri
Printing: Strauss Offsetdruck

SPIN 11571964 Printed on acid-free paper – 42/3153 – 5 4 3 2 1 0

Preface

Scheduling problems have been investigated since the late fifties. Two types of applications have mainly motivated research in this area: project planning and machine scheduling. While in machine scheduling a large number of specific scheduling situations depending on the machine environment and the job characteristics have been considered, the early work in project planning investigated scheduling situations with precedence constraints between activities assuming that sufficient resources are available to perform the activities. More recently, in project scheduling scarce resources have been taken into account leading to so-called resource-constrained project scheduling problems. On the other hand, also in machine scheduling more general and complex problems have been investigated. Due to these developments today both areas are much closer to each other. Furthermore, applications like timetabling, rostering or industrial scheduling are connected to both areas.

This book deals with such complex scheduling problems and methods to solve them. It consists of three parts: The first part (Chapters 1 and 2) contains a description of basic scheduling models with applications and an introduction into discrete optimization (covering complexity, shortest path algorithms, linear programming, network flow algorithms and general optimization methods). In the second part (Chapter 3) resource-constrained project scheduling problems are considered. Especially, methods like constraint propagation, branch-and-bound algorithms and heuristic procedures are described. Furthermore, lower bounds and general objective functions are discussed. In the last part (Chapter 4) generalizations of the job-shop problem are covered leading to applications like job-shop problems with flexible machines, transport robots or with limited buffers. Heuristic methods to solve such complex scheduling problems are presented.

We are indebted to many people who have helped in preparing this book. Students in our courses during the last two years at the University of Osnabrück have given suggestions for improving earlier versions of this material. Andreas Drexl and Silvia Heitmann read carefully parts of the book and gave constructive comments.

We are grateful to the Deutsche Forschungsgemeinschaft for supporting the research that underlies much of this book. In addition we like to acknowledge the advice and support of Philippe Baptiste, Rainer Kolisch, Rolf H. Möhring, Klaus Neumann and Erwin Pesch.

Osnabrück, Oktober 2005 Peter Brucker, Sigrid Knust

Contents

Chapter 1

Scheduling Models

The so-called resource-constrained project scheduling problem (RCPSP) is one of the basic complex scheduling problems. In Section 1.1 we introduce this problem and some of its generalizations. Machine scheduling problems, which may be considered as special cases, are introduced in Section 1.2.

1.1 The RCPSP and some Generalizations

The resource-constrained project scheduling problem (RCPSP) is a very general scheduling problem which may be used to model many applications in practice (e.g. a production process, a software project, a school timetable, the construction of a house or the renovation of an airport). The objective is to schedule some activities over time such that scarce resource capacities are respected and a certain objective function is optimized. Examples for resources may be machines, people, rooms, money or energy, which are only available with limited capacities. As objective functions e.g. the project duration, the deviation from deadlines or costs concerning resources may be minimized.

1.1.1 The RCPSP

The **resource-constrained project scheduling problem (RCPSP)** may be formulated as follows. Given are n **activities** (jobs) $i = 1, \ldots, n$ and r **renewable resources** $k = 1, \ldots, r$. A constant amount of R_k units of resource k is available at any time. Activity i must be processed for p_i time units. During this time period a constant amount of r_{ik} units of resource k is occupied. All data are assumed to be integers.

Furthermore, **precedence constraints** are defined between some activities. They are given by relations $i \rightarrow j$, where $i \rightarrow j$ means that activity j cannot start before activity i is completed.

The objective is to determine starting times S_i for the activities $i = 1, \ldots, n$ in such a way that

- at each time t the total resource demand is less than or equal to the resource availability of each resource $k = 1, \ldots, r$,
- the given precedence constraints are fulfilled, i.e. $S_i + p_i \leq S_j$ if $i \rightarrow j$, and
- the **makespan** $C_{\max} = \max\limits_{i=1}^{n} \{C_i\}$ is minimized, where $C_i := S_i + p_i$ is assumed to be the completion time of activity i.

The fact that an activity which starts at time S_i finishes at time $C_i = S_i + p_i$ implies that activities are not preempted. We may relax this condition by allowing **preemption** (activity splitting). In this case the processing of any activity may be interrupted and resumed later. It will be stated explicitly if we consider models with preemption.

It is often useful to add a unique **dummy starting activity** 0 and a unique **dummy termination activity** $n + 1$, which indicate the start and the end of the project, respectively. The dummy activities need no resources and have processing time zero. In order to impose $0 \rightarrow i \rightarrow n + 1$ for all activities $i = 1, \ldots, n$ we set $0 \rightarrow i$ for all activities i without any predecessor $j \rightarrow i$ and $i \rightarrow n+1$ for all activities i without any successor $i \rightarrow j$. Then S_0 is the starting time of the project and S_{n+1} may be interpreted as the makespan of the project. Usually we set $S_0 := 0$.

If preemption is not allowed, the vector $S = (S_i)$ defines a **schedule** of the project. S is called **feasible** if all resource and precedence constraints are fulfilled.

We may represent the structure of a project by a so-called **activity-on-node network** $G = (V, A)$, where the vertex set $V := \{0, 1, \ldots, n, n + 1\}$ contains all activities and the set of arcs $A = \{(i, j) \mid i, j \in V; i \rightarrow j\}$ represents the precedence constraints. Each vertex $i \in V$ is weighted with the corresponding processing time p_i.

For each activity i we define

$$Pred(i) := \{j \mid (j, i) \in A\} \text{ and } Succ(i) := \{j \mid (i, j) \in A\}$$

as the sets of **predecessors** and **successors** of activity i, respectively.

Another representation of projects is based on so-called **activity-on-arc networks** where each activity is modeled by an arc. Since this representation has some disadvantages, in most cases activity-on-node networks are preferred.

Example 1.1: Consider a project with $n = 4$ activities, $r = 2$ resources with capacities $R_1 = 5, R_2 = 7$, a precedence relation $2 \rightarrow 3$ and the following data:

i	1	2	3	4
p_i	4	3	5	8
r_{i1}	2	1	2	2
r_{i2}	3	5	2	4

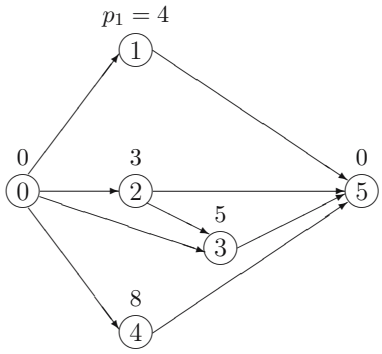
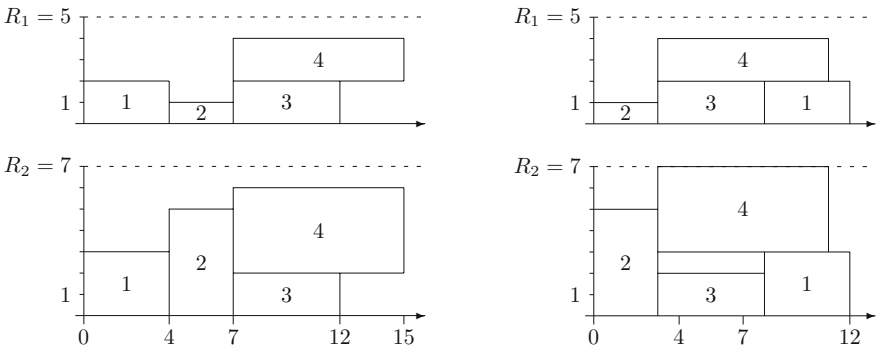


Figure 1.1: The activity-on-node network for Example 1.1



(a) A feasible schedule (b) An optimal schedule

Figure 1.2: Two feasible schedules for Example 1.1

Figure 1.1 illustrates the corresponding activity-on-node network, where the dummy activities 0 and 5 have been added and the vertices are weighted with the processing times. In Figure 1.2(a) a so-called **Gantt chart** of a feasible schedule with $C_{\max} = 15$ is drawn. This schedule does not minimize the makespan, since by moving activity 1 to the right, a shorter schedule is obtained. An optimal schedule with makespan $C_{\max} = 12$ is shown in (b). □

In the following we will discuss different generalizations of the RCPSP.

Generalized precedence constraints

A precedence relation $i \rightarrow j$ with the meaning $S_i + p_i \leq S_j$ may be generalized by a start-start relation of the form

$$S_i + d_{ij} \leq S_j \tag{1.1}$$

with an arbitrary integer number $d_{ij} \in \mathbb{Z}$. The interpretation of relation (1.1) depends on the sign of d_{ij}:

- If $d_{ij} \geq 0$, then activity j cannot start before d_{ij} time units after the start of activity i. This means that activity j does not start before activity i and d_{ij} is a minimal distance (time-lag) between both starting times (cf. Figure 1.3(a)).
- If $d_{ij} < 0$, then the earliest start of activity j is $-d_{ij}$ time units before the start of activity i, i.e. activity i cannot start more than $-d_{ij}$ time units later than activity j. If $S_j \leq S_i$, this means that $|d_{ij}|$ is a maximal distance between both starting times (cf. Figure 1.3(b)).

(a) positive time-lag (b) negative time-lag

Figure 1.3: Positive and negative time-lags

If $d_{ij} > 0$ holds, the value is also called a **positive time-lag**; if $d_{ij} < 0$, it is called a **negative time-lag**. Time-lags d_{ij} may be incorporated into the activity-on-node network G by adding all arcs $i \to j$ to G and weighting them with the distances d_{ij}.

Relations (1.1) are very general timing relations between activities. For example, (1.1) with $d_{ij} = p_i$ is equivalent to the precedence relation $i \to j$. More generally, besides start-start relations also finish-start, finish-finish or start-finish relations may be considered. But if no preemption is allowed, any type of these relations can be transformed to any other type. For example, finish-start relations $C_i + l_{ij} \leq S_j$ with finish-start time-lags l_{ij} can be transformed into start-start relations $S_i + p_i + l_{ij} \leq S_j$ by setting $d_{ij} := p_i + l_{ij}$.

Generalized precedence relations may for example be used in order to model certain timing restrictions for a chemical process. If $S_i + p_i + l_{ij} \leq S_j$ and $S_j - u_{ij} - p_i \leq S_i$ with $0 \leq l_{ij} \leq u_{ij}$ is required, then the time between the completion time of activity i and the starting time of activity j must be at least l_{ij} but no more than u_{ij}. This includes the special case $0 \leq l_{ij} = u_{ij}$ where activity j must start exactly l_{ij} time units after the completion of activity i. If $l_{ij} = u_{ij} = 0$, then activity j must start immediately after activity i finishes (**no-wait constraint**).

Also release times r_i and deadlines d_i of activities i can be modeled by relations of the form (1.1). While a **release time** r_i is an earliest starting time for activity i, a **deadline** d_i is a latest completion time for i. To model release times we add the restrictions $S_0 + r_i \leq S_i$. To model deadlines we add the restrictions $S_i - (d_i - p_i) \leq S_0$. In both cases we assume that $S_0 = 0$. If $r_i \leq d_i$ for a release time r_i and a deadline d_i, then the interval $[r_i, d_i]$ is called a **time window** for activity i. Activity i must be processed completely within its time window.

Time-dependent resource profiles

A time period t is defined as the time interval $[t-1, t[$ for $t = 1, 2, \ldots, T$, where T denotes the time horizon given for the project. Until now we assumed that R_k units of resource k are available in each time period and that r_{ik} units of resource k are occupied in each time period t in which activity i is processed. This assumption may be generalized using the concept of **time-dependent resource profiles**. In this situation the availability $R_k(t)$ of resource k is a function depending on the time periods t and may be represented by pairs (t_k^μ, R_k^μ) for $\mu = 1, \ldots, m_k$, where $0 = t_k^1 < t_k^2 < \ldots < t_k^{m_k} = T$ are the jump points of the function and R_k^μ denotes the resource capacity in the time interval $[t_k^\mu, t_k^{\mu+1}[$ for $\mu = 1, \ldots, m_k - 1$.

A resource is called **disjunctive** if $R_k(t) \in \{0, 1\}$ for $t = 1, \ldots, T$ holds, otherwise it is called **cumulative**. If resource k is disjunctive, two activities i, j with $r_{ik} = r_{jk} = 1$ cannot be processed simultaneously. $R_k(t) = 0$ states that resource k is not available in time period t.

Time-dependent resource profiles may for example be used in order to model a situation in which the numbers of available workers vary over time (e.g. at weekends less people are working).

Example 1.2: Consider an instance of a project with $n = 6$ activities, $r = 2$ resources, precedence constraints $1 \to 4 \to 5$, $2 \to 3$ and the following data:

i	1	2	3	4	5	6
p_i	2	2	3	2	2	4
r_{i1}	1	0	1	0	1	0
r_{i2}	1	2	3	2	4	2

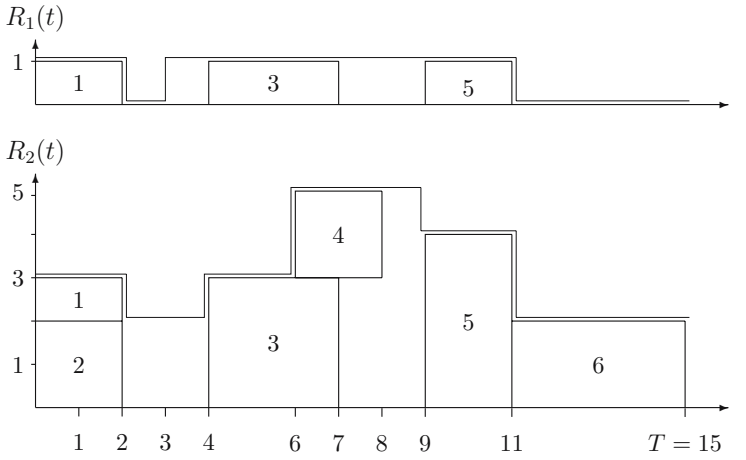


Figure 1.4: A feasible schedule for time-dependent resource profiles

Let the time-dependent resource profile of resource 1 be defined by the pairs $(0,1)$, $(2,0)$, $(3,1)$, $(11,0)$, $(15,0)$, the resource profile of resource 2 by the pairs $(0,3)$, $(2,2)$, $(4,3)$, $(6,5)$, $(9,4)$, $(11,2)$, $(15,0)$.

In Figure 1.4 a feasible schedule is represented by the corresponding Gantt chart. This schedule does not minimize the makespan because by moving activity 3 two units to the right and scheduling activity 6 between activity 1 and activity 3 a feasible schedule with smaller makespan is obtained. □

Multiple modes

In the **multi-mode case** a set $\mathcal{M}_i$ of so-called modes (processing alternatives) is associated with each activity i. The processing time of activity i in mode m is given by p_{im} and the per period usage of a renewable resource k is given by r_{ikm}. One has to assign a mode to each activity and to schedule the activities in the assigned modes.

Multiple modes may for example be used in order to model a situation in which an activity can be fastly processed by many workers or more slowly with less people.

Non-renewable resources

Besides renewable resources like machines or people we may also have so-called **non-renewable resources** like money or energy. While renewable resources are available with a constant amount in each time period again, the availability of non-renewable resources is limited for the whole time horizon of the project. This means that non-renewable resources are consumed, i.e. when an activity i is processed, the available amount R_k of a non-renewable resource k is decreased by r_{ik}.

Non-renewable resources are only important in connection with multi-mode problems, because in the single-mode case the resource requirements of non-renewable resources are schedule independent (the available non-renewable resources may be sufficient or not). On the other hand, in the multi-mode case the resource requirements of non-renewable resources depend on the choice of modes (i.e. feasible and infeasible mode assignments may exist).

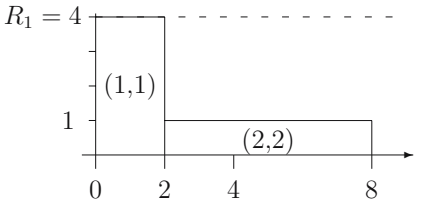
Besides upper bound values R_k for the total usage of a non-renewable resource in some applications also lower bound values L_k may have to be respected. For example, a worker may not only have a maximal but also a minimal working time per month. In such a situation only mode assignments satisfying both bounds are feasible.

To distinguish between renewable and non-renewable resources, if both are needed, we write $\mathcal{K}^\rho$ for the set of renewable resources, $\mathcal{K}^\nu$ for the set of non-renewable resources, r^ρ_{ikm}, R^ρ_k for the renewable resource data and r^ν_{ikm}, R^ν_k for the non-renewable resource data.

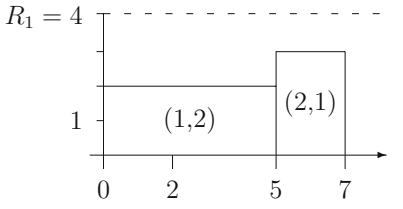
Example 1.3: Consider an instance of a multi-mode project with $n = 2$ activities, where each activity i can be processed in two different modes $m = 1, 2$. We have one renewable resource 1 with capacity $R_1^\rho = 4$ and one non-renewable resource 2 with capacity $R_2^\nu = 10$. Furthermore, the processing times p_{im} of the activities $i = 1, 2$ in modes $m = 1, 2$ are given by $p_{11} = 2$, $p_{12} = 5$, $p_{21} = 2$, $p_{22} = 6$ and the resource requirements r_{ikm} are

$$\begin{array}{llll} r_{111}^\rho = 4, & r_{112}^\rho = 2, & r_{121}^\nu = 6, & r_{122}^\nu = 2 \quad \text{for activity 1, and} \\ r_{211}^\rho = 3, & r_{212}^\rho = 1, & r_{221}^\nu = 5, & r_{222}^\nu = 3 \quad \text{for activity 2.} \end{array}$$

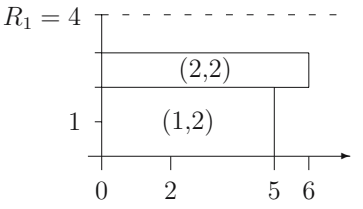
Due to $r_{121}^\nu + r_{221}^\nu = 6+5 = 11 > 10 = R_2^\nu$ the non-renewable resource 2 does not allow that activities 1 and 2 are both processed in mode one. Furthermore, we have $r_{121}^\nu + r_{222}^\nu = 6+3 = 9$, $r_{122}^\nu + r_{221}^\nu = 2+5 = 7$, and $r_{122}^\nu + r_{222}^\nu = 2+3 = 5$. Thus, the remaining three activity-mode combinations for the two activities are feasible: $(1,1)$ with $(2,2)$, $(1,2)$ with $(2,1)$ and $(1,2)$ with $(2,2)$. If we compare the corresponding schedules shown in Figure 1.5, we see that the schedule in (c) has the smallest makespan $C_{\max} = 6$. In this schedule 2+3=5 units of the non-renewable resource 2 are needed.



(a) Schedule with $C_{\max} = 8$



(b) Schedule with $C_{\max} = 7$



(c) Optimal schedule with $C_{\max} = 6$

Figure 1.5: Three feasible schedules for Example 1.3 □

Doubly-constrained resources

A combination of renewable and non-renewable resources are so-called **doubly-constrained resources** which are constrained in each time period and for the whole project. With this type of resources for example a situation can be modeled in which the working time of a worker per day and the working time for the whole project is limited. Since a doubly-constrained resource can be treated by introducing a renewable and a non-renewable resource for it, this type of resources does not have to be considered separately.

Partially renewable resources

A generalization of renewable and non-renewable resources are so-called **partially renewable** resources $\mathcal{K}^\pi$ for which the availability is restricted over subsets of time periods from the set $\{1,\ldots,T\}$ with a given time horizon T. Associated with each partially renewable resource $k \in \mathcal{K}^\pi$ are subsets of time periods $P_k(1),\ldots,P_k(u_k) \subseteq \{1,\ldots,T\}$, where for the time periods in the subset $P_k(\tau)$ $(\tau = 1,\ldots,u_k)$ in total $R_k^\pi(\tau)$ units of resource k are available. In all other time periods which are not contained in the subsets the resource capacity is unlimited.

Furthermore, for each activity i its per-period requirement r_{ik}^π of resource $k \in \mathcal{K}^\pi$ is given. If activity i is (partially) processed in $P_k(\tau)$, it consumes r_{ik}^π units of resource k in each time period of $P_k(\tau)$ in which it is processed.

Renewable resources $\mathcal{K}^\rho$ may be considered as a special case of partially renewable resources by introducing for each resource $k \in \mathcal{K}^\rho$ and for $t = 1,\ldots,T$ subsets $P_k(t) := \{t\}$ consisting of single time periods. Furthermore, the capacity is set to $R_k^\pi(t) := R_k^\rho$ (or $R_k(t)$ in the case of time-dependent resource profiles) and the per-period requirement of activity i is defined by $r_{ik}^\pi := r_{ik}^\rho$.

Also non-renewable resources $\mathcal{K}^\nu$ may be considered as a special case of partially renewable resources by introducing for each resource $k \in \mathcal{K}^\nu$ one subset $P_k(1) := \{1,\ldots,T\}$ covering the complete time horizon. Furthermore, the capacity is set to $R_k^\pi(1) := R_k^\nu$ and the per-period requirement of activity i is defined by $r_{ik}^\pi := r_{ik}^\nu / p_i$.

Example 1.4: Consider an instance with $n = 3$ activities, $r = 2$ partially renewable resources and time horizon $T = 10$. Associated with the resource $k = 1$ are the subsets $P_1(1) = \{1,2,3,4,5\}$ with $R_1^\pi(1) = 6$ and $P_1(2) = \{6,7,8,9,10\}$ with $R_1^\pi(2) = 8$. Associated with the resource $k = 2$ is the subset $P_2(1) = \{3,4,8,9\}$ with $R_2^\pi(1) = 1$, in the remaining time periods resource 2 is not constrained. Furthermore, the activities have the processing times $p_1 = 3, p_2 = 3, p_3 = 1$ and the per-period resource requirements r_{ik}^π are given by

$$r_{11}^\pi = 1,\ r_{21}^\pi = 2,\ r_{31}^\pi = 3, \quad r_{12}^\pi = r_{22}^\pi = r_{32}^\pi = 1.$$

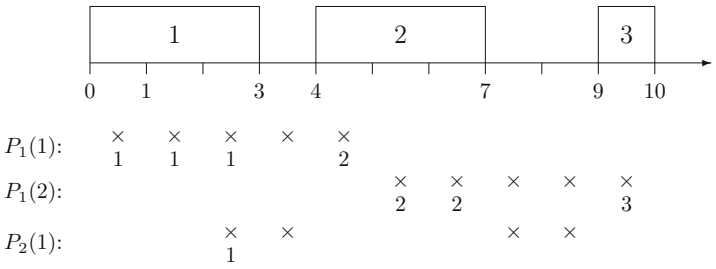


Figure 1.6: Feasible schedule for Example 1.4

A feasible schedule for this instance is shown in Figure 1.6. In this schedule activity 1 needs 3 units of resource 1 in the subset $P_1(1)$ and 1 unit of resource 2 in $P_2(1)$. Activity 2 needs 2 units of resource 1 in $P_1(1)$, $2 \cdot 2 = 4$ units of resource 1 in $P_1(2)$, and activity 3 needs 3 units of resource 1 in $P_1(2)$. Activity 2 cannot start in time period $4 \in P_2(1)$ since resource 2 is already completely consumed in time period $3 \in P_2(1)$ by activity 1. Furthermore, activity 3 cannot be scheduled in a time period of the subset $P_1(1)$ since after scheduling activities 1 and 2 the remaining capacity 1 in $P_1(1)$ is less than $r^{\pi}_{31} = 3$. Note that it would also be feasible to process activity 3 in the time periods 6 or 7 in $P_1(2)$. □

With the concept of partially renewable resources for example we may model a situation in which a worker is allowed to work on at most 5 weekend days per month. In this case a partially renewable resource with capacity 5 is introduced which is constrained over the subset of all weekend time periods.

Setup times

In a scheduling model with **sequence-dependent setup times** the set of all (single-mode) activities is partitioned into q disjoint sets $G_1, \ldots, G_q$, called groups. Associated with each pair (g, h) of group indices is a setup time s_{gh}. For any $i \in G_g$ and $j \in G_h$, if i is processed before j, then the restriction

$$C_i + s_{gh} \leq S_j \tag{1.2}$$

must be satisfied. Setup times may for example be used to model changeover times of a machine which occur when the machine is changed for the production of a different product (e.g. if a painting machine has to be prepared for a different color).

Often one assumes that $s_{gg} = 0$ for all groups G_g and that the setup times satisfy the strong triangle inequality

$$s_{fg} + s_{gh} \geq s_{fh} \quad \text{for all } f, g, h \in \{1, \ldots, q\} \tag{1.3}$$

or the weak triangle inequality

$$s_{fg} + \min_{i \in G_g} \{p_i\} + s_{gh} \geq s_{fh} \quad \text{for all } f, g, h \in \{1, \ldots, q\}. \tag{1.4}$$

Other objective functions

Besides the objective of minimizing the makespan $C_{\max} := \max_{i=1}^{n} \{C_i\}$ one may consider other objective functions $f(C_1, \ldots, C_n)$ depending on the completion times of the activities. Examples are the **total flow time** $\sum_{i=1}^{n} C_i$ or more generally the **weighted (total) flow time** $\sum_{i=1}^{n} w_i C_i$ with non-negative weights $w_i \geq 0$.

Other objective functions depend on **due dates** d_i which are associated with the activities. With the **lateness** $L_i := C_i - d_i$, the **tardiness** $T_i := \max\{0, C_i - d_i\}$, and the **unit penalty** $U_i := \begin{cases} 0, & \text{if } C_i \leq d_i \\ 1, & \text{otherwise} \end{cases}$ the following objective functions are common:

the **maximum lateness**	$L_{\max} := \max_{i=1}^{n} L_i$
the **total tardiness**	$\sum_{i=1}^{n} T_i$
the **total weighted tardiness**	$\sum_{i=1}^{n} w_i T_i$
the **number of late activities**	$\sum_{i=1}^{n} U_i$
the **weighted number of late activities**	$\sum_{i=1}^{n} w_i U_i$

All these objective functions f are monotone non-decreasing in the completion times C_i, i.e. they satisfy $f(C_1, \ldots, C_n) \leq f(C'_1, \ldots, C'_n)$ for completion time vectors C, C' with $C_i \leq C'_i$ for all $i = 1, \ldots, n$. They are also called **regular**. On the other hand, monotone non-increasing functions are called **antiregular**.

The **maximum earliness** $\max_{i=1}^{n} E_i$ with $E_i := \max\{0, d_i - C_i\}$ is an example for an antiregular objective function, the **weighted earliness-tardiness** $\sum_{i=1}^{n} w_i^E E_i + \sum_{i=1}^{n} w_i^T T_i$ with earliness weights $w_i^E \geq 0$ and tardiness weights $w_i^T \geq 0$ is neither regular nor antiregular. Also the objective function $\sum w_i C_i$ with arbitrary weights $w_i \in \mathbb{R}$ is nonregular. If $w_i > 0$, activity i should be completed as early as possible, if $w_i < 0$, activity i should be completed as late as possible.

Another nonregular objective function related to the last one deals with the so-called **net present value**. Associated with each activity i is a so-called cash-flow $c_i^F \in \mathbb{R}$ which is supposed to occur at the completion time C_i of i. It may be positive (i.e. a payment is received) or negative (i.e. a payment has to be given). The objective is to maximize the so-called net present value (NPV) $\sum_{i=1}^{n} c_i^F e^{-\alpha C_i}$, where $\alpha \geq 0$ is a given discount rate.

Besides these time-oriented objective functions also resource-based objective functions may be considered. They occur for example in the area of resource

investment and resource levelling problems. In the **resource investment problem** (RIP) the resource capacities R_k are not given, but have to be determined as additional decision variables. Providing one unit of resource k costs $c_k \geq 0$. The objective is to find a schedule with $C_{\max} \leq T$ for a given deadline T where the resource investment costs $\sum_{k=1}^{r} c_k R_k$ are minimized.

In **resource levelling problems** (RLP) the variation of the resource usage over time is measured. Let $c_k \geq 0$ be a cost for resource k and denote by $r_k^S(t)$ the resource usage of resource k in period $t \in \{1, \ldots, T\}$ for a given schedule S, where $r_k^S(0) := 0$ is assumed. Besides the resouce capacity R_k a target value $Y_k \geq 0$ for resource k is given.

In so-called **deviation** problems the deviations (overloads) of the resource usages from a given resource profile are minimized. This can be done by minimizing

$$\begin{array}{ll} \text{the deviation} & \sum_{k=1}^{r} c_k \sum_{t=1}^{T} |r_k^S(t) - Y_k|, \\ \text{the overload} & \sum_{k=1}^{r} c_k \sum_{t=1}^{T} \max\{0, r_k^S(t) - Y_k\}, \text{ or} \\ \text{the squared deviation} & \sum_{k=1}^{r} c_k \sum_{t=1}^{T} (r_k^S(t) - Y_k)^2. \end{array}$$

The value Y_k may also be replaced by the average resource usage

$$\overline{r}_k := \sum_{i=1}^{n} r_{ik} p_i / T.$$

On the other hand, in so-called **variation** problems, the resource usages should not vary much over time. This can be achieved by minimizing one of the objective functions

$$\begin{array}{l} \sum_{k=1}^{r} c_k \sum_{t=1}^{T} |r_k^S(t) - r_k^S(t-1)|, \\ \sum_{k=1}^{r} c_k \sum_{t=1}^{T} \max\{0, r_k^S(t) - r_k^S(t-1)\}, \text{ or} \\ \sum_{k=1}^{r} c_k \sum_{t=1}^{T} (r_k^S(t) - r_k^S(t-1))^2. \end{array}$$

Finally, we note that an RCPSP with maximum lateness objective function can be reduced to an RCPSP with makespan objective function and arbitrary time-lags. This follows from the fact that $L_{\max} = \max_{i=1}^{n}\{C_i - d_i\} = \max_{i=1}^{n}\{S_i + p_i - d_i\} \leq L$ for a threshold value L if and only if $S_i + p_i - d_i \leq L$ for all activities $i = 1, \ldots, n$. By setting time-lags $d_{i,n+1} := p_i - d_i$ for the dummy terminating activity $n+1$, the relations $S_i + p_i - d_i = S_i + d_{i,n+1} \leq S_{n+1}$ must be satisfied. Thus, by minimizing the makespan (which is equivalent to minimizing S_{n+1}) we minimize the maximum lateness $L_{\max}$.

1.1.2 Applications of the RCPSP

In the following four applications of the RCPSP are presented.

Application 1.1: A cutting-stock problem

Materials such as paper, textiles, cellophane, etc. are manufactured in standard rolls of a large width W which is the same for all rolls. These rolls have to be cut into smaller rolls $i = 1, \ldots, n$ with widths w_i such that the number of sliced rolls is minimized. In Figure 1.7 a solution of a cutting-stock problem with $n = 15$ smaller rolls is illustrated. In this solution 7 rolls have to be cut.

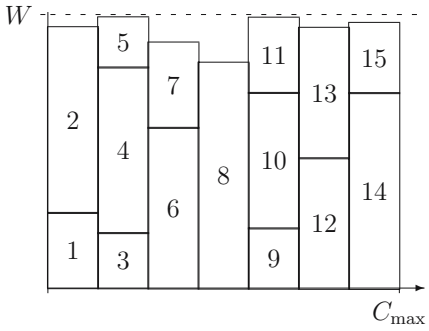


Figure 1.7: Solution of a cutting-stock problem

This problem can be formulated as an RCPSP with only one renewable resource where $R_1 = W$ units of the resource are available. The activities $i = 1, \ldots, n$ correspond to the rolls to be cut. Activity i has processing time $p_i = 1$ and uses $r_{i1} = w_i$ units of this resource. The makespan corresponds to the number of standard rolls to be cut, i.e. a schedule with a minimal makespan corresponds to a solution with a minimal number of sliced standard rolls. □

Application 1.2: High-school timetabling

In a basic high-school timetabling problem we are given m classes $c_1, \ldots, c_m$, h teachers $A_1, \ldots, A_h$ and T teaching periods $t = 1, \ldots, T$. Furthermore, we have lectures $i = 1, \ldots, n$. Associated with each lecture is a unique teacher and a unique class. A teacher A_j may be available only in certain teaching periods. The corresponding timetabling problem is to assign the lectures to the teaching periods such that

- each class has at most one lecture in any time period,
- each teacher has at most one lecture in any time period,
- each teacher has only to teach in time periods where he is available.

This problem may be formulated as an RCPSP with time-dependent resource profiles and n activities, where each activity corresponds to a lecture given by some teacher for some class.

Furthermore, we have $r := m + h$ resources $k = 1, \ldots, m, m+1, \ldots, m+h$. The first m resources $1, \ldots, m$ correspond to the classes, the last h resources $m+1, \ldots, m+h$ to the teachers $A_1, \ldots, A_h$. We have $R_k = 1$ for $k = 1, \ldots, m$. The availability of the resources $k = m+1, \ldots, m+h$ is time-dependent:

$$R_{m+j}(t) = \begin{cases} 1, & \text{if teacher } A_j \text{ is available in period } t \\ 0, & \text{otherwise.} \end{cases}$$

If activity i is a lecture for class c_l given by teacher A_j, then its resource requirement for resource k is

$$r_{ik} = \begin{cases} 1, & \text{if } k = l \text{ or } k = m+j \\ 0, & \text{otherwise.} \end{cases}$$

In a basic version of the problem one has to find a feasible schedule with $C_{\max} \leq T$. In practice, many additional constraints may have to be satisfied, e.g.

- for each class or teacher the number of teaching periods per day is bounded from above and below,
- certain lectures must be taught in special rooms,
- some pairs of lectures have to be scheduled simultaneously,
- the lectures of a class given by the same teacher should be spread uniformly over the week,
- classes or teachers should not have much idle periods on a day, etc.

□

Application 1.3: An audit-staff scheduling problem

A set of jobs $J_1, \ldots, J_g$ are to be processed by auditors $A_1, \ldots, A_m$. Job J_l consists of n_l tasks ($l = 1, \ldots, g$). There may be precedence constraints $i_1 \to i_2$ between tasks i_1, i_2 of the same job. Associated with each job J_l is a release time r_l, a due date d_l and a weight w_l.

Each task must be processed by exactly one auditor. If task i is processed by auditor A_k, then its processing time is p_{ik}. Auditor A_k is available during disjoint time intervals $[s_k^\nu, l_k^\nu]$ ($\nu = 1, \ldots, m_k$) with $l_k^\nu \leq s_k^{\nu+1}$ for $\nu = 1, \ldots, m_k - 1$. Furthermore, the total working time of A_k is bounded from below by H_k^- and from above by H_k^+ with $H_k^- \leq H_k^+$ ($k = 1, \ldots, m$).

We have to find an assignment $\alpha(i)$ for each task $i = 1, \ldots, n := \sum_{l=1}^{g} n_l$ to an auditor $A_{\alpha(i)}$ and to schedule the assigned tasks such that

- each task is processed without preemption in a time window of the assigned auditor,
- the total workload of A_k is bounded by H_k^- and H_k^+ for $k = 1, \ldots, m$,
- the precedence constraints are satisfied,
- all tasks of J_l do not start before time r_l, and
- the total weighted tardiness $\sum_{l=1}^{g} w_l T_l$ is minimized.

Other features may be added to the model:

- Restricted preemption in the sense that if a task cannot be finished within one working interval of an auditor it must be continued at the beginning of the next interval of the same auditor.
- Setup times and setup costs if an auditor moves from one job to another.
- Costs c_{ik} for assigning task i to auditor A_k. By setting $c_{ik} := \infty$ we may model that task i cannot be assigned to auditor A_k. In this situation the term $\sum_{i=1}^{n} c_{i\alpha(i)}$ may be added to the objective function.

Audit-staff scheduling problems may be modeled as multi-mode resource-constrained project scheduling problems with time-dependent resource profiles. For each auditor a doubly-constrained resource is introduced which on the one hand as a renewable resource is constrained by the availability profile of the auditor and, on the other hand as a non-renewable resource, by the working time bounds H_k^-, H_k^+.

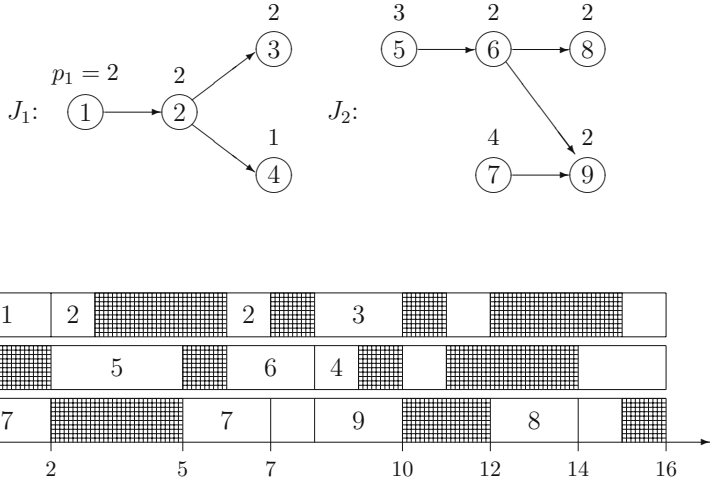


Figure 1.8: An audit staff schedule

In Figure 1.8 an example for an audit-staff scheduling problem with $g = 2$ jobs and $m = 3$ auditors is considered. We assume $r_1 = r_2 = 0$, $d_1 = 11, d_2 = 12$ and $w_1 = 1, w_2 = 2$. Auditor A_1 is not available in the time intervals $[3, 6]$, $[7, 8]$, $[10, 11]$, $[12, 15]$, A_2 is not available in the intervals $[0, 2]$, $[5, 6]$, $[9, 10]$, $[11, 14]$, and A_3 is not available in $[2, 5]$,$[10, 12]$,$[15, 16]$. Additionally, we assume that restricted preemptions are allowed.

In Figure 1.8 also a feasible schedule is shown, where the shaded areas indicate the time periods during which the auditors are not available. The objective function value of this schedule is $w_1T_1 + w_2T_2 = 1 \cdot 0 + 2 \cdot (14 - 12) = 4$. □

Application 1.4: Sports league scheduling

In a basic sports league scheduling problem we are given a league consisting of $2n$ different teams $i = 1, \ldots, 2n$. These teams play a double round robin tournament during a season, i.e. each team plays against each other team twice. The season is partitioned into two half series, each pairing has to occur exactly once in each half. If the pairing (i, j) takes place at the stadium of team i in the first half, it has to take place in the stadium of j in the second half. Furthermore, each half series consists of $2n - 1$ rounds (days for games) and every team has to play exactly one game in each round.

The second half series is usually not scheduled independently from the first series. Often the second series is planned complementarily to the first, i.e. the pairings of round t in the second half are the same as in round t of the first half (with exchanged home rights). We assume that all requirements for the second half can be transformed into requirements for the first half series (e.g. if in a round of the second half the stadium for a team is unavailable, the team has to play at home in the corresponding round of the first half). Thus, it is sufficient to find a schedule for the first half series consisting of the rounds $t = 1, \ldots, T := 2n - 1$ taking into account constraints for the second half series.

Such a problem may be formulated as a multi-mode RCPSP with time-dependent resource profiles as follows. Since we only plan one half series, we have $n(2n-1)$ games which are represented by all pairs (i, j) with $i, j \in \{1, \ldots, 2n\}$ and $i < j$. These pairs correspond to activities, which may be processed in two different modes (in mode H scheduling the game at the home stadium of team i, or in mode A scheduling the game at the home stadium of team j). All activities have unit processing times in each mode. We introduce $2n$ team resources for $i = 1, \ldots, 2n$ with constant capacity 1, ensuring that each team plays at most one game in each round. An activity corresponding to the game (i, j) needs one unit of the two team resources belonging to teams i and j.

In the following we describe additional constraints which can be formulated with the concepts of resources:

- **Stadium availabilities**: Due to some other events (e.g. concerts or games of other sports disciplines) some stadiums may not be available in certain

rounds. We assume that each team i specifies for all rounds $t = 1, \ldots, \hat{T} := 2(2n-1)$ the values

$$S_{it} = \begin{cases} 1, & \text{if the stadium of } i \text{ is available in round } t \\ 0, & \text{otherwise.} \end{cases}$$

If $S_{it} = 0$ for some round t holds, team i cannot play a home game in round t, i.e. it has to play away.

To model such a situation we introduce $4n$ stadium resources `STADRES1(i)` (first half series) and `STADRES2(i)` (transformation of second half) for $i = 1, \ldots, 2n$ with variable 0-1 resource profiles

$$\texttt{STADRES1(i)}(t) = S_{it} \in \{0,1\} \text{ und } \texttt{STADRES2(i)}(t) = S_{i,t+2n-1} \in \{0,1\}$$

for $t = 1, \ldots, 2n-1$ ensuring that the stadium availabilities $S_{i\tau}$ ($\tau = 1, \ldots, \hat{T}$) are respected in the first and second half series.

An activity corresponding to the game (i,j) needs one unit of stadium resources `STADRES1(i)` and `STADRES2(j)` in mode H and one unit of `STADRES1(j)` and `STADRES2(i)` in mode A, respectively.

- **Regions**: Teams which are located close to each other, should not all play at home in a round simultaneously. We assume that ρ regions $\mathcal{R}_1, \ldots, \mathcal{R}_\rho$ are specified as subsets of teams which are located in the same region. In each round at most $\lceil \frac{|\mathcal{R}_r|}{2} \rceil$ home games may be scheduled in region $\mathcal{R}_r$. If, especially, two teams share the same stadium, a region containing these two teams may be used to model the situation that in no round the two teams can play at home simultaneously.

  To model regions we introduce 2ρ region resources `REGRES1(r)` (first half series) and `REGRES2(r)` (transformation of second half) for regions $r = 1, \ldots, \rho$ with constant capacity $\lceil \frac{|\mathcal{R}_r|}{2} \rceil$.

  An activity corresponding to the game (i,j) needs one unit of region resources `REGRES1(reg[i])` and `REGRES2(reg[j])` in mode H and one unit of region resources `REGRES1(reg[j])` and `REGRES2(reg[i])` in mode A, respectively, where `reg[i]` denotes the region of team i.

- **Games with forbidden rounds**: Certain games may not be scheduled in certain rounds (e.g. games between two teams in the same region if in that region already another event takes place in this round, top games not at the beginning of the season, games between league climbers not at the end). We assume that for all such games (i,j) a subset $\hat{\mathcal{T}}_{(i,j)} \subset \{1, \ldots, T\}$ of forbidden rounds is given.

  To model this case we introduce for each game (i,j) with forbidden rounds one resource `FORBRES(i,j)` with variable 0-1 resource profile

$$\texttt{FORBRES(i,j)}(t) = \begin{cases} 0, & \text{if } t \in \hat{\mathcal{T}}_{(i,j)} \\ 1, & \text{otherwise.} \end{cases}$$

Then an activity corresponding to a forbidden game (i, j) needs one unit of resource `FORBRES(i,j)` in both modes.

- **Attractive games**: In order to distribute interesting games over the whole season, in each round the number of attractive games may not exceed a given number. We assume that all attractive games (e.g. top games or local derbies) are specified in a set AG and that a limit $ag_{\max}$ for the number of attractive games per round is given. We introduce one resource `AGRES` with constant capacity $ag_{\max}$ and each activity corresponding to an attractive game (i, j) needs one unit of resource `AGRES` in both modes.

- **Distribution of home and away games**: In each half series each team should play approximately half of its games at home, the other half away. Thus, at most $\lceil \frac{2n-1}{2} \rceil$ home games (away games) may be scheduled for each team in the first half. For this purpose we introduce $2n$ non-renewable home resources `HOMERES(i)` and $2n$ non-renewable away resources `AWAYRES(i)` for teams $i = 1, \ldots, 2n$ with capacity $\lceil \frac{2n-1}{2} \rceil$.

  An activity corresponding to the game (i, j) needs one unit of home resource `HOMERES(i)` and one unit of away resource `AWAYRES(j)` in mode H and one unit of `HOMERES(j)` and `AWAYRES(i)` in mode A, respectively.

Components in the objective function may be:

- **Home/away preferences**: Besides the (hard) stadium unavailabilities S_{it}, teams may have preferences for home or away games in certain rounds. For example, during public festivals home games are preferred, while away games are preferred when another attractive event in the same region is already scheduled on a certain date (e.g. also games of other leagues).

- **Opponent strengths**: In order to distribute games against stronger and weaker opponents evenly over the season, for each team the opponents in two consecutive rounds should have different strengths.

- **Breaks**: Often it is not desired that teams play two home or two away games in two consecutive rounds.

Violations of all these constraints may be penalized in the objective function (e.g. by summing up penalties for all violations). □

Other important special cases of resource-constrained project scheduling problems are machine scheduling problems (where the resources correspond to machines). They will be discussed in more detail in the next section.

1.2 Machine Scheduling

In this section we will introduce important classes of machine scheduling problems.

1.2.1 Single-machine scheduling

In the simplest machine scheduling model we are given n jobs $j = 1, \ldots, n$ with processing times p_j which have to be processed on a single machine. Additionally, precedence constraints may be given. Such a problem can be modeled by an RCPSP with one renewable disjunctive resource $r = 1$, capacity $R_1 = 1$ and resource requirements $r_{j1} = 1$ for all jobs $j = 1, \ldots, n$.

Example 1.5: Consider an instance with $n = 5$ jobs, processing times $p_1 = 3$, $p_2 = 2$, $p_3 = 4$, $p_4 = 2$, $p_5 = 3$ and precedence constraints $1 \rightarrow 3,\ 2 \rightarrow 4,\ 4 \rightarrow 5$. A feasible schedule for this instance with makespan $C_{\max} = 14$ is shown in Figure 1.9

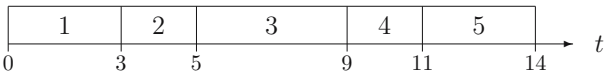


Figure 1.9: Single machine schedule

□

1.2.2 Parallel machine scheduling

Instead of a single machine we may have m machines $M_1, \ldots, M_m$ on which the jobs have to be processed. If the machines are **identical**, the processing time p_j of job j does not depend on the machine on which j is processed. This problem corresponds to an RCPSP with one cumulative resource where $R_1 = m$ and $r_{j1} = 1$ for $j = 1, \ldots, n$.

Example 1.6: Consider an instance with $n = 8$ jobs, $m = 3$ machines, processing times $p_1 = 1$, $p_2 = 3$, $p_3 = 4$, $p_4 = 2$, $p_5 = 2$, $p_6 = 3$, $p_7 = 1$, $p_8 = 5$ and precedence constraints as shown in the left part of Figure 1.10. A feasible instance with makespan $C_{\max} = 9$ is shown in the right part of the figure.

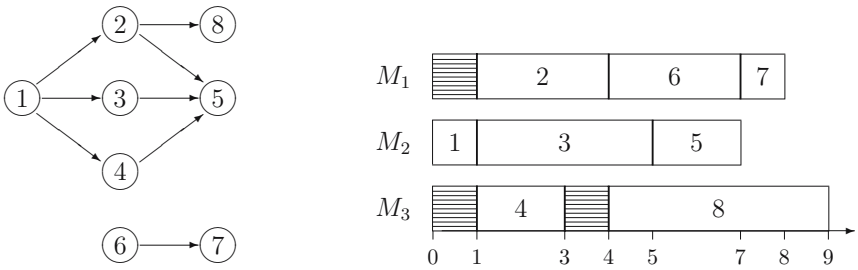


Figure 1.10: Schedule for identical parallel machines

□

Contrary to identical machines, for so-called **unrelated** machines the processing time p_{jk} of job j depends on the machine M_k ($k = 1, \ldots, m$) on which j is processed. The machines are called **uniform** if $p_{jk} = p_j/s_k$ where s_k may be interpreted as the speed of machine M_k. Problems with unrelated machines can be modeled by a multi-mode RCPSP with m renewable resources and $R_k = 1$ for $k = 1, \ldots, m$. Each job j has m modes corresponding to the machines on which j may be processed. If job j is processed in mode k, then p_{jk} is its processing time and j uses only one unit of resource k (machine M_k).

A further generalization are scheduling problems with **multi-purpose machines** (flexible machines). In this situation we associate with each job j a subset of machines $\mu_j \subseteq \{M_1, \ldots, M_m\}$ indicating that j can be executed by any machine of this set. If job j is processed on machine M_k, then its processing time is equal to p_{jk} (or simply to p_j if the processing time does not depend on the assigned machine). This problem can be formulated as above as a multi-mode RCPSP with m renewable resources where each job j has only $|\mu_j|$ modes corresponding to the machines on which j may be processed.

1.2.3 Shop scheduling

In so-called shop scheduling problems the jobs consist of several operations which have to be processed on different machines. In **general-shop scheduling problems** we have jobs $j = 1, \ldots, n$ and m machines $M_1, \ldots, M_m$. Job j consists of n_j operations $O_{1j}, \ldots, O_{n_j,j}$. Two operations of the same job cannot be processed at the same time and a machine can process at most one operation at any time. Operation O_{ij} must be processed for p_{ij} time units on a dedicated machine $\mu_{ij} \in \{M_1, \ldots, M_m\}$. Furthermore, precedence constraints may be given between arbitrary operations.

Such a general-shop scheduling problem can be modeled by an RCPSP with $r = m + n$ renewable resources and $R_k = 1$ for $k = 1, \ldots, m + n$. While the resources $k = 1, \ldots, m$ correspond to the machines, the resources $m + j$ ($j = 1, \ldots, n$) are needed to model the fact that different operations of job j cannot be processed at the same time. Furthermore, we have $\sum_{j=1}^{n} n_j$ activities O_{ij}, where operation O_{ij} needs one unit of "machine resource" μ_{ij} and one unit of "job resource" $m + j$.

Important special cases of the general-shop scheduling problem are job-shop, flow-shop, and open-shop problems, which will be discussed next.

Job-shop problems

A **job-shop problem** is a general-shop scheduling problem with chain precedences of the form

$$O_{1j} \rightarrow O_{2j} \rightarrow \ldots \rightarrow O_{n_j,j}$$

for $j = 1, \ldots, n$ (i.e. there are no precedences between operations of different jobs and the precedences between operations of the same job build a chain). Note that for a job-shop problem no "job resource" is needed, since all operations of the same job are linked by a precedence relation (and thus cannot be processed simultaneously).

Flow-shop problems

A **flow-shop problem** is a special job-shop problem with $n_j = m$ operations for $j = 1, \ldots, n$ and $\mu_{ij} = M_i$ for $i = 1, \ldots, m$, $j = 1, \ldots, n$, i.e. operation O_{ij} must be processed on M_i. In a so-called **permutation flow-shop problem** the jobs have to be processed in the same order on all machines.

Example 1.7: In Figure 1.11 a feasible schedule for a permutation flow-shop problem with $n = 4$ jobs and $m = 3$ machines is shown.

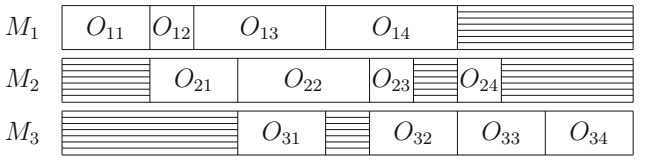


Figure 1.11: Permutation schedule for a flow-shop problem

□

Open-shop problems

An **open-shop problem** is like a flow-shop problem but without any precedence relations between the operations. Thus, it also has to be decided in which order the operations of a job are processed.

Example 1.8: In Figure 1.12 a feasible schedule for an open-shop problem with $n = 2$ jobs and $m = 3$ machines is shown. In this schedule the operations of job 1 are processed in the order $O_{11} \to O_{31} \to O_{21}$, the operations of job 2 are processed in the order $O_{32} \to O_{22} \to O_{12}$.

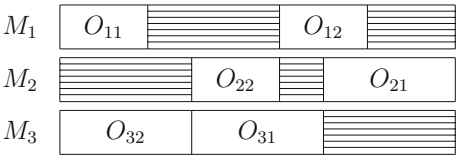


Figure 1.12: Feasible schedule for an open-shop problem

□

1.2.4 Multi-processor task scheduling

In a **multi-processor task scheduling problem** we have n jobs $j = 1, \ldots, n$ and m machines $M_1, \ldots, M_m$. Associated with each job j is a processing time p_j and a subset of machines $\mu_j \subseteq \{M_1, \ldots, M_m\}$. During its processing job j occupies each of the machines in μ_j. Finally, precedence constraints may be given between certain jobs.

This problem can be formulated as an RCPSP with $r = m$ renewable resources and $R_k = 1$ for $k = 1, \ldots, r$. Furthermore,

$$r_{jk} = \begin{cases} 1, & \text{if } M_k \in \mu_j \\ 0, & \text{otherwise.} \end{cases}$$

Example 1.9: Consider the following instance with $n = 5$ jobs and $m = 3$ machines:

j	1	2	3	4	5
μ_j	$\{M_1, M_2\}$	$\{M_2, M_3\}$	$\{M_1, M_2\}$	$\{M_3\}$	$\{M_1, M_2, M_3\}$
p_j	1	2	2	3	1

In Figure 1.13 a feasible schedule with makespan $C_{\max} = 7$ for this instance is shown. It does not minimize the makespan since by processing job 1 together with job 4 we can get a schedule with $C_{\max} = 6$.

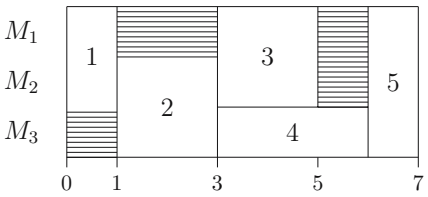


Figure 1.13: Feasible schedule for a multi-processor task problem

□

Finally, **multi-mode multi-processor task** scheduling problems are a combination of problems with multi-processor tasks and multi-purpose machines. This means that with each job a set of different machine subsets (processing alternatives) is associated and a job needs all machines from a subset simultaneously.

1.3 Reference Notes

In recent years a large number of papers has been published in the area of project scheduling. The most important results and corresponding references can be found in the books by Demeulemeester and Herroelen [42], Neumann et al. [110], the handbook [141] edited by Węglarz and the survey articles of Özdamar and Ulusoy [116], Herroelen et al. [72], Brucker et al. [17] and Kolisch and Padman [88].

The concept of partially renewable resources was introduced by Böttcher et al. [15] and Schirmer and Drexl [129]. Connections between packing problems and project scheduling models are studied in Hartmann [65]. A survey on timetabling problems can be found in Schaerf [128], connections between timetabling and resource-constrained project scheduling problems are discussed in Brucker and Knust [24]. Solution algorithms for audit-staff scheduling problems are developed in Dodin et al. [46] and Brucker and Schumacher [29]. Drexl and Knust [51] present graph- and resource-based models for sports league scheduling problems.

For machine scheduling problems cf. the books of Błażewicz et al. [13], Brucker [16], Pinedo [122], [123] and the handbook [98] edited by Leung.

Chapter 2

Algorithms and Complexity

In this chapter we discuss some basic concepts and solution methods for combinatorial optimization problems. In a **combinatorial (discrete) optimization problem** we are given a finite set $\mathcal{S}$ and an objective function $c : \mathcal{S} \to \mathbb{R}$. While for minimization problems we have to find a solution $s^* \in \mathcal{S}$ with $c(s^*) \leq c(s)$ for all $s \in \mathcal{S}$, for maximization problems we have to find a solution $s^* \in \mathcal{S}$ with $c(s^*) \geq c(s)$ for all s.

In Section 2.1 a short introduction into complexity theory is given which is useful to decide whether an optimization problem is easy or hard to solve. Examples for easy problems are shortest path problems which are discussed in Section 2.2. An important class of problems which are easy to solve are optimization problems which can be formulated as linear programs. Section 2.3 contains an introduction to linear programming, Section 2.4 is devoted to network flow problems, which may be considered as special linear programs.

For hard optimization problems exact algorithms (which always determine an optimal solution) and approximation algorithms (which only provide approximate solutions) are distinguished. Exact algorithms like branch-and-bound and dynamic programming algorithms are described in Sections 2.5 and 2.6, respectively. In Section 2.7 local search and genetic algorithms are presented as examples for approximation algorithms.

2.1 Easy and Hard Problems

When we consider scheduling problems (or more generally combinatorial optimization problems) an important issue is the complexity of a problem. For a new problem we often first try to develop an algorithm which solves the problem in an efficient way. If we cannot find such an algorithm, it may be helpful to prove that the problem is NP-hard, which implies that with high probability no efficient algorithm exists. In this section we review the most important aspects of complexity theory which allow us to decide whether a problem is "easy" or "hard".

2.1.1 Polynomially solvable problems

The efficiency of an algorithm may be measured by its running time, i.e. the number of steps it needs for a certain input. Since it is reasonable that this number is larger for a larger input, an efficiency function should depend on the input size. The size of an input for a computer program is usually defined by the length of a **binary encoding** of the input. For example, in a binary encoding an integer number a is represented as binary number using $\log_2 a$ bits, an array with m numbers needs $m \log_2 a$ bits when a is the largest number in the array. On the other hand, in a so-called **unary encoding** a number a is represented by a bits.

Since in most cases it is very difficult to determine the average running time of an algorithm for an input with size n, often the worst-case running time of an algorithm is studied. For this purpose, the function $T(n)$ may be defined as an upper bound on the running time of the algorithm on any input with size n. Since often it is also difficult to give a precise description of $T(n)$, we only consider the growth rate of $T(n)$ which is determined by its asymptotic order. We say that a function $T(n)$ is $O(g(n))$ if constants $c > 0, n_0 \in \mathbb{N}$ exist such that $T(n) \leq cg(n)$ for all $n \geq n_0$. For example, the function $T_1(n) = 37n^3 + 4n^2 + n$ is $O(n^3)$, the function $T_2(n) = 2^n + n^{100} + 4$ is $O(2^n)$.

If the running time of an algorithm is bounded by a polynomial function in n, i.e. if it is $O(n^k)$ for some constant $k \in \mathbb{N}$, the algorithm is called a **polynomial-time** algorithm. If the running time is bounded by a polynomial function in the input size of a unary encoding, an algorithm is called **pseudo-polynomial**. For example, an algorithm with running time $O(n^2 a)$ is pseudo-polynomial if the input size of a binary encoding is $O(n \log a)$.

Since exponential functions grow much faster than polynomial functions, exponential-time algorithms cannot be used for larger problems. If we compare two algorithms with running times $O(n^3)$ and $O(2^n)$ under the assumption that a computer can perform 10^9 steps per second, we get the following result: While for $n = 1000$ the first algorithm is still finished after one second, the second algorithm needs already 18 minutes for $n = 40$, 36 years for $n = 60$ and even 374 centuries for $n = 70$.

For this reason a problem is classified as "easy" (tractable) if it can be solved by a polynomial-time algorithm. In order to classify a problem as "hard" (intractable), the notion of NP-hardness was developed which will be discussed in the next subsection.

2.1.2 NP-hard problems

In order to define NP-hardness at first some preliminary definitions have to be given. A problem is called a **decision problem** if its answer is only "yes" or "no". A famous example for a decision problem is the so-called **partition**

problem: Given n integers $a_1, \ldots, a_n$ and the value $b := \frac{1}{2}\sum_{i=1}^{n} a_i$, is there a subset $I \subset \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i = b$?

On the other hand, most scheduling problems belong to the class of **optimization problems**, where the answer is a feasibe solution which minimizes (or maximizes) a given objective function c. For each scheduling problem a corresponding decision problem may be defined by asking whether a feasible schedule S with $c(S) \leq y$ for a given threshold value y exists.

The class of all polynomially solvable decision problems is denoted by $\mathcal{P}$. Another important complexity class is the set $\mathcal{NP}$ which contains all decision problems that are nondeterministically polynomially solvable. An equivalent definition of this class is based on so-called certificates which can be verified in polynomial time.

A **certificate** for a "yes"-instance of a decision problem is a piece of information which proves that a "yes"-answer for this instance exists. For example, for the partition problem a set I with $\sum_{i \in I} a_i = b$ is a certificate for a "yes"-instance. For a decision problem corresponding to a scheduling problem a feasible schedule S with $c(S) \leq y$ provides a certificate for a "yes"-instance. If such a certificate is short and can be checked in polynomial time, the corresponding decision problem belongs to the class $\mathcal{NP}$. More precisely, the set $\mathcal{NP}$ contains all decision problems where each "yes"-instance I has a certificate which

- has a length polynomially bounded in the input size of I, and
- can be verified by a polynomial-time algorithm.

Note that there is an important asymmetry between "yes"- and "no"-instances. While a "yes"-answer can easily be certified by a single feasible schedule S with $c(S) \leq y$, a "no"-answer cannot be certified by a short piece of information.

Example 2.1: Consider the decision version of the RCPSP. We are given resources $k = 1, \ldots, r$ with capacities R_k, activities $i = 1, \ldots, n$ with processing times p_i, resource requirements r_{ik} and a set A of precedence relations $i \rightarrow j \in A$. Thus, the input length of a binary encoding of an RCPSP instance is bounded by $O(nr \log z + |A|)$, where z denotes the largest number among the p_i, r_{ik}- and R_k-values. In the corresponding decision problem we ask for a feasible schedule with $C_{\max} \leq y$ for a given threshold value y.

A certificate is given by the n completion times $C_1, \ldots, C_n$, which is obviously polynomially bounded in the input size. In order to verify the certificate we have to check that

- the resource capacities R_k are respected in all intervals $[t, t']$, where t, t' are two successive time points in the ordered list of all starting times $S_i = C_i - p_i$ and all completion times C_i of the activities,

- $C_i + p_i \leq C_j$ for all activities i, j with $i \rightarrow j \in A$, and
- $\max\limits_{i=1}^{n} \{C_i\} \leq y$.

Since this can be done in $O(nr + |A|)$ time, the certificate can be checked in polynomial time and the decision version of the RCPSP belongs to the class $\mathcal{NP}$. □

In a similar way it can be shown that other decision problems corresponding to scheduling problems belong to the class $\mathcal{NP}$. Furthermore, it is easy to see that every polynomially solvable decision problem belongs to $\mathcal{NP}$ (the output of the polynomial-time algorithm may be used as a certificate which can also be verified by the algorithm in polynomial time). Thus, $\mathcal{P} \subseteq \mathcal{NP}$. One of the most challenging open problems in theoretical computer science today is the question whether $\mathcal{P} = \mathcal{NP}$ holds. Although it is generally conjectured that $\mathcal{P} \neq \mathcal{NP}$ holds, up to now nobody was able to prove this conjecture.

The central issue for the definition of NP-hardness is the notion of a polynomial **reduction**. A decision problems P is said to be **polynomially reducible** to another decision problem Q (denoted by $P \propto Q$) if a polynomial-time computable function g exists which transforms every instance I for P into an instance $I' = g(I)$ for Q such that I is a "yes"-instance for P if and only if I' is a "yes"-instance for Q.

A polynomial reduction $P \propto Q$ between two decision problems P, Q implies that Q is at least as difficult as P. This means that if Q is polynomially solvable, then also P is polynomially solvable. Equivalently, if P is not polynomially solvable, then also Q cannot be solved in polynomial time. Furthermore, reducibility is transitive, i.e. if $P \propto Q$ and $Q \propto R$ for three decision problems P, Q, R, then also $P \propto R$.

A decision problem P is called **NP-complete** if

(i) P belongs to the class $\mathcal{NP}$, and

(ii) every other decision problem $Q \in \mathcal{NP}$ is polynomially reducible to P.

A problem P is called **NP-hard** if only (ii) holds. Especially, an optimization problem is NP-hard if the corresponding decision problem is NP-complete.

If any single NP-complete problem could be solved in polynomial time, then all problems in $\mathcal{NP}$ would also be polynomially solvable, i.e. we would have $\mathcal{P} = \mathcal{NP}$. Since nobody has found a polynomial-time algorithm for any NP-complete problem yet, with high probability no polynomial-time algorithm exists for these problems.

An extension of NP-completeness is the concept of strongly NP-complete problems. If a problem P is **strongly NP-complete**, then it is unlikely that it can be solved by a pseudo-polynomial algorithm, since this would imply $\mathcal{P} = \mathcal{NP}$.

In 1971 Steve Cook was the first to prove that NP-complete problems exist. He showed that the so-called **satisfiability problem** (in which it has to be decided whether a given boolean formula can be satisfied or not) is NP-complete. This result together with the transitivity of reducibility implies that for an NP-completeness proof of a decision problem $P \in \mathcal{NP}$ it is sufficient to show that

(ii') **one** known NP-complete problem $Q \in \mathcal{NP}$ is polynomially reducible to P

(since then we have $R \propto P$ for **all** problems $R \in \mathcal{NP}$ due to $R \propto Q \propto P$).

Usually, it is easy to show that P belongs to the class $\mathcal{NP}$. Thus, the main task in proving NP-completeness of a problem P is to find a known NP-complete problem Q which can be reduced to P. In this way the partition problem and other fundamental combinatorial problems have been shown to be NP-complete by reductions from the satisfiability problem.

In the following example we show that the decision version of the RCPSP is NP-complete.

Example 2.2: Consider again the decision version of the RCPSP. In Example 2.1 we have already shown that this problem belongs to the class $\mathcal{NP}$. Now we will show that it is already NP-complete to decide whether a feasible schedule with $C_{\max} \leq 2$ exists. For this purpose we reduce the partition problem to the RCPSP.

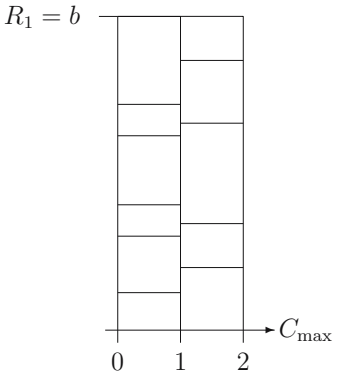


Figure 2.1: A feasible schedule with $C_{\max} \leq 2$

Given an instance of the partition problem with integers $a_1, \ldots, a_n$ and the value $b := \frac{1}{2}\sum_{i=1}^{n} a_i$, we define an instance of the RCPSP with a single resource with capacity $R_1 := b$. Additionally, we have n activities with unit processing times $p_i = 1$ and resource requirements $r_{i1} := a_i$. Obviously, this transformation can be done in polynomial time. We will show that a solution for the partition problem exists if and only if a feasible schedule for the RCPSP instance with $C_{\max} \leq 2$ exists.

If the partition problem has a solution, a set $I \subset \{1, \ldots, n\}$ with $\sum_{i \in I} a_i = \sum_{i \in I} r_{i1} = b = R_1$ exists. Thus, if we schedule all activities from the set I in the interval $[0, 1]$ and all remaining activities in the interval $[1, 2]$, we get a feasible schedule with $C_{\max} = 2$ (cf. Figure 2.1).

Conversely, if a feasible schedule with $C_{\max} \leq 2$ exists, the complete area $[0, 2] \times [0, b]$ in the Gantt chart has to be filled with activities (cf. Figure 2.1). Thus, the set I of activities scheduled in the interval $[0, 1]$ has to satisfy $\sum_{i \in I} r_{i1} = R_1 = b = \sum_{i \in I} a_i$, i.e. the set I defines a solution for the partition problem. □

The preceding proof implies that minimizing the makespan for an RCPSP is NP-hard. Thus, also all generalizations of the RCPSP or the RCPSP with more complicated objective functions are hard to solve. For the multi-mode RCPSP even the problem of determining a feasible mode assignment (i.e. modes for all activities which satisfy the non-renewable resource constraints) is hard. If generalized precedence relations with positive and negative time-lags are given, it is already NP-complete to decide whether a feasible schedule exists. It can be shown that only very special cases (with unit processing times) of the RCPSP are easy.

We conclude this subsection by giving some examples for the complexity status of important machine scheduling problems.

- **Single-machine problems**: The makespan minimization problem for a single machine is still polynomially solvable if precedence constraints and release dates are given. If additionally, due dates are introduced and the maximum lateness $L_{\max}$ has to be minimized, the problem becomes NP-hard. Minimizing the total tardiness $\sum T_i$ or the weighted number of late activities $\sum w_i U_i$ is already hard if no precedences and no release dates are given (but are polynomially solvable for unit processing times).

- **Parallel machine problems**: The makespan minimization problem is already NP-hard for two parallel identical machines, but is polynomially solvable for an arbitrary number of machines if preemption is allowed. On the other hand, minimizing the total flow time $\sum C_i$ is easy even for an arbitrary number of unrelated machines.

- **Shop problems**: The makespan minimization problem is polynomially solvable for open-shop and flow-shop environments with two machines and becomes NP-hard for three machines. Job-shop problems are already NP-hard for three machines and unit processing times.

- **Multi-processor task problems**: While for two machines the makespan minimization problem is polynomially solvable, it also becomes NP-hard for three machines.

2.2 Shortest Path Algorithms

Examples for easy (i.e. polynomially solvable) combinatorial optimization problems are shortest path problems, which will be discussed in this section.

Let $G = (V, A)$ be a directed graph with nodes (vertices) $V = \{1, \ldots, n\}$ and arc set $A \subseteq V \times V$. Furthermore, denote by $m := |A|$ the number of arcs in the graph. G is called a **network** if one or several numbers are associated with each arc.

Assume that we have a network where an **arc length** (or **arc cost**) $c_{ij} \in \mathbb{Z}$ is associated with each arc $(i, j) \in A$. An i_1-i_r-path is a sequence of nodes $(i_1, \ldots, i_r)$ with $(i_\nu, i_{\nu+1}) \in A$ for $\nu = 1, \ldots, r-1$. The value $l(P) := \sum_{\nu=1}^{r-1} c_{i_\nu i_{\nu+1}}$ is called the length of path $P = (i_1, \ldots, i_r)$. A **cycle** is an i_1-i_1-path. If its length is negative, it is called a **negative cycle**.

Let $s, t \in V$ be two specified nodes. In the **s-t-shortest path problem** we wish to find a path $P = (s = i_1, \ldots, i_r = t)$ from s to t with minimal length $l(P)$. In the **s-shortest path problem** we wish to find a shortest s-j-path for each node $j \neq s$. Finally, in an **all-pairs shortest path problem** we wish to calculate an i-j-shortest path for all pairs of nodes $i, j \in V$.

Similarly to shortest path problems, longest path problems can be formulated. To solve a longest path problem one may replace all arc lengths c_{ij} by $-c_{ij}$ and solve the shortest path problem with the transformed lengths.

In Section 2.2.1 we present an algorithm for the s-shortest path problem with non-negative arc lengths. The problem with arbitrary arc lengths is considered in Section 2.2.2. In both sections we assume that in G a path from s to every other node $j \in V$ in the network exists (otherwise we may add arcs (s, j) with large costs $c_{sj} := \infty$). In Section 2.2.2 we additionally assume that the network contains no negative cycle (otherwise the shortest path problem has no solution). How negative cycles can be detected is shown in Section 2.2.3. Finally, in Section 2.2.4 we solve the all-pairs shortest path problem.

2.2.1 Dijkstra's algorithm

In this subsection it is described how the s-shortest path problem with non-negative arc lengths can be solved by Dijkstra's algorithm. This algorithm associates a distance label $d(i)$ with each node $i \in V$. These labels are marked "temporary" or "permanent". Initially, we set $d(s) := 0$, $d(i) := \infty$ for all $i \neq s$ and mark all labels as temporary.

In each iteration of the algorithm a temporarily labeled node i with smallest $d(i)$-value is determined and permanently labeled. Furthermore, for all nodes j with $(i, j) \in A$ and $d(i)+c_{ij} < d(j)$ the labels $d(j)$ are updated by setting $d(j) := d(i)+c_{ij}$. In this case we also set $pred(j) := i$ to indicate that i is the predecessor

from j on a shortest s-j-path. After termination of the algorithm the *pred*-array can be used to reconstruct an s-j-shortest path in the reverse direction (i.e. from j to s by the sequence $j, pred[j], pred[pred[j]], pred[pred[pred[j]]]$, etc. until $pred[k] = s$ for some node $k \in V$ holds).

Figure 2.2 gives a formal description of Dijkstra's algorithm. While S denotes the set of permanently labeled nodes, $\overline{S} = V \setminus S$ is the set of temporarily labeled nodes.

```
Algorithm Dijkstra
 1.  S := ∅; S̄ := V;
 2.  d(s) := 0; pred(s) := 0;
 3.  FOR ALL i ∈ V \ {s} DO d(i) := ∞;
 4.  WHILE |S| < n DO
 5.      Let i ∈ S̄ be a node with d(i) = min{d(j) | j ∈ S̄};
 6.      S := S ∪ {i}; S̄ := S̄ \ {i};
 7.      FOR ALL j ∈ V with (i, j) ∈ A DO
 8.          IF d(j) > d(i) + c_ij THEN
 9.              d(j) := d(i) + c_ij; pred(j) := i
10.          ENDIF
11.  ENDWHILE
```

Figure 2.2: Dijkstra's algorithm

Example 2.3: Consider the s-shortest path problem with $s = 1$ for the graph $G = (V, A)$ with $n = 6$ nodes shown in Figure 2.3.

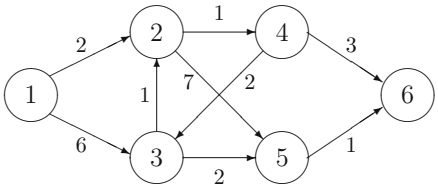


Figure 2.3: Example graph $G = (V, A)$ for Dijkstra's algorithm

In Figure 2.4(a)-(f) the iterations of Dijkstra's algorithm are illustrated. Nodes in S are indicated by an additional cycle, the smallest distance in each iteration is indicated by a square. After adding the node $s = 1$ to the set S in (a), the nodes 2 and 3 get finite distances $d(2) = 2$ and $d(3) = 6$. Since $d(2)$ is smaller, in (b) node 2 is added to S. Afterwards the nodes 4 and 5 get finite distances $d(4) = 3$ and $d(5) = 9$. After adding node 4 to S in (c), node 6 gets the label $d(6) = 6$. Additionally, the distance of node 3 reduces to $d(3) = d(4) + c_{43} = 3 + 2 = 5$. In the following iterations (d)-(f) successively the nodes 3, 6 and 5 are added to S.

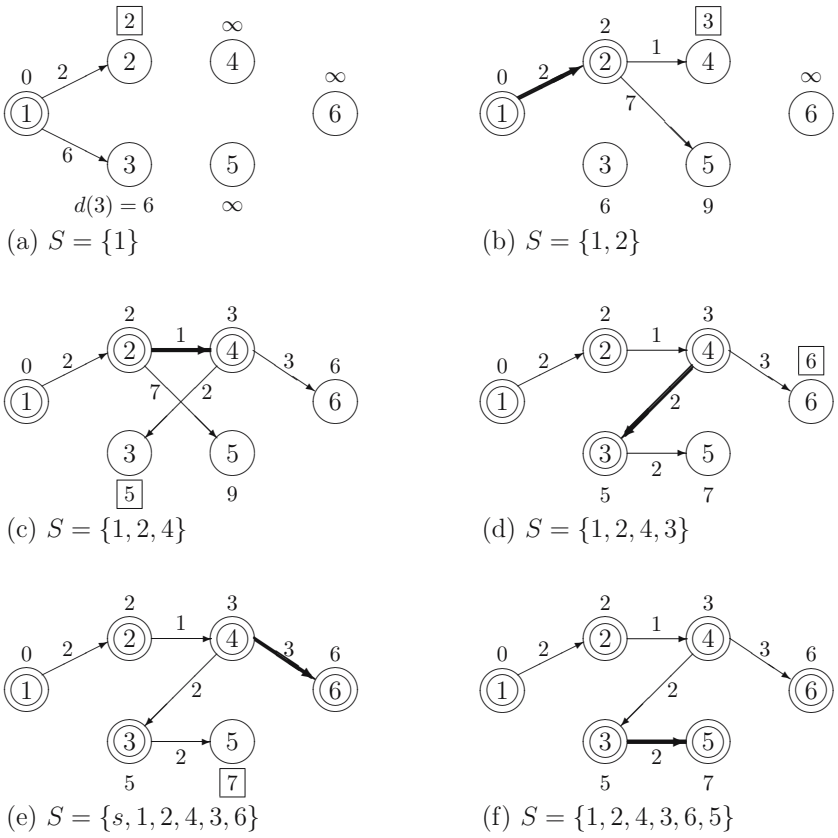


Figure 2.4: Iterations of Dijkstra's algorithm

Finally, the calculated shortest path lengths are $d(1) = 0$, $d(2) = 2$, $d(4) = 3$, $d(3) = 5$, $d(5) = 7$, $d(6) = 6$. □

Theorem 2.1 If all arc lengths are non-negative, Dijkstra's algorithm solves the s-shortest path problem in $O(n^2)$ time.

Proof: In each iteration one node i is added to S, i.e. we have n iterations of the while-loop. In each iteration the minimum in Step 5 can be calculated in $O(n)$ time. Furthermore, because each node i has at most n successors j with $(i, j) \in A$, the updates of the values $d(j)$ and $pred(j)$ in Steps 7-10 can be done in $O(n)$ time resulting in an overall time complexity of $O(n^2)$.

To prove correctness of the algorithm we show by induction on the number of iterations that the following properties hold:

(1) For each $i \in S$ the value $d(i)$ is the length of a shortest s-i-path.

(2) For each $j \in \overline{S}$ the value $d(j)$ is the length of a shortest s-j-path provided that each internal node of the path lies in S.

Clearly, conditions (1) and (2) hold for $S = \{s\}$, $d(s) = 0$ and the updated values $d(j) := c_{sj}$ after the first iteration.

Now assume that (1) and (2) hold until some iteration and consider the next iteration in which a node $i \in \overline{S}$ with smallest $d(i)$-value is added to S.

To verify (1), we have to prove that $d(i)$ is the length of a shortest s-i-path. By induction hypothesis we already know that $d(i)$ is the shortest path length among the paths in which all internal nodes belong to S. We have to show that the length of each path containing additional nodes from the set $\overline{S}$ is at least $d(i)$. For this purpose consider an arbitrary s-i-path P containing a node $k \in \overline{S}$ as internal node. Assume, that $k \in \overline{S}$ is the first such node in P (cf. Figure 2.5). Let P_1 be the subpath from s to k in P and P_2 be the subpath from k to i.

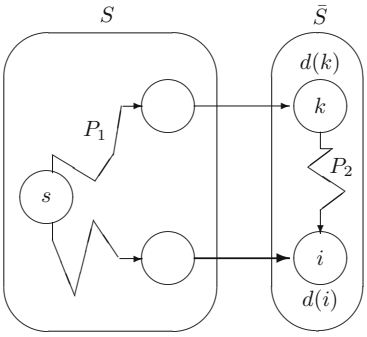


Figure 2.5: Sets S and $\bar{S}$ in Dijkstra's algorithm

By induction hypothesis the length of P_1 is at least $d(k)$. Furthermore, $d(k) \geq d(i) = \min\{d(j) \mid j \in \overline{S}\}$, which implies that the length of P is at least $d(i)$ because all arc lengths on the path P_2 from k to i are non-negative. Thus, $d(i)$ is the length of a shortest s-i-path, i.e. property (1) remains satisfied after we have added i to S.

It remains to prove that (2) holds for the sets $S \cup \{i\}$ and $\overline{S} \setminus \{i\}$ after replacing $d(j)$ by $d(i) + c_{ij}$ for all $(i, j) \in A$ with $d(j) > d(i) + c_{ij}$. For this purpose consider an arbitrary path $P = (s = i_1, i_2, \ldots, i_r, j)$ from s to $j \in \overline{S} \setminus \{i\}$ which is optimal among all paths from s to j containing only nodes from $S \cup \{i\}$ as internal nodes.

If P does not contain node i, then P is a shortest path with internal nodes in S and the distance $d(j)$ is correct by induction. Otherwise, if i is contained in P, we have $d(i) \geq d(i_\nu)$ for $\nu = 1, \ldots, r$ since otherwise i would have been added to S earlier.

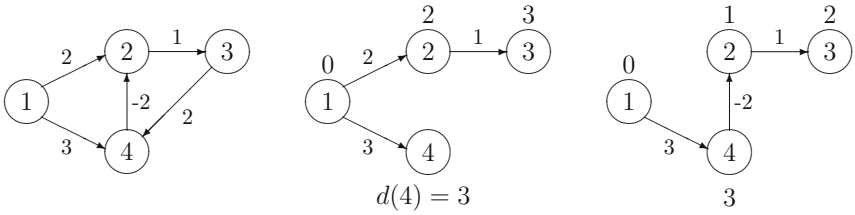
If $i = i_\mu$ for some $\mu < r$ in P holds, we must have $d(i) = d(i_\nu)$ for $\nu = \mu + 1, \ldots, r$. Since $i_{\mu+1}$ was added to S in a previous iteration, an s-$i_{\mu+1}$-path with length $d(i_{\mu+1})$ exists which does not contain node i. If we extend this path by $i_{\mu+2}, \ldots, i_r, j$, we obtain a shortest s-j-path in $S \setminus \{i\}$ and $d(j)$ is correct by induction.

If $i = i_r$ holds, then $d(j)$ is updated to $d(i)+c_{ij}$ in the iteration. Thus, afterwards $d(j)$ equals the length of the shortest s-j-path P in $S \cup \{i\}$. □

Note that the runnning time of Dijkstra's algorithm can be reduced from $O(n^2)$ to $O(m \log n)$ if more complicated data structures are used and the nodes $i \in \overline{S}$ are kept in heaps or priority queues.

The following example shows that Dijkstra's algorithm is not correct if the network contains negative arc lengths.

Example 2.4: Consider the s-shortest path problem with $s = 1$ for the network shown in Figure 2.6(a). While Dijkstra's algorithm calculates the distances $d(2) = 2$, $d(3) = 3$, $d(4) = 3$ as shown in (b), the correct distances are $d(2) = 1$, $d(3) = 2$, $d(4) = 3$ as shown in (c).



(a) Graph G (b) Dijkstra's algorithm (c) correct distances

Figure 2.6: Counterexample for Dijkstra's algorithm with negative arc lengths

□

2.2.2 Label-correcting algorithms

Contrary to Dijkstra's algorithm label-correcting algorithms can also be applied to networks with negative arc lengths. In this subsection we assume that the network does not contain a negative cycle, in the next subsection we show how negative cycles can be detected.

Also label-correcting algorithms maintain distance labels $d(i)$ for all nodes $i \in V$. They are initialized by setting $d(s) := 0$ and $d(i) := \infty$ for all $i \in V \setminus \{s\}$. In each iteration for an arc (i, j) with $d(j) > d(i) + c_{ij}$ the distance label $d(j)$ is replaced by $d(i)+c_{ij}$. Furthermore, an array $pred(j)$ is updated like in Dijkstra's algorithm. A generic label-correcting algorithm is shown in Figure 2.7.

In the following we will show that the label-correcting algorithm has calculated shortest path lengths after a finite number of iterations.

Property 2.1 Each value $d(j) \neq \infty$ calculated during the algorithm is equal to the length of an s-j-path in which no node is repeated.

Proof: We prove Property 2.1 by induction on the number of iterations. Clearly, $d(s) = 0$ has the desired property.

```
Algorithm Label-Correcting
1.  d(s) := 0;  pred(s) := 0;
2.  FOR ALL i ∈ V \ {s} DO d(i) := ∞;
3.  WHILE some arc (i,j) ∈ A satisfies d(j) > d(i) + c_ij DO
4.      d(j) := d(i) + c_ij;
5.      pred(j) := i
6.  ENDWHILE
```

Figure 2.7: A generic label-correcting algorithm

Consider an iteration k where $d(j) > d(i)+c_{ij}$ is replaced by $d^{new}(j) := d(i)+c_{ij}$. By induction, $d(i)$ is the length of an s-i-path $P = (s = i_1, i_2, \ldots, i_r = i)$ containing no cycle. Furthermore, $d^{new}(j)$ is the length of the path $\overline{P} = (s = i_1, i_2, \ldots, i_r = i, j)$. It remains to show that $\overline{P}$ has no cycle. Because P has no cycle by induction, a cycle in $\overline{P}$ must have the form $(j = i_q, i_{q+1}, \ldots, i_r = i, j)$ for some index $q > 1$. Because the network has no negative cycle, we have

$$\begin{aligned} d^{new}(j) &= d(i_r) + c_{i_r j} = d(i_{r-1}) + c_{i_{r-1},i_r} + c_{i_r j} = \ldots \\ &= d^{old}(j) + c_{j i_{q+1}} + \ldots + c_{i_r j} \geq d^{old}(j), \end{aligned}$$

where $d^{old}(j)$ denotes the label of j when $d(i_{q+1})$ was replaced by $d^{old}(j)+c_{j i_{q+1}}$. On the other hand, we must have $d^{new}(j) < d^{old}(j)$ because the $d(j)$-values never increase and in iteration k the $d(j)$-value decreased. This is a contradiction, i.e. $\overline{P}$ does not contain a cycle. □

Property 2.2 After termination of the label-correcting algorithm each $d(j)$-value is the length of a shortest s-j-path.

Proof: Let $(s = i_1, i_2, \ldots, i_r = j)$ be a shortest s-j-path. Then after termination of the algorithm we have

$$\begin{aligned} d(j) = d(i_r) &\leq d(i_{r-1}) + c_{i_{r-1},i_r} \\ d(i_{r-1}) &\leq d(i_{r-2}) + c_{i_{r-2},i_{r-1}} \\ &\vdots \\ d(i_2) &\leq d(i_1) + c_{i_1 i_2} = c_{i_1 i_2} \end{aligned}$$

because $d(i_1) = d(s) = 0$. If we add all inequalities, we get

$$d(j) \leq c_{i_{r-1},i_r} + \ldots + c_{i_1 i_2},$$

which must hold as equality because $d(j)$ is an s-j-path length due to Property 2.1. □

As a consequence of Property 2.1, for all calculated $d(j)$-values $-nC \leq d(j) \leq nC$ with $C := \max_{(i,j)\in A} |c_{ij}|$ holds. Therefore, each $d(j)$ is decreased at most $2nC$

times i.e. the algorithm stops after at most $O(n^2C)$ iterations of the while loop with $d(j) \leq d(i) + c_{ij}$ for all $(i,j) \in A$. Thus, the number of iterations of the algorithm is finite. However, the algorithm is not a polynomial-time algorithm since the input size of a shortest path problem is bounded by $O(n^2 \log C)$. In the following we discuss special implementations of the generic label-correcting algorithm leading to polynomial-time algorithms.

The generic label-correcting algorithm does not specify any method for selecting an arc $(i,j) \in A$ to check whether $d(i) + c_{ij} < d(j)$ holds. If these checks are performed in a specific way, smaller runnning times can be achieved.

For example, if the graph $G = (V, A)$ is acyclic, one can check the nodes i in a topological ordering, i.e. in an ordering $i_1, \ldots, i_n$ where $(i_k, i_l) \in A$ alway implies $k < l$. This provides an $O(m)$-algorithm since each arc in A is considered once.

Also Dijkstra's algorithm is a special implementation. It chooses a node i with minimal $d(i)$-value and checks all arcs $(i,j) \in A$ leading to an $O(n^2)$-algorithm for graphs with non-negative arc lengths.

For arbitrary networks an $O(nm)$-implementation is achieved in the following way: We first arrange the arcs in A in some specified order which may be chosen arbitrarily. Then we make passes through A. In each pass we scan the arcs $(i,j) \in A$ in the specified order and check the condition $d(j) > d(i) + c_{ij}$. For this implementation the following property holds, which implies that we need at most $n-1$ passes through A.

Property 2.3 At the end of the k-th pass, the algorithm has computed shortest path lengths for all nodes which are reachable from the source node s by a shortest path consisting of at most k arcs.

Proof: We prove this property by induction on the number of passes. The claim is clearly true for $k = 1$. Now suppose that the claim is true until the k-th pass where $k < n$. Let $P = (s = i_1, i_2 \ldots, i_{k+2} = j)$ be a shortest s-j-path with $k+1$ arcs and assume that among all s-j-paths with at most $k+1$ arcs no shortest path with less than $k+1$ arcs exists (otherwise the claim is true by induction). Then $i_1, \ldots, i_{k+1}$ must be a shortest s-i_{k+1}-path with at most k arcs and by induction hypothesis $d(i_{k+1})$ must be its length. After pass $k+1$ we must have $d(j) \leq d(i_{k+1}) + c_{i_{k+1}j}$ since the arc (i_{k+1}, j) is checked. But because P is a shortest path, we must have equality $d(j) = d(i_{k+1}) + c_{i_{k+1}j}$, i.e. the value $d(j)$ equals the length of P and is correct. □

Since a shortest s-j-path contains at most $n-1$ arcs, the algorithm terminates after at most $n-1$ iterations. Thus, its running time is $O(mn)$.

This special implementation of the label-correcting algorithm can be additionally improved by ordering the arcs in the arc list according to their tails, i.e. all arcs (i,j) with the same tail i appear consecutively in the arc list. Now suppose that the algorithm does not change the distance label $d(i)$ of some node i during one pass through the arc list. Then, during the next pass the condition $d(j) \leq d(i) + c_{ij}$ is satisfied for all nodes j, i.e. the algorithm does not have to check

```
Algorithm FIFO Label-Correcting
 1.  d(s) := 0; pred(s) := 0;
 2.  FOR ALL i ∈ V \ {s} DO d(i) := ∞;
 3.  List := {s}
 4.  WHILE List ≠ ∅ DO
 5.     Remove the first element i from List;
 6.     FOR ALL arcs (i,j) ∈ A DO
 7.          IF d(j) > d(i) + c_ij THEN
 8.             d(j) := d(i) + c_ij;
 9.             pred(j) := i
10.             IF j ∉ List THEN
11.                Add j as last element to List;
12.          ENDIF
13.  ENDWHILE
```

Figure 2.8: The FIFO Label-Correcting algorithm

these conditions. A possible way to implement this idea, is to store only those nodes in a list whose distance labels change in a pass and to organize the list in a first-in, first-out (FIFO) order. The corresponding so-called FIFO label-correcting algorithm is shown in Figure 2.8.

2.2.3 Detection of negative cycles

If the network contains a negative cycle, then the generic label-correcting algorithm will infinitely decrease distance labels of nodes in such a cycle. Since $-nC$ is a lower bound for the distance labels in the case without negative cycles, we can detect negative cycles by finding distance labels $d(j) < -nC$. Thus, negative cycles can be found in $O(n^2mC)$ time by the generic label-correcting algorithm.

If the network contains no negative cycle, then due to Property 2.3 the FIFO label-correcting algorithm stops after at most $n-1$ iterations with $List = \emptyset$. On the other hand, if the network contains a negative cycle, then labels are decreased in more than $n-1$ passes. Thus, we can detect negative cycles in $O(nm)$ time by checking whether $List \neq \emptyset$ after pass $n-1$ holds. Furthermore, in this case a negative cycle can be reconstructed using the *pred*-array.

2.2.4 Floyd-Warshall algorithm

In this section we consider the problem of finding a shortest i-j-path for each pair (i,j) of nodes $i,j \in V$. We assume that in G all nodes $i,j \in V$ are connected by an arc, otherwise we add arcs (i,j) with large costs $c_{ij} := \infty$.

The Floyd-Warshall algorithm, which solves this problem, is a label-correcting algorithm. It associates a distance label $d(i,j)$ with each pair $(i,j) \in V \times V$. These distance labels are initialized by $d(i,i) := 0$ for all $i \in V$ and $d(i,j) := c_{ij}$ for all $(i,j) \in V \times V$ with $i \neq j$. Then the labels $d(i,j)$ are systematically updated by checking the conditions

$$d(i,j) > d(i,k) + d(k,j) \text{ for some } k \in V. \tag{2.1}$$

If condition (2.1) is satisfied, then $d(i,j)$ is replaced by $d(i,k) + d(k,j)$. Details of this algorithm are given in Figure 2.9.

```
Algorithm Floyd-Warshall
 1.  FOR ALL i ∈ V DO d(i,i) := 0;
 2.  FOR ALL (i,j) ∈ V × V DO
 3.      d(i,j) := c_ij; pred(i,j) := i;
 4.  ENDFOR
 5.  FOR k := 1 TO n DO
 6.      FOR ALL (i,j) ∈ V × V DO
 7.          IF d(i,j) > d(i,k) + d(k,j) THEN
 8.              d(i,j) := d(i,k) + d(k,j);
 9.              pred(i,j) := pred(k,j)
10.          ENDIF
```

Figure 2.9: Floyd-Warshall Algorithm

In this algorithm the predecessor array $pred(i,j)$ is used to reconstruct a shortest path from i to j. After performing the algorithm $pred(i,j)$ is the predecessor of j on a shortest i-j-path. Thus, to reconstruct a shortest i-j-path we first calculate $j_1 = pred(i,j)$, then $j_2 = pred(i,j_1)$, etc. until we reach $i = pred(i,j_r)$. The resulting path $(i, j_r, j_{r-1}, \ldots, j_1, j)$ is a shortest i-j-path.

Clearly, the Floyd-Warshall algorithm runs in time $O(n^3)$. Its correctness for network with no negative cycles follows by

Property 2.4 Assume that the network contains no negative cycles. Then for $k = 0, \ldots, n$ after the k-th iteration of the loop 5 to 10 the value $d(i,j)$ is equal to the length of a shortest i-j-path subject to the condition that this path contains only nodes $1, 2, \ldots, k$ as additional nodes besides i and j.

Proof: We prove this property by induction on k. Property 2.4 is true for $k = 0$ because c_{ij} is the length of a shortest i-j-path containing no additional node (according to Step 3).

Now consider a shortest i-j-path P with nodes from the set $\{1, \ldots, k\}$ only. We may assume that P contains no cycle. Thus, if P contains node k, then it contains k only once and by induction hypothesis the length of P must be equal

to $d(i,k)+d(k,j)$. This follows from the fact that P is composed by a subpath P_1 from i to k and a subpath P_2 from k to j containing only nodes $1,\ldots,k-1$ as additional nodes. Both subpaths must be optimal and by induction their lengths are equal to $d(i,k)$ and $d(k,j)$, respectively. If P does not contain k, then P is optimal among all i-j-paths containing only nodes from $\{1,\ldots,k-1\}$ and by induction the length of P must be $d(i,j)$. □

The Floyd-Warshall algorithm may also be used to detect negative cycles. A negative cycle exists if and only if after terminating the algorithm $d(i,i)<0$ for at least one node i holds. To prove this, we first note that when running the algorithm a $d(i,j)$-value is never increased. Assume that the network contains a negative cycle. Then also a cycle $Z=(i=i_1,i_2,\ldots,i_r=i)$ exists in which each node appears only once. If Z is a loop (i,i), then we must have $d(i,i)\leq c_{ii}<0$. Otherwise, Z must contain an internal node $k=i_\nu$ and we have $d(i,i)\leq d(i,k)+d(k,i)\leq c_{i_1i_2}+\ldots+c_{i_{\nu-1}i_\nu}+c_{i_\nu i_{\nu+1}}+\ldots+c_{i_{r-1}i_r}<0$. If, on the other hand, $d(i,i)<0$ for some i, then $d(i,i)$ represents a negative cycle which may be reconstructed by the predecessor array.

2.3 Linear and Integer Programming

In this section we consider a more general class of polynomially solvable problems, namely combinatorial optimization problems which can be formulated as linear programs. After introducing basic concepts of linear programming the simplex algorithm is described, which is the most prominent procedure to solve linear programs. If in a linear program variables are restricted to be integer numbers, we obtain so-called integer-programs, which will be discussed in Section 2.3.4. Contrary to their continuous counterparts they are not easy to solve. Finally, in Section 2.3.5 so-called delayed column generation techniques are introduced, which are useful to solve linear programs with a huge number of variables.

2.3.1 Linear programs and the simplex algorithm

In this subsection basic notations for linear programs and the simplex algorithm are introduced. A **linear program in standard form** can be written as

$$\max \quad \sum_{j=1}^{n} c_j x_j \tag{2.2}$$

$$\text{subject to (s.t.)} \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad (i=1,\ldots,m) \tag{2.3}$$

$$x_j \geq 0 \qquad (j=1,\ldots,n) \tag{2.4}$$

where the a_{ij}, c_j and b_i are given real numbers.

Real numbers $x_1, \ldots, x_n$ satisfying the inequalities (2.3) and (2.4) are called a **feasible solution**. One has to find a feasible solution (if it exists) which minimizes the **objective function** $\sum_{j=1}^{n} c_j x_j$.

Example 2.5 shows that a feasible solution may not exist and even if a feasible solution exists, the linear program (2.2) to (2.4) may not have an optimal solution because the problem is **unbounded**. This is shown in Example 2.6.

Example 2.5: The problem

$$\begin{array}{lrcl} \max & 3x_1 - & x_2 & \\ \text{s.t.} & x_1 + & x_2 \leq & 2 \\ & -2x_1 - & 2x_2 \leq & -10 \\ & x_1, x_2 \geq & & 0 \end{array}$$

has no feasible solution because the second inequality is equivalent to $x_1 + x_2 \geq 5$ which contradicts the first inequality. □

Example 2.6: The problem

$$\begin{array}{lrcl} \max & x_1 - & x_2 & \\ \text{s.t.} & -2x_1 + & x_2 \leq & -1 \\ & -x_1 - & 2x_2 \leq & -2 \\ & x_1, x_2 \geq & & 0 \end{array}$$

has no optimal solution because for any real number $t \geq 2$ we have the feasible solution $x_1 = t$, $x_2 = 0$ with objective value $x_1 - x_2 = t$. This implies that for any number U we can find a feasible solution with $x_1 - x_2 \geq U$. We say that the problem is unbounded. □

In a linear program we may also have restrictions of the form

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \tag{2.5}$$

or

$$\sum_{i=1}^{n} a_{ij} x_j = b_i. \tag{2.6}$$

However, (2.5) is equivalent to

$$\sum_{j=1}^{n} -a_{ij} x_j \leq -b_i$$

and (2.6) may be replaced by the two inequalities

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \text{ and } \sum_{j=1}^{n} -a_{ij} x_j \leq -b_i.$$

In some applications we do not have the restrictions $x_j \geq 0$ for some variables x_j. This is indicated by $x_j \in \mathbb{R}$. In this case we can replace x_j by $x_j = x_j^+ - x_j^-$ with $x_j^+, x_j^- \geq 0$. Variables $x_j \in \mathbb{R}$ are called **unrestricted**.

Thus, in all cases the problem can be transformed into an equivalent linear program in standard form.

To solve a problem of the form

$$\min \sum_{j=1}^{n} c_j x_j \quad \text{s.t. (2.3) and (2.4)}$$

we solve the problem

$$\max \sum_{j=1}^{n} -c_j x_j \quad \text{s.t. (2.3) and (2.4).}$$

A standard procedure for solving a linear program of the form (2.2) to (2.4) is the **simplex method**. We will illustrate this method at first by an example.

Example 2.7: Consider the problem

$$\begin{array}{lrcl} \max & 5x_1 + 4x_2 + 3x_3 & & \\ \text{s.t.} & 2x_1 + 3x_2 + x_3 & \leq & 5 \\ & 4x_1 + x_2 + 2x_3 & \leq & 11 \\ & 3x_1 + 4x_2 + 2x_3 & \leq & 8 \\ & x_1, x_2, x_3 & \geq & 0 \end{array}$$

The first step is to transform the inequalities into equalities by introducing so-called **slack variables** $x_4, x_5, x_6 \geq 0$:

$$\begin{array}{lrcl} \max & 5x_1 + 4x_2 + 3x_3 & & \\ \text{s.t.} & 2x_1 + 3x_2 + x_3 + x_4 & = & 5 \\ & 4x_1 + x_2 + 2x_3 + x_5 & = & 11 \\ & 3x_1 + 4x_2 + 2x_3 + x_6 & = & 8 \\ & x_1, x_2, \ldots, x_6 & \geq & 0 \end{array} \tag{2.7}$$

Clearly, this linear program is equivalent to the previous one. Now we write (2.7) as

$$\begin{array}{lrcl} \max & z & & \\ \text{s.t.} & x_4 & = & 5 - 2x_1 - 3x_2 - x_3 \leftarrow \\ & x_5 & = & 11 - 4x_1 - x_2 - 2x_3 \\ & x_6 & = & 8 - 3x_1 - 4x_2 - 2x_3 \\ \hline & z & = & 5x_1 + 4x_2 + 3x_3 \\ & & & \uparrow \\ & & & x_1, x_2, \ldots, x_6 \geq 0 \end{array} \tag{2.8}$$

If we set $x_1 = x_2 = x_3 = 0$, we get a feasible solution with $x_4 = 5$, $x_5 = 11$, $x_6 = 8$ and objective value $z = 0$. z can be increased by increasing x_1 while

keeping $x_2 = x_3 = 0$. The variable x_1 can be increased as long as x_4, x_5, and x_6 remain non-negative, i.e. as long as the following inequalities are satisfied:

$$\begin{array}{lll} x_4 = \ 5 - 2x_1 \geq 0 & & x_1 \leq \frac{5}{2} \\ x_5 = 11 - 4x_1 \geq 0 & \text{or equivalently} & x_1 \leq \frac{11}{4} \\ x_6 = \ 8 - 3x_1 \geq 0 & & x_1 \leq \frac{8}{3} \end{array}$$

Therefore we can increase x_1 up to $x_1 = \min\{\frac{5}{2}, \frac{11}{4}, \frac{8}{3}\} = \frac{5}{2}$. Note that in (2.8) the arrows in the column of x_1 and the row of x_4 indicate that the variable x_1 can be increased as long as $x_4 \geq 0$ holds.

If we set $x_2 = x_3 = 0$, $x_1 = \frac{5}{2}$ which provide $x_4 = 0$, $x_5 = 1$, $x_6 = \frac{1}{2}$, we get a new feasible solution with improved value $z = \frac{25}{2}$. In (2.8) we got a feasible solution by setting the variables on the right hand side equal to 0. In the new solution we have $x_2 = x_3 = x_4 = 0$. Therefore we rewrite (2.8) in such a way that the variables x_1, x_5, x_6, and z are expressed in terms of x_2, x_3, x_4. This is accomplished by rewriting the first equation in (2.8) such that x_1 appears on the left hand side and by substituting the expression for x_1 into the other equations:

$$\begin{array}{rl} x_1 = & \frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4 \\ x_5 = 11 - 4\left(\frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4\right) & - \ x_2 - 2x_3 = \ 1 + 5x_2 \qquad + 2x_4 \\ x_6 = \ 8 - 3\left(\frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4\right) & - 4x_2 - 2x_3 = \ \frac{1}{2} + \frac{1}{2}x_2 - \frac{1}{2}x_3 + \frac{3}{2}x_4 \\ z \ = \qquad 5\left(\frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4\right) & + 4x_2 + 3x_3 = \frac{25}{2} - \frac{7}{2}x_2 + \frac{1}{2}x_3 - \frac{5}{2}x_4 \end{array}$$

Hence our new systems reads as follows:

$$\begin{array}{l} x_1 = \ \frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4 \\ x_5 = \ 1 + 5x_2 \qquad + 2x_4 \\ \underline{x_6 = \ \frac{1}{2} + \frac{1}{2}x_2 - \frac{1}{2}x_3 + \frac{3}{2}x_4 \ \leftarrow} \\ z = \frac{25}{2} - \frac{7}{2}x_2 + \frac{1}{2}x_3 - \frac{5}{2}x_4 \\ \qquad\qquad\qquad\qquad\qquad\qquad \uparrow \end{array} \tag{2.9}$$

Now, the only way to increase z by increasing one variable on the right hand side, is to increase x_3.

We can increase x_3 as long as $x_1 = \frac{5}{2} - \frac{1}{2}x_3 \geq 0$ and $x_6 = \frac{1}{2} - \frac{1}{2}x_3 \geq 0$. Thus, the new solution is $x_3 = \min\{5, 1\} = 1$, $x_1 = 2$, $x_2 = 0$, $x_4 = 0$, $x_5 = 1$, $x_6 = 0$ and the corresponding system of equations is:

$$\begin{array}{l} x_3 = \ 1 + \ x_2 + 3x_4 - 2x_6 \\ x_1 = \ 2 - 2x_2 - 2x_4 + \ x_6 \\ \underline{x_5 = \ 1 + 5x_2 + 2x_4} \\ z = 13 - 3x_2 - \ x_4 - \ x_6 \end{array} \tag{2.10}$$

Because the coefficients of all variables x_j in the last equation of (2.10) are non-positive, there is no way to improve z by choosing other values $x_2, x_4, x_6 \geq 0$

instead of $x_2 = x_4 = x_6 = 0$. Since (2.10) is equivalent to (2.8), the solution $x_1 = 2$, $x_2 = 0$, $x_3 = 1$, $x_4 = 0$, $x_5 = 1$, $x_6 = 0$ is an optimal solution of the linear program. □

The schemes (2.8) to (2.10) are called **dictionaries**. The variables on the left hand side in a dictionary are called **basic variables**. The variables on the right hand side are called **non-basic variables**. A dictionary is called **feasible** if setting all non-basic variables to zero and evaluating the basic variables accordingly, leads to a feasible solution. Dictionaries (2.8) to (2.10) are feasible. The simplex algorithm starts with a feasible dictionary. In each iteration it moves from one feasible dictionary to the next by replacing a basic variable by a non-basic variable.

If a linear program is given in standard form (2.2) to (2.4) and $b_i \geq 0$ for $i = 1, \ldots, m$ holds, then a first feasible dictionary can be provided by introducing slack variables and choosing the slack variables as basic variables.

If B is the set of indices belonging to basic variables, then the corresponding dictionary can be written in the form

$$\begin{aligned} x_i &= \overline{b}_i - \sum_{j \notin B} \overline{a}_{ij} x_j \qquad i \in B \\ \hline z &= \overline{z} + \sum_{j \notin B} \overline{c}_j x_j \end{aligned} \tag{2.11}$$

The system (2.11) is feasible if $\overline{b}_i \geq 0$ for all $i \in B$. If (2.11) is feasible, we may distinguish three cases:

- If $\overline{c}_j \leq 0$ for all $j \notin B$, then x^* with $x_j^* = 0$ for all $j \notin B$ and $x_i^* = \overline{b}_i$ for all $i \in B$ is an optimal solution. Furthermore, $\overline{z}$ is the optimal solution value.

- If an index $l \notin B$ exists with $\overline{c}_l > 0$ and $\overline{a}_{il} \leq 0$ for all $i \in B$, then for arbitrary $x_l \geq 0$ we have

  $$x_i = \overline{b}_i - \overline{a}_{il} x_l \geq 0 \text{ for all } i \in B.$$

  Thus, $z = \overline{z} + \overline{c}_l x_l$ will be arbitrary large if x_l is arbitrary large, i.e. the linear program is unbounded.

- If an index $l \notin B$ with $\overline{c}_l > 0$ and at least one index i with $\overline{a}_{il} > 0$ exist, then we calculate an index $k \in B$ such that

  $$\frac{\overline{b}_k}{\overline{a}_{kl}} = \min \left\{ \frac{\overline{b}_i}{\overline{a}_{il}} \mid i \in B, \overline{a}_{il} > 0 \right\} \tag{2.12}$$

  and replace the basic variable x_k by the non-basic variable x_l. We transform the dictionary (2.11) into a new dictionary for the basis $B \backslash \{k\} \cup \{l\}$. This transformation is called **pivoting**, $\overline{a}_{kl}$ is the corresponding **pivot element**. In a **simplex iteration** we choose an appropriate pivot element and do the pivoting.

By a simplex iteration with pivot element $\overline{a}_{kl}$ the basic solution

$$x_i = \begin{cases} \overline{b}_i, & \text{for } i \in B \\ 0, & \text{otherwise} \end{cases}$$

is transformed into the new basic solution

$$x_i = \begin{cases} \overline{b}_i - \overline{a}_{il}\frac{\overline{b}_k}{\overline{a}_{kl}}, & \text{for } i \in B \\ \frac{\overline{b}_k}{\overline{a}_{kl}}, & \text{for } i = l \\ 0, & \text{otherwise.} \end{cases}$$

It may happen that $\overline{b}_k = 0$. In this case, despite the fact that the basis changes, the basic solution and the objective value do not change. Such an iteration is called **degenerate**.

In the absence of degeneracy the objective value increases in each iteration. This means that all basic sets B are different. Because the number of different sets B is bounded by $\binom{m}{n}$, the procedure must stop after a finite number of iterations.

In the presence of degeneracy it is possible that the same dictionary appears in different iterations. The sequence of dictionaries between these two appearances is called a **cycle**. If a cycle appears, the simplex algorithm does not terminate.

We have several choices for the so-called **entering variable** x_l if there are several indices $j \notin B$ with $\overline{c}_j > 0$. Also a **leaving variable** x_k defined by (2.12) may not be unique. In the so-called **smallest index tie-breaking rule** always candidates x_l and x_k with the smallest indices are chosen. It can be shown that cycles can be avoided if the smallest index rule is used.

We conclude that when a linear program is not unbounded, the simplex method (starting with a feasible dictionary) can always find an optimal dictionary after a finite number of iterations if the smallest index tie-breaking rule is applied. It remains to show how to find a feasible dictionary for the linear program (2.2) to (2.4) in the case that $b_i < 0$ for some indices i.

In this case we consider the auxiliary linear program

$$\min\ x_0 \qquad (\text{ or } \max\ -x_0\ ) \tag{2.13}$$

$$\text{s.t.} \qquad \sum_{j=1}^{n} a_{ij}x_j - x_0 \le b_i \qquad (i = 1, \ldots, m) \tag{2.14}$$

$$x_j \ge 0 \qquad (j = 0, 1, \ldots, n) \tag{2.15}$$

where $x_0 \ge 0$ is an additional variable.

The optimal solution value of (2.13) to (2.15) is equal to zero if and only if the linear program (2.2) to (2.4) has a feasible solution.

If we introduce slack variables in the auxiliary linear program, we get the infeasible dictionary

$$\begin{array}{rcl} x_{n+i} &=& b_i - \sum_{j=1}^{n} a_{ij}x_j + x_0 \qquad (i = 1, \ldots, m) \\ \hline w &=& -x_0 \end{array} \tag{2.16}$$

where at least one b_i is negative.

If b_k is the smallest b_i-value, one pivot step with x_0 as entering variable and x_{n+k} as leaving variable provides a feasible dictionary because $b_i - b_k \geq 0$ for $i = 1, \ldots, m$. This feasible dictionary may afterwards be used as a starting dictionary to solve the auxiliary linear program. If in one iteration x_0 competes with some other variable for leaving the basis, we prefer x_0 as leaving variable. As soon as x_0 becomes a non-basic variable, we have reached a feasible solution with $x_0 = 0$ which must be optimal. The corresponding basic variables provide a feasible dictionary for the original problem (2.2) to (2.4).

This method is also called **two-phase method**. In the first phase a feasible basic solution is provided or it is shown that no feasible solution exists. Starting with this basic solution the original problem (2.2) to (2.4) is solved in the second phase.

Example 2.8: Consider the linear program

$$\begin{array}{lrcl} \max & x_1 - x_2 + x_3 & & \\ \text{s.t.} & 2x_1 - x_2 + 2x_3 & \leq & 4 \\ & 2x_1 - x_2 + x_3 & \leq & -5 \\ & -x_1 - 2x_2 - 2x_3 & \leq & -1 \\ & x_1, x_2, x_3 & \geq & 0 \end{array} \tag{2.17}$$

The auxiliary problem is

$$\begin{array}{lrcl} \max & - x_0 & & \\ \text{s.t.} & 2x_1 - x_2 + 2x_3 - x_0 & \leq & 4 \\ & 2x_1 - x_2 + x_3 - x_0 & \leq & -5 \\ & -x_1 - 2x_2 - 2x_3 - x_0 & \leq & -1 \\ & x_0, x_1, x_2, x_3 & \geq & 0 \end{array}$$

By introducing slack-variables x_4, x_5, x_6 we get the infeasible dictionary

$$\begin{array}{rl} x_4 = & 4 - 2x_1 + x_2 - 2x_3 + x_0 \\ x_5 = & -5 - 2x_1 + x_2 - x_3 + x_0 \leftarrow \\ x_6 = & -1 + x_1 + 2x_2 + 2x_3 + x_0 \\ \hline w = & - x_0 \\ & \uparrow \end{array}$$

If we are pivoting with x_0 as entering variable and x_5 as leaving variable, we get the feasible dictionary

$$\begin{array}{rl} x_0 = & 5 + 2x_1 - x_2 + x_3 + x_5 \leftarrow \\ x_4 = & 9 - x_3 + x_5 \\ x_6 = & 4 + 3x_1 + x_2 + 3x_3 + x_5 \\ \hline w = & -5 - 2x_1 + x_2 - x_3 - x_5 \\ & \uparrow \end{array}$$

The next simplex iteration with x_2 as entering variable and x_0 as leaving variable provides the optimal dictionary of the first phase:

$$\begin{aligned} x_2 &= 5 + 2x_1 + x_3 + x_5 - x_0 \\ x_4 &= 9 - x_3 + x_5 \\ x_6 &= 9 + 5x_1 + 4x_3 + 2x_5 - x_0 \\ \hline w &= - x_0 \end{aligned}$$

We set $x_0 = 0$ and substitute $x_2 = 5 + 2x_1 + x_3 + x_5$ in the objective function of (2.17). This provides the starting dictionary for the second phase:

$$\begin{aligned} x_2 &= 5 + 2x_1 + x_3 + x_5 \\ x_4 &= 9 - x_3 + x_5 \\ x_6 &= 9 + 5x_1 + 4x_3 + 2x_5 \\ \hline z &= -5 - x_1 + 0x_3 - x_5 \end{aligned} \tag{2.18}$$

However, (2.18) is already an optimal dictionary for the original problem (2.17). A corresponding optimal basic solution is $x_1 = 0$, $x_2 = 5$, $x_3 = 0$. □

In the first phase we may reach an optimal dictionary in which x_0 is still a basic variable. In the corresponding solution the objective value $z = -x_0$ must be negative. Otherwise, in one of the previous iterations z must have been increased to zero which means that x_0 has been decreased to zero. Thus, x_0 would have competed with some other basic variables for leaving the basis and according to the proposed strategy x_0 would have left the basis, which is a contradiction. Hence, in this situation the optimal objective value of the auxiliary problem is negative and the first phase shows that no feasible solution for the original problem exists.

2.3.2 Duality

Every maximization linear program in standard form induces a corresponding minimization problem which is called the dual linear program. In this subsection we state the important duality theorem, from which optimality conditions can be derived.

The **dual linear program** for

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} c_j x_j \\ \text{s.t.} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad (i = 1, \ldots, m) \\ & x_j \geq 0 \qquad (j = 1, \ldots, n) \end{aligned} \tag{2.19}$$

is given by:

$$\begin{array}{lrcll} \min & \sum\limits_{i=1}^{m} b_i y_i & & & \\ \text{s.t.} & \sum\limits_{i=1}^{m} a_{ij} y_i & \geq & c_j & (j = 1, \ldots, n) \\ & y_i & \geq & 0 & (i = 1, \ldots, m) \end{array} \tag{2.20}$$

In connection with duality (2.19) is also called **primal linear program**.

If $x = (x_1, \ldots, x_n)$ is a feasible solution for the primal problem and $y = (y_1, \ldots, y_m)$ is a feasible solution for the dual problem, then the following inequality holds:

$$\sum_{j=1}^{n} c_j x_j \leq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \right) y_i \leq \sum_{i=1}^{m} b_i y_i. \tag{2.21}$$

Thus, the objective value of any feasible solution for the dual problem is an upper bound for the optimal solution value of the primal problem. Symmetrically, the objective value of any feasible solution for the primal problem is a lower bound for the optimal solution value of the dual problem. If for feasible solutions x and y the equality $\sum\limits_{j=1}^{n} c_j x_j = \sum\limits_{i=1}^{m} b_i y_i$ holds, then both x and y are optimal solutions.

Additionally, the following theorem is valid, which is one of the main results in duality theory.

Theorem 2.2 If the primal linear program (2.19) has an optimal solution $x^* = (x_1^*, \ldots, x_n^*)$, then the dual linear program (2.20) has an optimal solution $y^* = (y_1^*, \ldots, y_m^*)$ and

$$\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} b_i y_i^*. \tag{2.22}$$

Proof: Let x^* be the optimal solution provided by the simplex algorithm. Then the dictionary

$$\begin{array}{rcl} x_{n+i} & = & b_i - \sum\limits_{j=1}^{n} a_{ij} x_j \qquad (i = 1, \ldots, m) \\ z & = & \sum\limits_{j=1}^{n} c_j x_j \end{array} \tag{2.23}$$

is equivalent to the optimal dictionary in which the last equation (which corresponds to the objective function) can be written in the form

$$z = z^* + \sum_{j=1}^{n+m} \overline{c}_j x_j \tag{2.24}$$

where $\overline{c}_j \leq 0$ for $j = 1, \ldots, n+m$ and $\overline{c}_j = 0$ for the basic variables x_j. Furthermore, $z^* = \sum\limits_{j=1}^{n} c_j x_j^*$ is the optimal objective value.

We define

$$y_i^* = -\overline{c}_{n+i} \text{ for } i = 1, \ldots, m. \tag{2.25}$$

With equations (2.23) to (2.25) we get

$$\begin{aligned} \sum_{j=1}^{n} c_j x_j \quad = z = z^* + \quad \sum_{j=1}^{n+m} \overline{c}_j x_j \quad &= \quad z^* + \sum_{j=1}^{n} \overline{c}_j x_j - \sum_{i=1}^{m} (-\overline{c}_{n+i}) x_{n+i} \\ &= \quad z^* + \sum_{j=1}^{n} \overline{c}_j x_j - \sum_{i=1}^{m} y_i^* \left(b_i - \sum_{j=1}^{n} a_{ij} x_j \right) \end{aligned}$$

or equivalently

$$\sum_{j=1}^{n} c_j x_j = \left(z^* - \sum_{i=1}^{m} y_i^* b_i \right) + \sum_{j=1}^{n} \left(\overline{c}_j + \sum_{i=1}^{m} a_{ij} y_i^* \right) x_j. \tag{2.26}$$

This identity holds for all n-vectors $x = (x_1, \ldots, x_n)$. Choosing the zero vector $x = 0$ and the unit vectors $x = e^i$ with $e_j^i = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{otherwise} \end{cases}$ for $i = 1, \ldots, n$, we get

$$\begin{aligned} \sum_{j=1}^{n} c_j x_j^* \quad &= \quad z^* = \sum_{i=1}^{m} b_i y_i^* \\ c_j \quad &= \quad \overline{c}_j + \sum_{i=1}^{m} a_{ij} y_i^* \leq \sum_{i=1}^{m} a_{ij} y_i^* \quad \text{for } j = 1, \ldots, n \\ y_i^* \quad &= \quad -\overline{c}_{n+i} \geq 0 \qquad\qquad\qquad\quad \text{for } i = 1, \ldots, m \end{aligned}$$

because $\overline{c}_j \leq 0$ for $j = 1, \ldots, n + m$.

We conclude that y^* is a feasible solution for the dual linear program which satisfies $\sum_{i=1}^{m} b_i y_i^* = \sum_{j=1}^{n} c_j x_j^*$. Therefore, y^* must be optimal. □

The proof also shows that an optimal dictionary of the primal program provides an optimal solution y^* for the dual program: Set $y_i^* := 0$ if the slack variable x_{n+i} is a basic variable in the optimal dictionary and $y_i^* := -\overline{c}_{n+i}$ otherwise where $\overline{c}_{n+i}$ is the coefficient of the slack variable x_{n+i} in the objective function row in the optimal primal dictionary.

The dual of the dual linear program is equivalent to the primal problem. This can be seen by writing the dual linear program as a linear program in standard form and calculating the dual of it. We conclude that the primal problem has an optimal solution if and only if the dual problem has an optimal solution.

Condition (2.21) implies that if the primal (dual) problem is unbounded, then the dual (primal) problem has no feasible solution. It is also possible that both primal and dual problem have no feasible solution as the following example shows.

Example 2.9: The dual of

$$\begin{array}{lrcl} \max & 2x_1 - x_2 & & \\ \text{s.t.} & x_1 - x_2 & \leq & 1 \\ & -x_1 + x_2 & \leq & -2 \\ & x_1, x_2 & \geq & 0 \end{array}$$

can be written as

$$\begin{array}{lrcl} \min & y_1 - 2y_2 & & \\ \text{s.t.} & y_1 - y_2 & \geq & 2 \\ & -y_1 + y_2 & \geq & -1 \\ & y_1, y_2 & \geq & 0 \end{array}$$

Obviously, both problems have no feasible solution. □

primal \ dual	optimal	infeasible	unbounded
optimal	×		
infeasible		×	×
unbounded		×	

Table 2.1: Possible combinations for primal and dual problems

In Table 2.1 all possible combinations for primal and dual problems are marked (all other combinations are not possible).

The next theorem provides necessary and sufficient conditions for a pair of feasible solutions x^* and y^* of a primal and dual linear program, respectively, to be optimal.

Theorem 2.3 Let $x^* = (x_1^*, \ldots, x_n^*)$ be a feasible solution of (2.19) and $y^* = (y_1^*, \ldots, y_m^*)$ be a feasible solution of (2.20). Then x^* and y^* are both optimal if and only if the following two conditions hold:

$$\sum_{i=1}^{m} a_{ij} y_i^* = c_j \quad \text{or} \quad x_j^* = 0 \quad \text{for } j = 1, \ldots, n \tag{2.27}$$

$$\sum_{j=1}^{n} a_{ij} x_j^* = b_i \quad \text{or} \quad y_i^* = 0 \quad \text{for } i = 1, \ldots, m \tag{2.28}$$

Proof: Due to (2.19) and (2.20) we have

$$c_j x_j^* \leq \left(\sum_{i=1}^{m} a_{ij} y_i^* \right) x_j^* \quad \text{for } j = 1, \ldots, n \tag{2.29}$$

$$\left(\sum_{j=1}^{n} a_{ij} x_j^* \right) y_i^* \leq b_i y_i^* \quad \text{for } i = 1, \ldots, m \tag{2.30}$$

implying

$$\sum_{j=1}^{n} c_j x_j^* \leq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i^* \right) x_j^* = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j^* \right) y_i^* \leq \sum_{i=1}^{m} b_i y_i^*. \tag{2.31}$$

Theorem 2.2 implies that both x^* and y^* are optimal if and only if

$$\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} b_i y_i^*$$

holds. Due to (2.31) this is the case if and only if in all inequalities (2.29) and (2.30) equality holds or equivalently, conditions (2.27) and (2.28) are satisfied. □

The following theorem is an immediate consequence of the previous theorems.

Theorem 2.4 (Complementary Slackness Condition): A feasible solution $x^* = (x_1^*, \ldots, x_n^*)$ of the primal problem (2.19) is optimal if and only if a feasible solution $y^* = (y_1^*, \ldots, y_m^*)$ of the dual problem exists satisfying the following two conditions:

$$\begin{aligned} x_j^* > 0 \quad &\text{implies} \quad \sum_{i=1}^{m} a_{ij} y_i^* = c_j \qquad \text{for } j = 1, \ldots, n \\ \sum_{j=1}^{n} a_{ij} x_j^* < b_i \quad &\text{implies} \quad y_i^* = 0 \qquad \text{for } i = 1, \ldots, m. \end{aligned} \tag{2.32}$$

2.3.3 The revised simplex method

In subsection 2.3.1 we described the simplex algorithm in terms of dictionaries which are updated in each iteration. Another implementation using matrices instead of dictionaries is the so-called **revised simplex method**, which will be described in this subsection.

After the introduction of slack variables the linear program (2.2) to (2.4) can be written in matrix form as

$$\begin{array}{lrcl} \max & cx & & \\ \text{s.t.} & Ax & = & b \\ & x & \geq & 0 \end{array} \tag{2.33}$$

where $c = (c_1, \ldots, c_n, 0, \ldots, 0)$,

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} & 1 & & 0 \\ \vdots & & \vdots & & \ddots & \\ a_{m1} & \cdots & a_{mn} & 0 & & 1 \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \text{ and } x = \begin{pmatrix} x_1 \\ \vdots \\ x_{n+m} \end{pmatrix}.$$

To write a dictionary

$$\begin{array}{rcl} x_i = \bar{b}_i - \sum\limits_{j \notin B} \bar{a}_{ij} x_j & & i \in B \\ \hline z = z^* + \sum\limits_{j \notin B} \bar{c}_j x_j & & \end{array} \tag{2.34}$$

in matrix form we set $x = (x_B, x_N)$, $c = (c_B, c_N)$, $A = (A_B, A_N)$ where the components of x have been reordered in such a way that the basic variables are followed by the non-basic variables. The components in c and the columns in A are reordered accordingly. With this notation (2.33) writes

$$\begin{array}{ll} \max & c_B x_B + c_N x_N \\ \text{s.t.} & A_B x_B + A_N x_N = b \\ & x_B \geq 0,\ x_N \geq 0. \end{array} \tag{2.35}$$

The $m \times m$-matrix A_B must be regular because

$$A_B x_B = b \tag{2.36}$$

has a unique solution, which can be seen as follows. Each solution x_B^* of (2.36) provides the solution $(x_B^*, 0)$ of (2.35) which also must be a solution of (2.34) implying $x_B^* = \overline{b}$.

Furthermore, $A_B x_B + A_N x_N = b$ is equivalent to $x_B = A_B^{-1} b - A_B^{-1} A_N x_N$ and we have

$$\begin{aligned} z = c_B x_B + c_N x_N &= c_B(A_B^{-1} b - A_B^{-1} A_N x_N) + c_N x_N \\ &= c_B A_B^{-1} b + (c_N - c_B A_B^{-1} A_N) x_N. \end{aligned}$$

From now on we will replace the index set B by the corresponding submatrix A_B of A, i.e. we set $B := A_B$. With this notation the corresponding dictionary writes

$$\begin{array}{rcl|rcl} x_B & = & B^{-1}b & - & B^{-1}A_N x_N \\ \hline z & = & c_B B^{-1} b & + & (c_N - c_B B^{-1} A_N) x_N. \end{array}$$

The corresponding basic solution has the form $(x_B^*, x_N^*) = (B^{-1}b, 0)$.

Given the **basic matrix** B, the vector $y = c_B B^{-1}$ can be calculated by solving the system of linear equations

$$yB = c_B.$$

Similarly, a column d of the matrix $B^{-1}A_N$ can be calculated by solving the system

$$Bd = a$$

(which is equivalent to $d = B^{-1}a$) where a is the corresponding column in A_N.

With these notations an iteration of the simplex method for a maximization problem is shown in Figure 2.10. Note, that we do not have to solve another system of equations for the calculation of $x_B^* = B^{-1}b$ since x_B^* can be updated as described in the algorithm.

It remains to determine the complexity of the simplex algorithm. Although this algorithm is very fast in practice, it is not a polynomial-time algorithm. There are counterexamples for which the simplex algorithm with the largest coefficient rule (i.e. in Step 2 always a non-basic variable x_j with largest c_j-value is chosen)

1. Solve the system $yB = c_B$.
2. Choose a column a entering the basis. This may be any column a^j of A_N such that ya^j is less than the corresponding component c_j of c_N. If no such column exists, then the current basic solution is optimal.
3. Solve the system $Bd = a$.
4. Find the largest $t \geq 0$ such that $x_B^* - td \geq 0$. If there is no such t, then the problem is unbounded. Otherwise, at least one component of $x_B^* - td$ equals zero and the corresponding variable is chosen as variable which leaves the basis.
5. Set the value of the entering variable to t and replace the values x_B^* of the basic variables by $x_B^* - td$. Replace the leaving column of B by the entering column a.

Figure 2.10: Iteration of the revised simplex method for a maximization problem

needs an exponential number of iterations. Additional counterexamples exist for other pivoting rules. It is still an open question whether an alternative pivoting rule exists, for which the simplex algorithm always needs a polynomial number of iterations. Nevertheless, linear programming problems belong to the class $\mathcal{P}$. In 1979, the so-called **ellipsoid method** was proposed, which is a polynomial-time algorithm for linear programming. Although this algorithm has a better worst-case complexity than the simplex algorithm, in practice the simplex algorithm is preferred since it is much faster in the average case.

2.3.4 Linear programs with integer variables

If in the linear program (2.2) to (2.4) all variables x_j are restricted to be integers, then this restricted version is called an **integer linear program**. A **binary linear program** is an integer linear program with the additional restriction $x_j \in \{0, 1\}$ for all variables x_j. A **mixed integer linear program** (MIP) is a linear program in which we have continuous and integer variables.

Contrary to their continuous counterparts integer linear programs are NP-hard. While even very large continuous linear programs can be solved by commercial software packages in a few minutes today, this is not the case for integer programs. For these problems branch-and-bound algorithms (cf. Section 2.5) have to be applied which are more time consuming.

Next we will show that the RCPSP can be formulated as binary linear program. For this purpose we introduce so-called **time-indexed variables** x_{jt} for $j \in V = \{0, 1, \ldots, n, n+1\}$ and $t = 0, \ldots, T$ where T is a given time horizon for the

project. We set

$$x_{jt} := \begin{cases} 1, & \text{if activity } j \text{ completes at time } t \\ 0, & \text{otherwise.} \end{cases}$$

Then, using the notations from Section 1.1.1, the RCPSP may be formulated as

$$\min \quad \sum_{t=0}^{T} t x_{n+1,t} \tag{2.37}$$

$$\text{s.t.} \quad \sum_{t=0}^{T} x_{jt} = 1 \qquad (j \in V) \tag{2.38}$$

$$\sum_{t=0}^{T} t x_{it} - \sum_{t=0}^{T} (t - p_j) x_{jt} \leq 0 \qquad ((i,j) \in A) \tag{2.39}$$

$$\sum_{j=1}^{n} r_{jk} \sum_{\tau=t+1}^{t+p_j} x_{j\tau} \leq R_k \qquad (k = 1, \ldots, r;\ t = 0, \ldots, T - p_j) \tag{2.40}$$

$$x_{jt} \in \{0,1\} \quad (j \in V;\ t = 0, \ldots, T) \tag{2.41}$$

In (2.37) the completion time of the dummy terminating activity $n+1$ (i.e. the makespan) is minimized. Due to the constraints (2.38) and (2.41) exactly one completion time is chosen for each activity. Constraints (2.39) ensure that the precedence constraints $C_i - S_j \leq 0$ for $(i,j) \in A$ are respected. Finally, restrictions (2.40) guarantee that the resource constraints are satisfied (since activity j is processed in the interval $[t, t+1[$ if and only if it completes in the interval $[t+1, t+p_j[$).

If time-lags l_{ij} are given, constraints (2.39) may be replaced by

$$\sum_{t=0}^{T} t x_{it} - \sum_{t=0}^{T} (t - l_{ij}) x_{jt} \leq 0 \qquad ((i,j) \in A). \tag{2.42}$$

If we have time-dependent resource profiles, the right hand side of (2.40) has to be replaced by $R_k(t)$.

Similarly, the multi-mode RCPSP can be formulated if we set

$$x_{jmt} := \begin{cases} 1, & \text{if activity } j \text{ is processed in mode } m \text{ and completes at time } t \\ 0, & \text{otherwise.} \end{cases}$$

2.3.5 Delayed column generation techniques

In this section we discuss a technique which is useful to solve linear programs with a huge number of variables. We will illustrate this technique with the cutting-stock problem introduced in Application 1.1.

Recall that in this problem we have standard rolls of width W which have to be cut into smaller rolls (so-called finals) of widths w_i for $i = 1, \ldots, m$. Assume that for $i = 1, \ldots, m$ there is a demand of b_i finals with width w_i. We want to cut the large rolls such that the demand are satisfied and the number of sliced large rolls is minimized.

First, we consider all ways of cutting rolls of demanded widths w_i from the standard rolls. A corresponding cutting pattern is given by an integer vector $a = (a_1, \ldots, a_m)$ with

$$w_1 a_1 + \ldots + w_m a_m \leq W. \tag{2.43}$$

Here a_i denotes the number of times width w_i is cut from a standard roll. Consider all possible cutting patterns which we enumerate by $a^1, \ldots, a^n$ and denote by x_j the number of times cutting pattern a^j is applied. Then the cutting-stock problem is equivalent to the integer linear program

$$\begin{array}{lrcll} \min & \sum_{j=1}^{n} x_j & & & \\ \text{s.t.} & \sum_{j=1}^{n} a_i^j x_j & = & b_i & (i = 1, \ldots, m) \\ & x_j & \geq & 0 \text{ and integer} & (j = 1, \ldots, n) \end{array} \tag{2.44}$$

This formulation has two disadvantages. The first disadvantage is that it is an integer program, which is hard to solve. The second disadvantage is that we usually have a huge number of cutting patterns, i.e. the linear program has a huge number of variables.

One may handle the first difficulty by solving the continuous version of (2.44) and by rounding fractional solutions up or down. This procedure is quite satisfactory in typical industrial applications. If m different widths of finals are ordered, then the fractional optimum produced by the simplex method involves at most m nonzero variables x_j. Hence the cost difference between the rounded off solution and the true integer optimum will be at most the cost of m standard rolls (and often considerably less). Since a typical value of m is more than 50, and most final widths are ordered in hundreds or thousands, the possible cost increment due to inefficient cuts is small.

Example 2.10: Assume that from rolls of standard width $W = 100$ we have to cut

97	finals of width	45,
610	finals of width	36,
395	finals of width	31,
211	finals of width	14.

Four feasible cutting patterns are

$$\begin{pmatrix}2\\0\\0\\0\end{pmatrix}, \quad \begin{pmatrix}0\\2\\0\\2\end{pmatrix}, \quad \begin{pmatrix}0\\2\\0\\0\end{pmatrix}, \quad \begin{pmatrix}0\\1\\2\\0\end{pmatrix}.$$

Considering the corresponding variables as basic variables for (2.44), we get a basic solution x^* with $x_1^* = 48.5$, $x_2^* = 105.5$, $x_3^* = 100.75$, $x_4^* = 197.5$ and $x_j^* = 0$ for all other cutting patterns by solving

$$Bx = \begin{pmatrix}2&0&0&0\\0&2&2&1\\0&0&0&2\\0&2&0&0\end{pmatrix}\begin{pmatrix}x_1\\x_2\\x_3\\x_4\end{pmatrix} = \begin{pmatrix}97\\610\\395\\211\end{pmatrix}.$$

We claim that x^* is an optimal solution for the continuous version of problem (2.44). To prove this we have to show that the optimality conditions

$$y_1a_1 + y_2a_2 + y_3a_3 + y_4a_4 - 1 \leq 0 \tag{2.45}$$

are satisfied for all feasible patterns $a = (a_1, a_2, a_3, a_4)$ where $y = (y_1, y_2, y_3, y_4)$ is the solution of $yB = c_B = (1, 1, 1, 1)$. The system $yB = (1, 1, 1, 1)$ has the form

$$\begin{aligned} 2y_1 &= 1\\ 2y_2 + 2y_4 &= 1\\ 2y_2 &= 1\\ y_2 + 2y_3 &= 1 \end{aligned}$$

Thus, $y = (\frac{1}{2}, \frac{1}{2}, \frac{1}{4}, 0)$ and (2.45) writes

$$\frac{1}{2}a_1 + \frac{1}{2}a_2 + \frac{1}{4}a_3 \leq 1 \quad \text{or} \quad 50a_1 + 50a_2 + 25a_3 \leq 100. \tag{2.46}$$

Furthermore, all feasible patterns are defined by all integer vectors $a = (a_1, a_2, a_3, a_4)$ with $a_i \geq 0$ satisfying

$$45a_1 + 36a_2 + 31a_3 + 14a_4 \leq 100. \tag{2.47}$$

To prove that (2.46) and (2.47) hold for all non-negative vectors a, we show that no vector a exists satisfying (2.47) and

$$50a_1 + 50a_2 + 25a_3 > 100. \tag{2.48}$$

Assume to the contrary that such a vector exists. Then (2.47) implies $a_3 \leq 3$. If $a_3 = 3$, then (2.47) implies $a_1 = a_2 = 0$, which is a contradiction to (2.48). If $a_3 = 2$, then (2.47) implies $a_3 = 0$ and $a_2 \leq 1$, which again contradicts (2.48). If $a_3 = 1$, then (2.47) implies $a_1 + a_2 \leq 1$, which also contradicts (2.48).

Thus, we have $a_3 = 0$ and (2.47) writes $45a_1 + 36a_2 + 14a_4 \leq 100$. This implies $a_1 + a_2 \leq 2$ and $50a_1 + 50a_2 + 25a_3 = 50(a_1 + a_2) \leq 100$, which contradicts (2.48).

The optimal solution x^* of the continuous version of (2.44) provides the lower bound

$$x_1^* + x_2^* + x_3^* + x_4^* = 48.5 + 105.5 + 100.75 + 197.5 = 452.25 \tag{2.49}$$

for the optimal objective value of the integer linear program (2.44).

If we round down the rational solution values $x_1^*, x_2^*, x_3^*, x_4^*$, i.e. if we apply 48 times the cutting pattern $(2, 0, 0, 0)$, 105 times the cutting pattern $(0, 2, 0, 2)$, 100 times the cutting pattern $(0, 2, 0, 0)$, and 197 times the cutting pattern $(0, 1, 2, 0)$, then

96	finals of width	45
607	finals of width	36
394	finals of width	31
210	finals of width	14

are provided.

The total number of sliced rolls sum up to 450. We can satisfy the demand by three additional cuts of the form $(0, 2, 0, 0)$, $(1, 0, 1, 1)$ and $(0, 1, 0, 0)$. The corresponding solution with 453 rolls must be optimal because due to (2.49) the value $\lceil 452.25 \rceil = 453$ is a lower bound for the optimal integer solution. □

To solve the continuous version of (2.44) we apply the revised simplex algorithm where we may choose the $m \times m$-diagonal matrix with diagonal elements $d_i = \lfloor \frac{W}{w_i} \rfloor$ as basis matrix B to start with.

In a general simplex iteration we calculate a vector y with $yB = c_B$. Next we try to find an integer vector $a = (a_1, \ldots, a_m)$ with $a_i \geq 0$ satisfying

$$\sum_{i=1}^{m} w_i a_i \leq r \text{ and } \sum_{i=1}^{m} y_i a_i > 1. \tag{2.50}$$

If such a vector a exists, we may replace some column in B by a. Otherwise, we have found an optimal basic solution for the continuous version of (2.44).

To find one or more solutions of (2.50) we may apply an enumeration procedure called branch-and-bound, which will be discussed in Section 2.5.

The proposed method to solve the cutting-stock problem is called **delayed column generation**. It is a general method which can be applied to other problems as well. We conclude this section by describing the main ideas of delayed column generation and by presenting a generic column generation algorithm.

If we apply the revised simplex method to a linear program as in Figure 2.10, then it is not necessary that the whole matrix A_N of non-basic columns is available. It is sufficient to store the current basic matrix B and to have a procedure at hand which calculates an entering column a^j in Step 2 (i.e. a column a^j of A_N satisfying $ya^j < c_j$), or proves that no such column exists. This problem is called **pricing problem**, a procedure for solving it is a **pricing procedure**.

Usually, the pricing procedure does not calculate only a single column a^j, but a set of columns which possibly may enter the basis during the next iterations. These columns are added to a so-called **working set** of columns, furthermore, some non-basic columns may be deleted from this set. Afterwards, the linear program is solved to optimality with the current working set of columns. If the pricing procedure states that no entering column exists, the current basic solution is optimal and the algorithm is stopped.

```
Algorithm Column Generation
1.  Initialize;
2.  WHILE CalculateColumns produces new columns DO
3.        InsertDeleteColumns;
4.        Optimize;
5.  END
```

Figure 2.11: A generic column generation algorithm

A generic column generation algorithm is summarized in Figure 2.11. In this algorithm we have the following generic procedures: `Initialize` provides a basic starting solution, `CalculateColumns` is the pricing procedure, the procedure `InsertDeleteColumns` organizes the insertion and deletion of columns, and `Optimize` solves the linear program restricted to the current working set of columns to optimality.

Several variants of the procedure `InsertDeleteColumns` may be provided (varying the maximal number of columns in the working set, using different strategies to delete columns from the working set, etc.). The procedure `Optimize` can be done by a standard LP-solver. Thus, only `Initialize` and `CalculateColumns` must be implemented problem-specifically. Realizations of these procedures for other problems will be presented in later sections.

2.4 Network Flow Algorithms

In this section we consider a special class of problems which can be formulated as linear programs, namely the so-called network flow problems. Due to specific properties they can be solved more efficiently than general linear programs. After introducing some basic concepts, solution algorithms for the maximum flow problem, the minimum cut problem, and the minimum cost flow problem are presented.

2.4.1 The minimum cost flow problem

The **minimum cost flow problem** can be formulated as follows. Let $G = (V, A)$ be a directed network defined by a set V of n nodes and a set A of m

directed arcs. Each arc $(i,j) \in A$ has an associated **cost** c_{ij}, where c_{ij} denotes the cost for sending one unit of flow through the arc (i,j). Furthermore, with each arc $(i,j) \in A$ a **capacity** u_{ij} is associated, where u_{ij} is a non-negative integer or $u_{ij} := \infty$. If u_{ij} is finite, it defines an upper bound for the flow through (i,j). Each node $i \in V$ has an associated integer number $b(i)$ representing its supply or demand, respectively. If $b(i) > 0$, node i is a **supply node**, if $b(i) < 0$, node i is a **demand node**. If $b(i) = 0$, then i is a **transshipment node**.

The decision variables x_{ij} in the minimum cost flow problem represent flows on the arcs $(i,j) \in A$. One has to find a flow on each arc such that

- all supplies and demands $b(i)$ of the nodes are satisfied,
- all capacities u_{ij} of the arcs are respected, and
- the total cost of flows on all arcs is minimized.

The minimum cost flow problem can be formulated as the linear program

$$\min \quad \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{2.51}$$

$$\text{s.t.} \quad \sum_{\{j|(i,j)\in A\}} x_{ij} - \sum_{\{j|(j,i)\in A\}} x_{ji} = b(i) \qquad i \in V \tag{2.52}$$

$$0 \le x_{ij} \le u_{ij} \qquad (i,j) \in A \tag{2.53}$$

A vector $x = (x_{ij})$ satisfying (2.52) and (2.53) is called a **feasible flow**, a feasible flow minimizing the total cost is called an **optimal flow**.

Additionally, $\sum_{i=1}^{n} b(i) = 0$ may be assumed since otherwise the problem does not have a feasible solution. Conditions (2.52) ensure that

- if $b(i) > 0$, then the total flow out of node i equals the total flow into node i plus the supply of i,
- if $b(i) < 0$, then the total flow into node i equals the total flow out of i plus the demand in i,
- if $b(i) = 0$, then the total flow into i equals the total flow out of i.

The coefficient matrix of the linear program is an $n \times m$-matrix in which the n rows correspond to the nodes $i \in V$ and the m columns correspond to the arcs $(i,j) \in A$. The column corresponding to the arc (i,j) has the entry $+1$ in row i and the entry -1 in row j. All other entries in this column are zero. Such a matrix is also called **node-arc incidence matrix** of the graph $G = (V,A)$.

A minimum cost flow problem with $b(i) = 0$ for all $i \in V$ is called a **minimum cost circulation problem**. Another important special case of the minimum cost flow problem is the **maximum flow problem** in which we have to send a

maximum amount of flow from a specified source node s to a specified sink node t. A third special case of the minimum cost flow problem is the **transportation problem** where the node set V is partitioned into two subsets V_1 and V_2 such that

- each node in V_1 is a supply node,
- each node in V_2 is a demand node, and
- for each arc $(i, j) \in A$ we have $i \in V_1$ and $j \in V_2$.

The minimum cost flow problem with integer arc capacities $u_{ij} < \infty$ and integer supply/demand values $b(i)$ has the important property that if an optimal solution exists, then also an optimal solution with integer-valued flows x_{ij} exists. Such an integer solution will be provided by algorithms, which are described later.

Due to this **integrality property** shortest path problems and the assignment problem can be formulated as special minimum cost flow problems:

- To find a **shortest s-t-path** in a network $G = (V, A)$, one has to find an optimal integer solution for a corresponding minimum cost flow problem in G with $b(s) := 1$, $b(t) := -1$, $b(i) := 0$ for all $i \neq s, t$, and $u_{ij} := 1$ for all $(i, j) \in A$. Due to the integrality property, all flows on the arcs are zero or one. Then in an optimal solution all arcs which carry one unit of flow constitute a shortest s-t-path. If we want to determine shortest paths from the source node s to every other node in the network, then in the minimum cost flow problem we set $b(s) := n - 1$, $b(i) := -1$ for all other nodes $i \neq s$, and $u_{ij} := n - 1$ for all $(i, j) \in A$. Then in an optimal integral solution unit flows from node s to every other node i are sent along shortest s-i-paths.
- The data of the **assignment problem** consists of two equally-sized sets V_1 and V_2, a collection of pairs $A \subseteq V_1 \times V_2$ representing possible assignments, and a cost c_{ij} associated with each arc $(i, j) \in A$. In the assignment problem we wish to pair each object in V_1 with exactly one object in V_2 at minimum possible total cost. Each assignment corresponds to a feasible integer solution of a corresponding transportation problem with $b(i) := 1$ for all $i \in V_1$, $b(i) := -1$ for all $i \in V_2$ and $u_{ij} := 1$ for all $(i, j) \in A$.

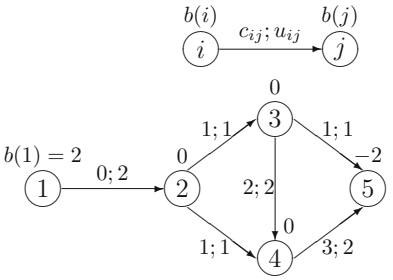
Before developing algorithms for the minimum cost flow problem we will discuss some general concepts, which are useful in connection with network flows.

2.4.2 The residual network and decomposition of flows

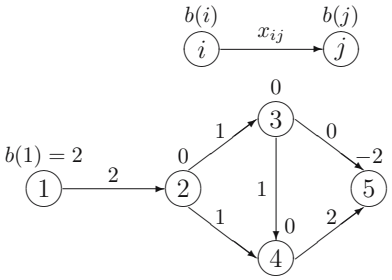
The network flow algorithms we will present are based on the idea of flow augmentation: Starting with some (not necessarily feasible) flow this flow is incrementally changed until an optimal flow is reached. These incremental flow

changes are guided by paths or cycles belonging to a so-called residual network $G(x)$ associated with the current flow x. Before giving a formal definition of $G(x)$ we will illustrate these ideas by an example.

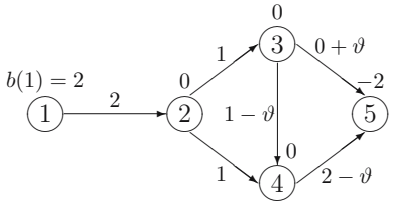
Example 2.11: Consider the network shown in Figure 2.12(a). A feasible flow x with $x_{12} = 2$, $x_{23} = 1$, $x_{24} = 1$, $x_{34} = 1$, $x_{35} = 0$, $x_{45} = 2$ is shown in Figure 2.12(b).



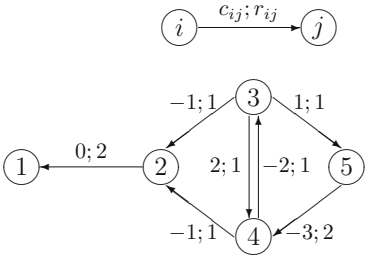
(a) Network $G = (V, A)$



(b) A feasible flow x



(c) An improved flow x^1



(d) The residual network $G(x)$

Figure 2.12: Improving a flow

This flow is not optimal. It can be improved by increasing the flow on the arc (3,5) by $\vartheta = 1$ and decreasing the flow on the arcs (3,4) and (4,5) by $\vartheta = 1$ (cf. Figure 2.12(c)). This changes the costs by $c_{35} - c_{34} - c_{45} = 1 - 2 - 3 = -4$, i.e. the costs are reduced by 4 units. □

For a given flow x the **residual network** $G(x)$ is a network with node set V and the following arcs:

- For each arc $(i, j) \in A$ with $x_{ij} < u_{ij}$ there is a **forward arc** (i, j) with **residual capacity** $r_{ij} := u_{ij} - x_{ij}$ and **residual cost** $c^r_{ij} := c_{ij}$.
- For each arc $(i, j) \in A$ with $x_{ij} > 0$ there is a **backward arc** (j, i) with **residual capacity** $r_{ji} := x_{ij}$ and residual cost $c^r_{ji} := -c_{ij}$.

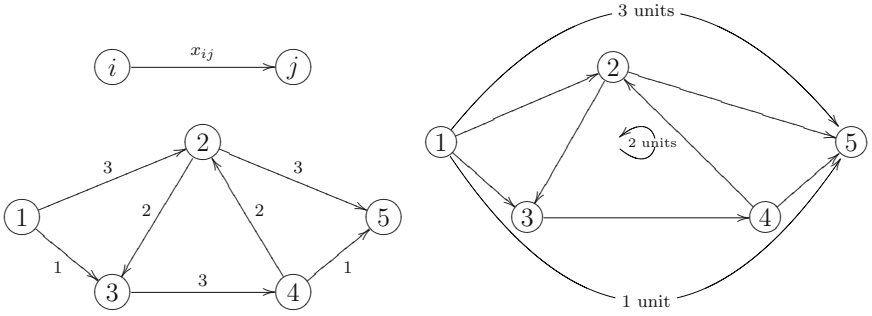
In connection with the maximum flow problem $G(x)$ is defined in the same way but the arcs have no costs. A path $P = (i_1, i_2, \ldots, i_k)$ in $G(x)$ is called an

augmenting path. If P is a cycle (i.e. $i_1 = i_k$), it is called an **augmenting cycle**. The value $\vartheta := \min\{r_{i_\nu,i_{\nu+1}} \mid \nu = 1,\dots,k-1\}$ is called **residual capacity** of the augmenting path P. If P is an augmenting path or cycle, in the original network G the flow x can be changed along P by

- increasing the flow in $(i,j) \in A$ by ϑ for all forward arcs (i,j) in P, and
- decreasing the flow in $(i,j) \in A$ by ϑ for all backward arcs (j,i) in P.

Such a flow change along P is called **ϑ-augmentation along P**. By such an augmentation the costs are changed by the amount $\vartheta \sum_{(i,j)\in P} c^r_{ij}$, where c^r_{ij} are the residual costs as defined above.

Coming back to Example 2.11, the residual network $G(x)$ for the flow x depicted in Figure 2.12(b) is shown in Figure 2.12(d). In this situation, $(3,5,4,3)$ is an augmenting cycle with residual capacity $\vartheta = \min\{1,2,1\} = 1$. If we perform a ϑ-augmentation along this cycle, the costs change by $\vartheta(c_{35} - c_{45} - c_{34}) = 1 \cdot (1-3-2) = -4$.



(a) Arc flows (b) Path and cycle flows

Figure 2.13: Decomposition of a flow into path and cycle flows

Next we prove an important structural property of a flow. It can be decomposed into flows in paths and cycles. In Figure 2.13(b) a decomposition of the flow shown in Figure 2.13(a) is illustrated.

In the network flow formulation (2.51) to (2.53) the variables x_{ij} are called **arc flows**. There is an alternative formulation in which the variables correspond to paths and cycles of the network. In the following we will describe this alternative formulation and show that both formulations are equivalent.

Denote by $\mathcal{P}$ and $\mathcal{Z}$ the set of all paths P and all cycles Z, respectively, of the network (V,A). The decision variables in the path and cycle flow formulation are variables $f(P)$, the flows on paths $P \in \mathcal{P}$, and variables $f(Z)$, the flows on cycles $Z \in \mathcal{Z}$. Note that every set of path and cycle flows uniquely determines

arc flows in a natural way: The flow x_{ij} on arc (i, j) equals the sum of flows $f(P)$ and $f(Z)$ for all paths P and cycles Z that contain this arc. If we set

$$\delta_{ij}(P) := \begin{cases} 1, & \text{if } (i,j) \text{ belongs to } P \\ 0, & \text{otherwise} \end{cases}$$

and define $\delta_{ij}(Z)$ in a similar way, then

$$x_{ij} = \sum_{P \in \mathcal{P}} \delta_{ij}(P) f(P) + \sum_{Z \in \mathcal{Z}} \delta_{ij}(Z) f(Z).$$

Thus each **path and cycle flow** $(f(P))_{P \in \mathcal{P}}$ and $(f(Z))_{Z \in \mathcal{Z}}$ determines arc flows in a unique way. In the following theorem we show that also the reverse is true.

Theorem 2.5 (Flow Decomposition Theorem): Every non-negative arc flow x satisfying conditions (2.52) can be represented as a path and cycle flow with the following properties:

(1) Every directed path with positive flow leads from a node i with $b(i) > 0$ to a node j with $b(j) < 0$.

(2) At most $n + m$ paths and cycles have nonzero flow and out of these, at most m cycles have nonzero flow.

Proof: We prove the theorem by constructing the decomposition. Suppose that i_1 is a node with $b(i_1) > 0$. Then some arc (i_1, i_2) carries a positive flow. If $b(i_2) < 0$, we stop. Otherwise constraint (2.52) for node i_2 implies that some arc (i_2, i_3) carries a positive flow. We repeat this argument until we reach a node i_k with $b(i_k) < 0$ or we revisit a previously examined node. Since the network constains n different nodes, one of these two cases occurs after at most n steps. In the first case we obtain an i_1-i_k-path P with $b(i_1) > 0$ and $b(i_k) < 0$. In the second case we obtain a cycle Z. In both cases P or Z contains only arcs with positive flow.

If we obtain a path P, we let $f(P) := \min\{b(i_1), -b(i_k), \min\{x_{ij} \mid (i,j) \in P\}\}$ and redefine $b(i_1) := b(i_1) - f(P)$, $b(i_k) := b(i_k) + f(P)$, and $x_{ij} := x_{ij} - f(P)$ for all $(i, j) \in P$. If we obtain a cycle Z, we let $f(Z) := \min\{x_{ij} \mid (i,j) \in Z\}$ and redefine $x_{ij} := x_{ij} - f(Z)$ for all $(i, j) \in Z$.

We repeat this process with the redefined problem until $b(i) = 0$ for all nodes i holds. Then we select some node with at least one outgoing arc with a positive flow as new starting node. In this case the procedure must find a directed cycle. We terminate the whole process when $x = 0$ holds for the redefined problem. At this stage the original flow must be equal to the sum of the flows on the paths and cycles identified by our method.

Observe that each time we identify a path, we reduce either a $|b(i)|$-value or the flow on some arc to zero. On the other hand, if we identify a cycle, we reduce the flow on some arc to zero. Consequently, the constructed path and cycle

representation of the given flow x contains at most $n + m$ directed paths and cycles, and at most m of these are directed cycles. □

By applying the same procedure to a circulation we get

Theorem 2.6 A circulation x can be represented as cycle flow along at most m directed cycles.

Similarly, the following theorem can be proved.

Theorem 2.7 (Augmenting Cycle Theorem): Let x and x^0 be two arbitrary feasible solutions of a network flow problem. Then x equals x^0 plus the flows on at most m directed cycles in the residual network $G(x^0)$. Furthermore, the cost of x is equal to the cost of x^0 plus the costs of the flows in $G(x^0)$ in these augmenting cycles.

Proof: Define $x^1 := x - x^0$. If $x^1_{ij} = x_{ij} - x^0_{ij} \geq 0$, then $0 \leq x^1_{ij} \leq x_{ij} \leq u_{ij}$ and we consider x^1_{ij} as a flow on the arc (i, j) in $G(x^0)$. On the other hand, if $x^1_{ij} = x_{ij} - x^0_{ij} < 0$, then $0 < -x^1_{ij} = -x_{ij} + x^0_{ij} \leq x^0_{ij}$ and we consider $-x^1_{ij}$ as a flow on the arc (j, i) in $G(x^0)$. Therefore, $x^1 = x - x^0$ can be considered as a feasible circulation in the residual network $G(x^0)$ which decomposes into at most m cycle flows. We have $x = x^0 + x^1$ and x^1 has the desired property. □

Example 2.12: In order to illustrate the augmenting cycle theorem consider the network $G = (V, A)$ shown in Figure 2.14(a). Two feasible flows x and x^0 in G are depicted in Figure 2.14(b) and (c). The difference flow $x^1 := x - x^0$ can be found in Figure 2.14(d), the residual network $G(x^0)$ in Figure 2.14(e). Finally, a circulation in $G(x^0)$ which corresponds to the flow x^1 is shown in Figure 2.14(f). This circulation can be decomposed into the 3 cycles $(1, 3, 5, 6, 4, 2, 1)$, $(2, 3, 5, 6, 4, 2)$ and $(4, 5, 6, 4)$ each carrying one unit of flow. □

2.4.3 The maximum flow problem

As before we consider a directed graph $G = (V, A)$ with arc capacities u_{ij} which are non-negative integers or equal to ∞. To define the maximum flow problem, we distinguish two special nodes in G: a **source node** s and a **sink node** t. We wish to find a maximum flow from s to t which satisfies the capacity constraints. If we again denote the flow on the arc $(i, j) \in A$ by x_{ij}, the problem has the formulation

$$\max \quad v \tag{2.54}$$

$$\text{s.t.} \quad \sum_{\{j|(i,j)\in A\}} x_{ij} - \sum_{\{j|(j,i)\in A\}} x_{ji} = \begin{cases} v & \text{for } i = s \\ 0 & \text{for } i \in V \backslash \{s, t\} \\ -v & \text{for } i = t \end{cases} \tag{2.55}$$

$$0 \leq x_{ij} \leq u_{ij} \qquad (i, j) \in A \tag{2.56}$$

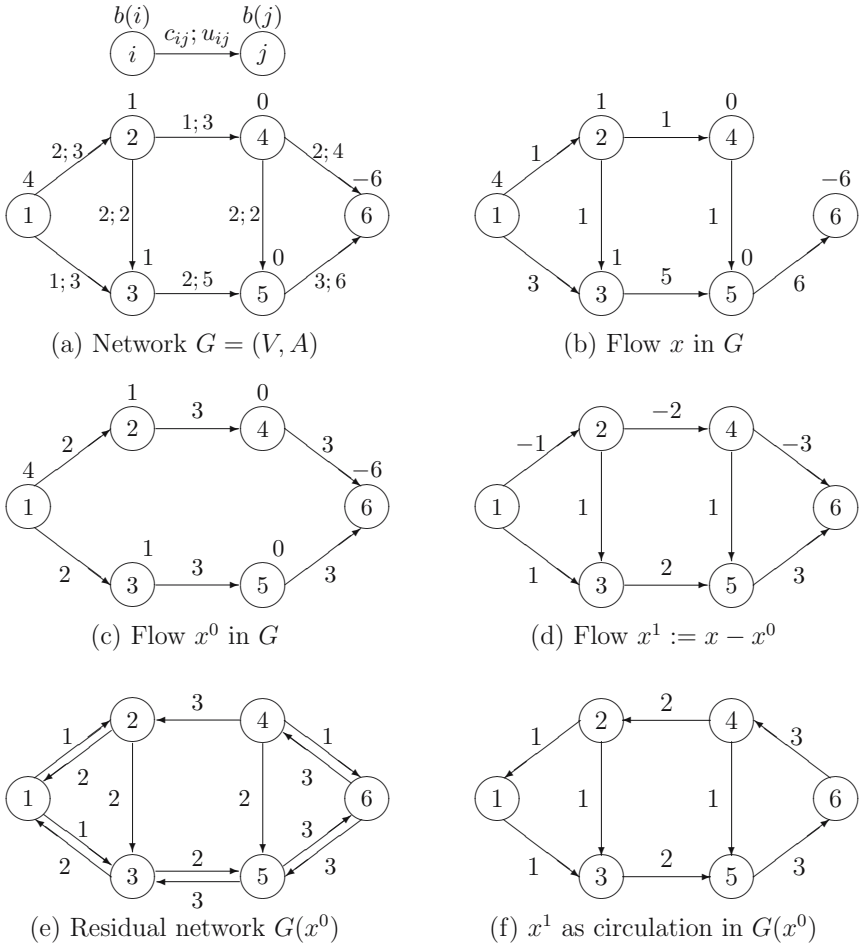


Figure 2.14: Illustrating the augmenting cycle theorem

A vector $x = (x_{ij})$ satisfying (2.55) and (2.56) is again called a **feasible flow** and the corresponding number v is called the **value of the flow** x.

Furthermore, we assume that the network does not contain a directed s-t-path on which all arcs have infinite capacity (otherwise an infinite amount of flow can be sent from s to t, i.e. the problem is unbounded).

Next we describe some applications of the maximum flow problem.

Application 2.1: Feasible flow problem

The feasible flow problem requires to find a feasible flow $x = (x_{ij})$ in a network $G = (V, A)$ satisfying the constraints (2.52) and (2.53). We can solve this problem by solving a maximum flow problem defined on an augmented network as follows. We add to the network a source node s and a sink node t. For each node $i \in V$ with $b(i) > 0$ we add an arc (s, i) with capacity $b(i)$, and for each

node $i \in V$ with $b(i) < 0$ we add an arc (i, t) with capacity $-b(i)$. Then we calculate a maximum flow from s to t in the augmented network. It is easy to see that a feasible flow in the original network exists if and only if the maximum flow x^* saturates all arcs (s, i) with $b(i) > 0$ (i.e. we have $x^*_{si} = b(i)$ for all $i \in V$ with $b(i) > 0$). □

Application 2.2: Scheduling on identical parallel machines

Consider the problem of scheduling n jobs $j = 1, \ldots, n$ on m identical parallel machines $M_1, \ldots, M_m$. Each job j has a processing time p_j, a release date r_j, and a deadline $d_j \geq r_j + p_j$. We assume that a machine can process only one job at a time and that each job can be processed by at most one machine at a time. However, we allow **preemptions**, i.e. we can interrupt a job and process it (possibly on different machines) later. The scheduling problem is to determine a feasible schedule in which all jobs do not start earlier than their release dates and do not complete later than their deadlines or to show that no such schedule exists.

To formulate the feasibility problem as a maximum flow problem let $t_1 < t_2 < \ldots < t_\tau$ be the ordered sequence of all different r_i- and d_i-values. Furthermore, let $I_\nu = [t_\nu, t_{\nu+1}]$ for $\nu = 1, \ldots, \tau - 1$ be the corresponding intervals with lengths $T_\nu := t_{\nu+1} - t_\nu$. We associate with this scheduling problem a network with the following nodes:

- job nodes $j = 1, \ldots, n$,
- interval nodes $I_1, \ldots, I_{\tau-1}$, and
- a source node s and a sink node t.

For each job j we introduce an arc (s, j) with capacity $u_{sj} := p_j$ and for each interval I_ν an arc (I_ν, t) with capacity $u_{I_\nu, t} := mT_\nu$. Additionally, we have an arc (j, I_ν) with capacity T_ν if parts of job j can be scheduled in the interval I_ν (i.e. if $I_\nu \subseteq [r_j, d_j]$).

Clearly, if there exists a feasible schedule and we denote by $x_{j\nu} \leq T_\nu$ the total processing time of job j in the interval I_ν, then the values $x_{j\nu}$ together with

$$x_{sj} := p_j = \sum_{\nu=1}^{\tau-1} x_{j\nu} \quad \text{for } j = 1, \ldots, n$$

and

$$x_{\nu t} := \sum_{j=1}^{n} x_{j\nu} \leq mT_\nu \quad \text{for } \nu = 1, \ldots, \tau - 1$$

define a maximum flow with value $P := \sum_{j=1}^{n} p_j$. Conversely, it is not difficult to see that if we have a maximum flow with value P, then the flow values $x_{j\nu}$ corresponding to the arcs (j, I_ν) define a solution of the scheduling problem. □

2.4.4 Flows and cuts

The dual linear program of the maximum flow problem (2.54) to (2.56) is called **minimum cut problem**. Rather than deriving this dual problem, we give a direct description of the minimum cut problem. A **cut** is a partition of the node set V into two subsets X and $\overline{X} = V \backslash X$. A cut is an **s-t-cut** if $s \in X$ and $t \in \overline{X}$. All arcs (i, j) with $i \in X$ and $j \in \overline{X}$ are called **forward arcs** and all arcs (j, i) with $j \in \overline{X}$ and $i \in X$ are called **backward arcs**. A cut is denoted by $[X, \overline{X}]$, the sets of forward and backward arcs are denoted by $(X, \overline{X})$ and $(\overline{X}, X)$, respectively.

We define the **capacity** $u[X, \overline{X}]$ of an s-t-cut $[X, \overline{X}]$ as the sum of the capacities of all forward arcs in the cut, i.e.

$$u[\overline{X}, \overline{X}] = \sum_{(i,j)\in(X,\overline{X})} u_{ij}.$$

A **minimum cut** is an s-t cut whose capacity is minimum among all s-t-cuts. We refer to the problem of finding a minimum cut as **minimum cut problem**.

Let x be a flow in the network and let $[X, \overline{X}]$ be an s-t-cut. By adding all equations (2.55) for $i \in X$ we get

$$v = \sum_{i\in X} \left(\sum_{\{j|(i,j)\in A\}} x_{ij} - \sum_{\{j|(j,i)\in A\}} x_{ji} \right) \tag{2.57}$$

which can be simplified to

$$v = \sum_{(i,j)\in(X,\overline{X})} x_{ij} - \sum_{(j,i)\in(\overline{X},X)} x_{ji} \tag{2.58}$$

because whenever both nodes p and q belong to X and $(p, q) \in A$, the variable x_{pq} in the first term within the brackets of (2.57) (for node $i = p$) cancels the variable $-x_{pq}$ in the second term within the brackets (for node $j = q$). Moreover, if both nodes p and q belong to $\overline{X}$, then x_{pq} does not appear in the expression. Due to $0 \leq x_{ij} \leq u_{ij}$ for all arcs $(i, j) \in A$ condition (2.58) implies

$$v \leq \sum_{(i,j)\in(X,\overline{X})} u_{ij} = u[X, \overline{X}]. \tag{2.59}$$

Therefore we have the important

Property 2.5 The value of any flow is not larger than the capacity of any s-t cut in the network.

A consequence of Property 2.5 is that if we discover a flow x whose value v equals the capacity of some s-t-cut $[X, \overline{X}]$, then x is a maximum flow and the cut $[X, \overline{X}]$ is a minimum cut.

The so-called augmenting path algorithm, which will be derived in the next subsection, provides such flows and cuts. However, before moving to this subsection we will present an application for the minimum cut problem.

Application 2.3: Distributed computing on a two-processor computer

Consider a computer system with two processors (which do not have to be identical) on which a large program should be executed. The program contains n modules $i = 1, \ldots, n$ that interact with each other during the execution of the program. Let α_i and β_i denote the cost for computing module i on processors 1 and 2, respectively. Assigning different modules to different processors incurs additional costs due to interprocessor communication. Let c_{ij} denote the interprocessor communication costs if modules i and j are assigned to different processors. We wish to allocate modules of the program to the two processors such that the total cost of processing and interprocessor communication is minimized.

This problem can be formulated as a minimum cut problem as follows. We define a source node s representing processor 1, a sink node t representing processor 2, and nodes $i = 1, \ldots, n$ for the modules of the program. For each node $i \neq s, t$, we include an arc (s, i) with capacity $u_{si} := \beta_i$ and an arc (i, t) with capacity $u_{it} := \alpha_i$. Futhermore, if module i interacts with module j during program execution, we include arcs (i, j) and (j, i) with capacities $u_{ij} = u_{ji} := c_{ij}$.

As an example consider an instance with 4 modules, interprocessor communication costs $c_{12} = 5, c_{23} = c_{24} = 2, c_{34} = 3, c_{ij} = 0$ otherwise, and the following processing costs:

i	1	2	3	4
α_i	6	5	10	4
β_i	4	11	3	4

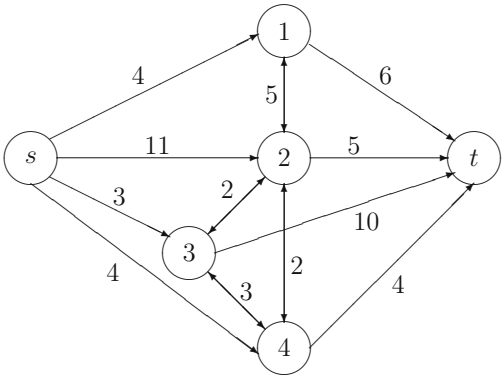
In Figure 2.15(a) the corresponding network is shown, all arcs are weighted with their capacities.

Let A_1 and A_2 be assignments of modules to processors 1 and 2, respectively. Then $[\{s\} \cup A_1, \{t\} \cup A_2]$ defines an s-t-cut and its value is equal to the assignment costs

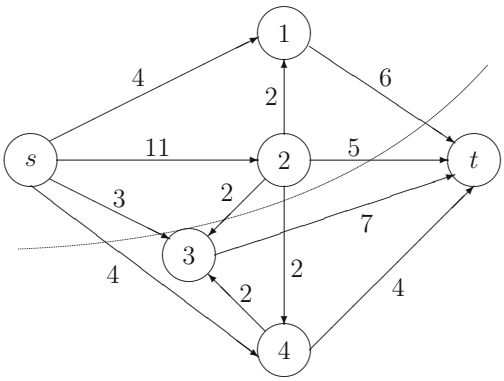
$$\sum_{i \in A_1} \alpha_i + \sum_{i \in A_2} \beta_i + \sum_{(i,j) \in A_1 \times A_2} c_{ij}.$$

Conversely, an s-t cut defines an assignment and the capacity of this cut is equal to the assignment costs.

For the example network in Figure 2.15(a) a corresponding flow is shown in (b). It has the value $4 + 11 + 3 + 4 = 22$, which is equal to the capacity $4+3+2+2+5+6 = 22$ of the s-t-cut $[\{s, 1, 2\}, \{t, 3, 4\}]$. Thus, this cut defines an optimal solution for the module assignment problem, namely to run modules 1 and 2 on processor 1 and the other modules on processor 2. □



(a) Network with arc capacities



(b) Maximum flow and minimum cut

Figure 2.15: Network for the distributed computing model

2.4.5 Algorithms for the maximum flow problem

In this section we will describe algorithms which solve the maximum flow problem (2.54) to (2.56). The simplest and most intuitive algorithm for solving the maximum flow problem is an algorithm known as augmenting path algorithm.

Recall that an **augmenting path** is a directed path from the source to the sink in the residual network and the **residual capacity** of an augmenting path is the minimum residual capacity of any arc on the path. By definition, this capacity ϑ is always positive. Consequently, whenever the network contains an augmenting path we can send additional flow from the source to the sink. The algorithm proceeds by identifying augmenting paths and augmenting flows on these paths until the network contains no such path anymore. The algorithm usually starts with the zero flow. A description of the algorithm is shown in Figure 2.16.

```
Algorithm Augmenting Path
1.  x := 0
2.  WHILE G(x) contains a directed s-t-path DO
3.     Identify an augmenting path P from s to t;
4.     ϑ := min{r_ij | (i,j) ∈ P};
5.     Augment ϑ units of flow along P and update G(x);
6.  ENDWHILE
```

Figure 2.16: Generic augmenting path algorithm

Example 2.13: The generic augmenting path algorithm is illustrated with the example in Figure 2.17, where we have the source $s = 1$ and the sink $t = 4$. In Figure 2.17(a) we start with the zero flow $x^0 = 0$. The corresponding residual network $G(x^0)$ is shown in Figure 2.17(b) where we choose the path $(1, 2, 3, 4)$ with capacity $\vartheta = \min\{2, 3, 5\} = 2$ as augmenting path. The augmented flow x^1 is shown in Figure 2.17(c). Then $(1, 3, 4)$ is the next augmenting path, etc. We end up with the flow x^3 in Figure 2.17(g). In the corresponding residual network $G(x^3)$ there is no path from the source node 1 to the sink node 4. The value of the flow is 6 which is equal to the capacity of the cut $[\{1\}, \{2, 3, 4\}]$. Thus, we have identified a maximum flow and a minimum cut. □

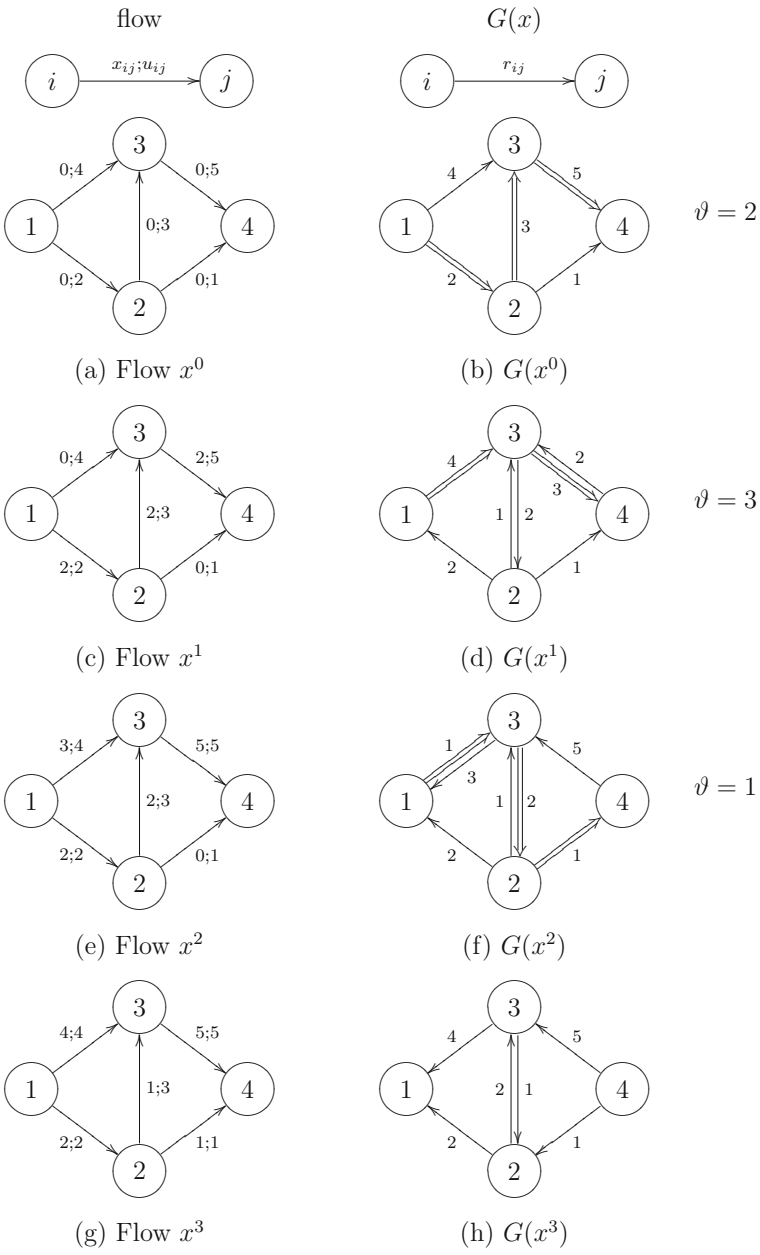
In the generic augmenting path algorithm we have not specified how an augmenting path (if one exists) is determined in Step 3. In the following we will describe the so-called **labeling algorithm** which systematically searches for an s-t path in $G(x)$ by labeling all nodes which can be reached from s by a path in $G(x)$. The algorithm iteratively selects a node i from a list of labeled nodes, removes it from the list and adds all unlabeled neighbors j with (i, j) in $G(x)$ to the list.

If the sink t becomes labeled, an s-t path has been found and an augmentation along this path is performed. Then all nodes are again unlabeled and the search process is restarted. The algorithm stops when all labeled nodes have been scanned and the sink t remains unlabeled. In this situation, no s-t-path in the residual network exists.

A detailed description of the labeling algorithm can be found in Figure 2.18.

In this algorithm the predecessor array $pred(j)$ is used to reconstruct an augmenting path in the same way as it was used in connection with shorthest path algorithms (cf. Section 2.2). The procedure `AUGMENT` does the augmentation and updates the residual network.

It remains to prove the correctness of the algorithm and to determine its complexity. Assume that all capacities u_{ij} are finite integers (otherwise the values $u_{ij} = \infty$ may be replaced by a sufficiently large integer, e.g. by $\sum_{\{(i,j)\in A | u_{ij} < \infty\}} u_{ij}$) and let $U := \max_{(i,j)\in A} u_{ij}$.



(a) Flow x^0 (b) $G(x^0)$

(c) Flow x^1 (d) $G(x^1)$

(e) Flow x^2 (f) $G(x^2)$

(g) Flow x^3 (h) $G(x^3)$

Figure 2.17: Illustrating the generic augmenting path algorithm

```
Algorithm Labeling
 1.  x := 0; label node t;
 2.  WHILE t is labeled DO
 3.     Unlabel all nodes;
 4.     FOR ALL j ∈ V DO pred(j) := 0;
 5.     Label node s; List := {s};
 6.     WHILE List ≠ ∅ AND t is unlabeled DO
 7.        Remove a node i from List;
 8.        FOR ALL arcs (i,j) in G(x) DO
 9.            IF j is unlabeled THEN
10.               pred(j) := i;
11.               Label node j;
12.               Add j to List;
13.            ENDIF
14.     ENDWHILE
15.     IF t is labeled THEN Augment;
16.  ENDWHILE

Procedure Augment
1.  Use the predecessor labels to trace back from t to s
    in order to obtain an augmenting path P;
2.  ϑ := min{r_ij | (i,j) ∈ P};
3.  Augment ϑ units of flow along P and update the
    residual capacities;
```

Figure 2.18: The labeling algorithm

Theorem 2.8 The labeling algorithm solves the maximum flow problem in $O(nmU)$ time.

Proof: Since the value nU is an upper bound for the capacity of the s-t-cut $(\{s\}, V\backslash\{s\})$, due to Property 2.5 the maximum flow value is also bounded by nU. Each augmentation increases the flow value v by at least one unit, i.e. the algorithm stops after at most nU augmentations. Since the search method examines any arc at most once, each augmentation requires $O(m)$ time. Thus, the overall complexity is $O(nmU)$.

When the algorithm stops, let X be the set of labeled nodes and $\overline{X} = V\backslash X$ be the set of unlabeled nodes. Clearly, $s \in X$ and $t \in \overline{X}$. Since the algorithm cannot label any node in $\overline{X}$ from any node in X, we have $r_{ij} = 0$ for each (i,j) in $G(x)$ with $i \in X$ and $j \in \overline{X}$. For forward arcs $(i,j) \in (X, \overline{X})$ the condition $r_{ij} = u_{ij} - x_{ij} = 0$ implies $x_{ij} = u_{ij}$. For backward arcs $(i,j) \in (X, \overline{X})$ the condition $r_{ij} = x_{ji} = 0$ implies $x_{ji} = 0$.

Thus, with (2.58) we get

$$v = \sum_{(i,j)\in(X,\overline{X})} x_{ij} - \sum_{(j,i)\in(\overline{X},X)} x_{ji} = \sum_{(i,j)\in(X,\overline{X})} u_{ij} = u[X,\overline{X}].$$

But then Property 2.5 implies that x is a maximum flow and $[X,\overline{X}]$ is a minimum cut. This conclusion establishes the correctness of the labeling algorithm. □

As a by-product we have proven the

Theorem 2.9 (Max-Flow Min-Cut Theorem): The maximum value of a flow from a source node s to a sink node t in a capacitated network equals the minimum capacity among all s-t-cuts.

Theorem 2.10 (Integrality Property): If all arc capacities are integer, the maximum flow problem has an integer maximum flow.

Proof: If all capacities are integers, then starting with the zero flow $x = 0$ all ϑ-values and augmented flows are integer as well. Thus, the labeling algorithm provides an optimal flow which is integer. □

The presented labeling algorithm is only a pseudo-polynomial algorithm since its complexity is proportional to U and there are examples with $U = 2^n$, where the algorithm performs 2^n augmentations. However, if special implementations are used, such a large number of augmentations can be avoided. One possibility is to choose in each step a shortest augmenting path, i.e. among all possible augmenting paths one with a smallest number of arcs is determined. This so-called **successive shortest path labeling algorithm** runs in $O(n^2m)$ time. Another possibility to avoid a large number of augmentations is to augment only on paths with a sufficiently large residual capacity. This so-called **capacity scaling algorithm** runs in $O(nm \log U)$ time.

Besides augmenting path algorithms some other maximum flow algorithms with low complexities exist (cf. Figure 2.19). They are based on the concept of so-called **preflows** and **pushes**.

algorithm	complexity
generic preflow-push algorithm	$O(n^2m)$
FIFO preflow-push algorithm	$O(nm \log(n^2/m))$
highest-label preflow-push algorithm	$O(n^2\sqrt{m})$

Figure 2.19: Preflow-push algorithms for the maximum flow problem

2.4.6 Algorithms for the minimum cost flow problem

In this section we will describe algorithms which solve the minimum cost flow problem (2.51) to (2.53). To derive bounds for running times of these algorithms let C denote the maximum of all $|c_{ij}|$-values and let U denote the largest magnitude of all $b(i)$-and finite u_{ij}-values.

Additionally, we assume that the supplies and the demands of the nodes satisfy the condition $\sum_{i \in V} b(i) = 0$ and that the minimum cost flow problem has a feasible solution (which can be checked as indicated in Example 2.1 by applying a maximum flow algorithm).

Recall that, given a feasible flow x, the residual network $G(x)$ is a network with node set V and the following arcs:

- arcs (i, j) with cost c_{ij} and residual capacity $r_{ij} = u_{ij} - x_{ij}$ if $(i, j) \in A$ and $x_{ij} < u_{ij}$, and
- arcs (j, i) with cost $-c_{ij}$ and residual capacity $r_{ji} = x_{ij}$ if $(i, j) \in A$ and $x_{ij} > 0$.

Next we will formulate and prove an optimality condition for the minimum cost flow problem.

Theorem 2.11 (Negative Cycle Optimality Condition): A feasible flow x^* is an optimal solution for the minimum cost flow problem if and only if the residual network $G(x^*)$ contains no cycle with negative costs.

Proof: Suppose that x is a feasible flow and that $G(x)$ contains a negative cycle. Then x cannot be an optimal flow, since by augmenting the flow along the cycle by one unit we can improve the objective function value. Thus, if x^* is an optimal solution, then $G(x^*)$ cannot contain a negative cycle.

Conversely, let x^* be a feasible and x^0 be an optimal flow. By Theorem 2.7 the difference $x^0 - x^*$ is a circulation in $G(x^*)$ and this circulation can be decomposed into at most m augmenting cycles with respect to the flow x^* and the sum of the costs of flows on these cycles equals $cx^0 - cx^*$. If the lengths of all cycles in $G(x^*)$ are non-negative, $cx^0 - cx^* \geq 0$ or equivalently $cx^0 \geq cx^*$ holds, which implies that x^* must be optimal, too. □

The negative cycle optimality condition suggests a simple algorithm for the minimum cost flow problem, which is called **cycle-canceling algorithm**. This algorithm first establishes a feasible flow by solving a corresponding maximum flow problem. Then it iteratively finds negative cost cycles in the residual network and augments the flow on these cycles. The algorithm terminates when the residual network contains no negative cycle anymore. A general description of this algorithm can be found in Figure 2.20.

A by-product of the cycle-canceling algorithm is the following result:

```
Algorithm Cycle Canceling
1.  Establish a feasible flow x in the network;
2.  WHILE G(x) contains a negative cycle DO
3.     Calculate a negative cycle Z in G(x);
4.     ϑ := min{r_ij | (i,j) ∈ Z};
5.     Augment ϑ units of flow in Z and update G(x);
6.  ENDWHILE
```

Figure 2.20: The cycle-canceling algorithm

Theorem 2.12 (Integrality Property): If all capacities u_{ij} and the supplies/demands $b(i)$ of the nodes are integers, the minimum cost flow problem always has an integer minimum cost flow.

Proof: If all capacities are integer, by Theorem 2.10 we can provide an integer feasible flow x if such a flow exists. Furthermore, all residual capacities in $G(x)$ are integer. Thus, the flow augmented by the cycle-canceling algorithm in each iteration is integer as well. □

Let us now calculate a bound for the number of iterations that the algorithm performs. Since $-C \leq c_{ij} \leq C$ and $u_{ij} \leq U$ for all $(i,j) \in A$, the initial flow costs are bounded from above by mCU and the costs of each flow are bounded from below by $-mCU$. Each iteration of the cycle-canceling algorithm changes the objective function value by an amount $\vartheta \sum_{(i,j)\in Z} c_{ij}$, which is strictly negative. If we assume that all numbers of the problem are integral, the algorithm terminates within $O(mCU)$ iterations and runs in $O(nm^2CU)$ time if we use an $O(nm)$ label-correcting algorithm to detect a negative cycle.

In order to get polynomial-time algorithms for the minimum cost flow problem, more sophisticated ideas have to be used. In Figure 2.21 some of these algorithms and their complexities can be found.

algorithm	complexity
minimum mean cycle-canceling algorithm	$O(n^2m^3 \log n)$
capacity scaling algorithm	$O((m \log U)(m + n \log n))$
cost scaling algorithm	$O(n^3 \log(nC))$

Figure 2.21: Polynomial algorithms for the minimum cost flow problem

2.5 Branch-and-Bound Algorithms

Branch-and-bound algorithms are enumerative procedures for solving combinatorial optimization problems exactly. In this section we consider maximization problems, i.e. we have to find a solution

$$s^* \in \mathcal{S} \text{ with } c(s^*) \geq c(s) \text{ for all } s \in \mathcal{S}, \tag{2.60}$$

where $\mathcal{S}$ is a finite set of feasible solutions and c is a real-valued cost function.

2.5.1 Basic concepts

For a given objective function c problem (2.60) can be represented by the solution set $\mathcal{S}$. A subproblem of (2.60) is defined by a subset $\mathcal{S}' \subseteq \mathcal{S}$. The main components of a branch-and-bound algorithm can be described as follows.

- **Branching:** $\mathcal{S}$ is replaced by subproblems $\mathcal{S}_i \subseteq \mathcal{S}$ $(i = 1, \ldots, r)$ such that $\bigcup_{i=1}^{r} \mathcal{S}_i = \mathcal{S}$. This process is called **branching**. Branching is a recursive process, i.e. each $\mathcal{S}_i$ is the basis of another branching. The whole branching process is represented by a **branching tree**. $\mathcal{S}$ is the root of the branching tree, $\mathcal{S}_i$ $(i = 1, \ldots, r)$ are the children of $\mathcal{S}$, etc.

- **Upper bounding:** An algorithm is available for calculating an upper bound for the objective function values of all feasible solutions of a subproblem.

- **Lower bounding:** We calculate a lower bound L for problem (2.60). The objective value of any feasible solution will provide such a lower bound. If the upper bound of a subproblem is smaller than or equal to L, then this subproblem cannot provide a better solution for (2.60). Thus, we need not continue to branch from the corresponding node in the branching tree. To stop the branching process in many nodes of the branching tree, the bound L should be as large as possible. Therefore, at the beginning of the branch-and-bound algorithm some heuristic is applied to find a good feasible solution with large value L. After many recursive branchings we may reach a situation in which the subproblem has only one feasible solution. Then the upper bound UB for the subproblem is equal to the objective value of this solution and we replace L by UB if $UB > L$.

A generic recursive branch-and-bound algorithm for a maximization problem is illustrated in Figure 2.22.

Note, that for minimization problems the roles of upper- and lower bounds have to be interchanged. Every feasible solution provides an upper bound and if the lower bound for a subproblem is greater or equal to the upper bound, then

```
Algorithm Branch-and-Bound
 1. Calculate a starting solution s ∈ S;
 2. L := c(s);
 3. B&B(S);

Procedure B&B (S)
 1. Partition S into subsets S_1, ..., S_r with ⋃_{i=1}^{r} S_i = S;
 2. FOR ALL subsets S_i DO
 3.     Calculate an upper bound UB_i for S_i;
 4.     IF UB_i > L THEN
 5.        IF S_i contains only one solution s_i THEN
 6.           L := max{L, c(s_i)};
 7.        ELSE
 8.           B&B(S_i);
 9. ENDFOR
```

Figure 2.22: A generic recursive branch-and-bound algorithm

no feasible solution of the subproblem improves the current best solution. The corresponding subtree can be ignored.

By this informal description we have sketched the main concepts for branch-and-bound algorithms. Other shortcuts of the enumeration process may be possible by exploiting further problem-dependent dominance properties. This will be illustrated in connection with branch-and-bound algorithms for specific optimization problems later.

2.5.2 The knapsack problem

In this subsection we will present a branch-and-bound algorithm for the **integer knapsack problem**. In this problem we are given a knapsack with weight capacity b and n items $j = 1, \ldots, n$ with utility values c_j and weights a_j. For $j = 1, \ldots, n$ let x_j denote the number of items of type j which are packed into the knapsack. We have to calculate numbers x_j such that the total weight of the knapsack is bounded by b and the value of its content is maximized.

The knapsack problem can be formulated as the following integer linear program with one constraint:

$$\begin{array}{lrcl} \max & \sum_{j=1}^{n} c_j x_j & & \\ \text{s.t.} & \sum_{j=1}^{n} a_j x_j & \leq & b \\ & x_j & \geq & 0 \text{ and integer } (j = 1, \ldots, n) \end{array} \tag{2.61}$$

If all variables x_j are restricted to the set $\{0,1\}$, we have a so-called **binary knapsack problem**. In this case, for each item j we have only to decide whether j is packed into the knapsack or not.

We assume that all weights a_j are positive integer numbers. Furthermore, all utility values c_j can be assumed to be positive integers as well (if $c_j \leq 0$ holds, then $x_j = 0$ in an optimal solution and item j can be ignored).

Additionally, assume that the items are numbered according to

$$c_1/a_1 \geq c_2/a_2 \geq \ldots \geq c_n/a_n. \tag{2.62}$$

Then every optimal solution satisfies the condition

$$0 \leq b - \sum_{j=1}^{n} a_j x_j < a_n, \tag{2.63}$$

since otherwise the value could be increased by including one more item with weight a_n.

The first branching is done by fixing x_1 to one of the possible values $0, 1, ..., \left\lfloor \frac{b}{a_1} \right\rfloor$. Then for fixed value $\overline{x}_1$ we branch by fixing x_2 to one of the possible values $0, 1, \ldots, \left\lfloor \frac{b-a_1\overline{x}_1}{a_2} \right\rfloor$, etc.

Thus, the nodes of the enumeration tree correspond to a set $\overline{x}_1, \ldots, \overline{x}_i$ ($i = 1, \ldots, n$) of fixed variables. Solutions are represented by the leaves of this tree (where all variables are fixed).

Note, that for fixed non-negative values $\overline{x}_1, \ldots, \overline{x}_{n-1}$ due to (2.63) the value of x_n is uniquely determined by

$$\overline{x}_n = \left\lfloor \left(b - \sum_{j=1}^{n-1} a_j \overline{x}_j \right) / a_n \right\rfloor. \tag{2.64}$$

The branch-and-bound-algorithm systematically enumerates the leaves of the enumeration tree looking for the best feasible solution. The first feasible solution is obtained by setting x_1 to the largest possible value and after fixing x_1 by setting x_2 to the largest possible value, etc. This means we calculate for $j = 1, \ldots, n$ the values

$$x_j = \left\lfloor \left(b - \sum_{i=1}^{j-1} a_i x_i \right) / a_j \right\rfloor. \tag{2.65}$$

This solution $(x_1, \ldots, x_n)$ provides a first lower bound $L = \sum_{j=1}^{n} c_j x_j$. Whenever we reach a new feasible solution with an objective value greater than L, we replace L by this greater value. From each leaf that has been just examined, we backtrack towards the root until the largest $k < n$ with $x_k > 0$ is found. Then we replace x_k by $x_k - 1$ and calculate an upper bound UB for the node ν of

the enumeration tree defined by $x_1, \ldots, x_{k-1}$ and the new x_k-value. If $UB \leq L$ holds, the subtree rooted in ν can be ignored because it provides no solution better than L. Therefore we continue to backtrack by looking for the largest $k' < k$ with $x_{k'} > 0$, etc. If $UB \geq L + 1$, then $(x_1, \ldots, x_k)$ is extended to the next feasible solution by recursively evaluating (2.65) for $j = k + 1, \ldots, n$.

It remains to show how to find an upper bound UB for all feasible solutions $\overline{x}$ with

$$\overline{x}_i = \begin{cases} x_i & \text{for } i = 1, \ldots, k-1 \\ x_i - 1 & \text{for } i = k. \end{cases}$$

We have $\frac{c_i}{a_i} \leq \frac{c_{k+1}}{a_{k+1}}$ or equivalently $c_i \leq \frac{c_{k+1}}{a_{k+1}} a_i$ for $i = k + 1, \ldots, n$ and therefore

$$\sum_{i=k+1}^{n} c_i \overline{x}_i \leq \frac{c_{k+1}}{a_{k+1}} \sum_{i=k+1}^{n} a_i \overline{x}_i. \tag{2.66}$$

Furthermore,

$$\sum_{i=1}^{n} a_i \overline{x}_i \leq b \text{ implies } \sum_{i=k+1}^{n} a_i \overline{x}_i \leq b - \sum_{i=1}^{k} a_i \overline{x}_i. \tag{2.67}$$

From (2.66) and (2.67) we derive

$$\sum_{i=1}^{n} c_i \overline{x}_i \leq \sum_{i=1}^{k} c_i \overline{x}_i + \frac{c_{k+1}}{a_{k+1}} \left(b - \sum_{i=1}^{k} a_i \overline{x}_i \right).$$

Thus, if

$$UB := \sum_{i=1}^{k} c_i \overline{x}_i + \frac{c_{k+1}}{a_{k+1}} \left(b - \sum_{i=1}^{k} a_i \overline{x}_i \right) < L + 1 \tag{2.68}$$

holds, then we can prune off the subtree defined by $\overline{x}_1, \ldots, \overline{x}_k$. In (2.68) we use the fact that all c_j are integers and thus the optimal objective function value is integer as well.

Note that after pruning off the subtree defined by $\overline{x}_1, \ldots, \overline{x}_k$, we can also immediately prune off the subtree defined by $\overline{x}_1, \ldots, \overline{x}_k - 1$. This is due to the fact that in this case the corresponding inequality

$$\begin{aligned} &\sum_{i=1}^{k-1} c_i \overline{x}_i + c_k(\overline{x}_k - 1) + \frac{c_{k+1}}{a_{k+1}} \left(b - \sum_{i=1}^{k-1} a_i \overline{x}_i - a_k(\overline{x}_k - 1) \right) \\ &\quad = UB - c_k + \frac{c_{k+1}}{a_{k+1}} a_k \leq UB < L + 1 \end{aligned} \tag{2.69}$$

due to $\frac{c_k}{a_k} \geq \frac{c_{k+1}}{a_{k+1}}$ is already implied by (2.68).

In Figure 2.23 a detailed description of a branch-and-bound algorithm for the knapsack problem is given. It calculates an optimal solution x^* for the problem using the bounding rule (2.68).

```
Algorithm Branch-and-Bound Knapsack
 1.  L := 0; k := 0;
 2.  REPEAT
 3.      FOR j := k + 1 TO n DO
 4.          x_j := ⌊(b − Σ_{i=1}^{j−1} a_i x_i)/a_j⌋;
 5.      IF Σ_{i=1}^{n} c_i x_i > L THEN
 6.          L := Σ_{i=1}^{n} c_i x_i; x* := x
 7.      ENDIF;
 8.      l := n − 1;
 9.      WHILE l > 0 DO
10.          Determine the largest k ≤ l with x_k > 0;
11.          x_k := x_k − 1;
12.          IF Σ_{i=1}^{k} c_i x_i + (c_{k+1}/a_{k+1}) (b − Σ_{i=1}^{k} a_i x_i) < L + 1 THEN
13.              l := k − 1;
14.      ENDWHILE
15.  UNTIL l = 0;
```

Figure 2.23: A branch-and-bound algorithm for the knapsack problem

Example 2.14: Consider the knapsack problem instance

$$\begin{array}{lrcrcrcrcl} \max & 4x_1 & + & 5x_2 & + & 5x_3 & + & 2x_4 & & \\ \text{s.t.} & 33x_1 & + & 49x_2 & + & 51x_3 & + & 22x_4 & \leq & 120 \\ & & & & & & & x_1, x_2, x_3, x_4 & \geq & 0 \quad \text{and integer} \end{array}$$

Then the branch-and-bound algorithm produces the enumeration tree shown in Figure 2.24.

The starting solution is given by

$$x_1 = \lfloor 120/33 \rfloor = 3, \qquad x_2 = \lfloor (120 - 99)/49 \rfloor = 0,$$
$$x_3 = \lfloor (120 - 99)/51 \rfloor = 0, \qquad x_4 = \lfloor (120 - 99)/22 \rfloor = 0.$$

It provides the lower bound $L = 4 \cdot 3 = 12$ and $x^* = (3, 0, 0, 0)$. Next we decrease x_1 by 1. With $k = 1$ and $\overline{x}_1 = 2$ the condition in Step 12 of the algorithm is not fulfilled:

$$4 \cdot 2 + \frac{5}{49}(120 - 33 \cdot 2) = 13.5102 \geq 13.5 \geq L + 1 = 13,$$

i.e. in Step 4 the next solution is calculated by

$$x_2 = \lfloor (120-66)/49 \rfloor = 1, x_3 = \lfloor (120-115)/51 \rfloor = 0, x_4 = \lfloor (120-115)/22 \rfloor = 0.$$

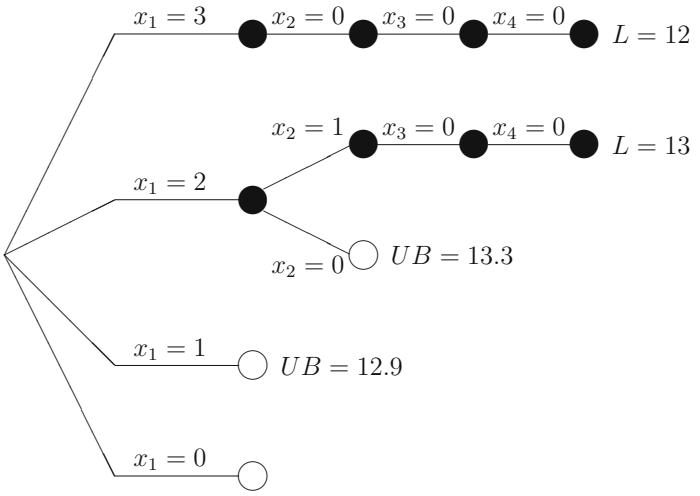


Figure 2.24: Enumeration tree for the knapsack problem

Since the new lower bound satisfies $L = 4 \cdot 2 + 5 \cdot 1 = 13 > 12$, we replace x^* by $x^* = (2, 1, 0, 0)$. After decreasing x_2 from 1 to 0, with $k = 2$, $\overline{x}_1 = 2$, $\overline{x}_2 = 0$ we have

$$4 \cdot 2 + \frac{5}{51}(120 - 66) = 13.2941 \leq 13.3 < L + 1 = 14,$$

i.e. the subtree corresponding to $\overline{x}_1 = 2$, $\overline{x}_2 = 0$ can be pruned off.

After decreasing x_1 from 2 to 1, with $k = 1$, $\overline{x}_1 = 1$ we get

$$4 \cdot 1 + \frac{5}{49}(120 - 33) = 12.8776 \leq 12.9 < L + 1 = 14,$$

i.e. the subtree corresponding to $\overline{x}_1 = 1$ can also be pruned off. Due to (2.69) also all solutions with $\overline{x}_1 = 0$ can immediately be eliminated. After decreasing the variable l to 0 in Step 13 the algorithm is stopped with the optimal solution $x^* = (2, 1, 0, 0)$. □

A modified version of the branch-and-bound algorithm for the knapsack problem can be used in connection with delayed column generation techniques to solve the cutting stock problem. To calculate all integer vectors $a \geq 0$ satisfying (2.50), one has to store all solutions calculated in Step 4 which satisfy the condition $\sum_{i=1}^{n} y_i a_i > 1$ and to delete Steps 5 to 7. Furthermore, one has to replace L by 1.

2.6 Dynamic Programming

Dynamic programming is another general technique to solve optimization problems exactly. This technique can only be applied if the studied problem satisfies the so-called Bellman's optimality principle which states that an optimal solution can be constructed recursively from optimal solutions of (smaller) subproblems. Usually, in a dynamic programming approach the recursion is organized in stages and is calculated "bottom-up" starting with the smallest subproblems.

In the following we demonstrate the method of dynamic programming using different examples. A first example for dynamic programming has already been considered in Subsection 2.2.4, namely the Floyd-Warshall algorithm to solve the all-pairs shortest path problem.

Example 2.15: Floyd-Warshall algorithm

In this algorithm values $d^k(i,j)$ are calculated in n stages $k = 1, \ldots, n$ by the recursion

$$d^k(i,j) := \min\{d^{k-1}(i,j), d^{k-1}(i,k) + d^{k-1}(k,j)\},$$

where in stage k only values $d^{k-1}(\cdot,\cdot)$ from the previous stage $k-1$ are used. According to Property 2.4 the value $d^k(i,j)$ is the length of a shortest i-j-path containing only nodes $1, 2, \ldots, k$ as internal nodes.

Starting with the initial values $d^0(i,j) := c_{ij}$ the values $d^k(i,j)$ for all nodes $i, j = 1, \ldots, n$ and stages $k = 1, \ldots, n$ can be calculated in $O(n^3)$ time. Furthermore, a solution of the all-pairs shortest path problem is given by the values $d^n(i,j)$ since these values are the lengths of shortest i-j-paths where all nodes $1, \ldots, n$ may be used as internal nodes. □

Another dynamic programming approach for the all-pairs shortest path problem is the algorithm developed by Bellman and Ford, which is a special implementation of the label-correcting algorithm.

Example 2.16: Bellman-Ford algorithm

In this algorithm, also variables $d^k(i,j)$ are defined for $k = 1, \ldots, n$, where $d^k(i,j)$ denotes the length of a shortest i-j-path containing at most k arcs. It is easy to see that after setting $d^1(i,j) := c_{ij}$ the values for $k = 2, \ldots, n$ can be calculated by the recursion

$$d^k(i,j) := \min\{d^{k-1}(i,j), \min_{(h,j)\in A}\{d^{k-1}(i,h) + c_{hj}\}\}. \tag{2.70}$$

While the first term in the recursion is the length of a shortest i-j-path containing at most $k-1$ arcs, in the second term for another node h all i-h-paths containing at most $k-1$ arcs are enlarged by the arc (h,j). In order to calculate the values $d^k(i,j)$ for all $i, j = 1, \ldots, n$ and $k = 1, \ldots, n$, we need $\sum_{k=1}^{n}\sum_{i=1}^{n}(\sum_{j=1}^{n}|A(j)|) = O(n^2 m)$ steps, where $|A(j)|$ denotes the number of nodes h with $(h,j) \in A$.

If the network contains no negative cycle, a shortest path contains at most $n-1$ arcs, i.e. the recursion (2.70) can already be stopped after iteration $k = n-1$ and the values $d^{n-1}(i,j)$ define a solution of the all-pairs shortest path problem. On the other hand, if negative cycles exist, the algorithm may also detect this by finding an improved value $d^n(i,j) < d^{n-1}(i,j)$ in iteration $k = n$. □

Also the binary knapsack problem introduced in Subsection 2.5.2 can be solved with dynamic programming.

Example 2.17: Knapsack problem Recall that in the binary knapsack problem we are given a knapsack with capacity b and n items $j = 1, \ldots, n$ with utility values c_j and weights a_j. The objective is to find a subset $I \subseteq \{1, \ldots, n\}$ with maximum value $\sum_{j \in I} c_j$ subject to the constraint that the total weight does not exceed the capacity b of the knapsack, i.e. such that $\sum_{j \in I} a_j \leq b$ holds. For each item j it has to be decided whether j is packed into the knapsack or not.

We define variables $c(j,k)$ for $j = 1, \ldots, n$ and $k = 0, \ldots, b$, where $c(j,k)$ denotes the maximum value of items from the subset $\{1, \ldots, j\}$ subject to the constraint that the weight of these items does not exceed the value k. These values can be calculated for $j = 1, \ldots, n$ and $k = 1, \ldots, b$ by the recursion

$$c(j,k) := \max\{c(j-1,k), c(j-1,k-a_j) + c_j\},$$

where initially $c(0,k) := 0$ for $k = 0, \ldots, b$ and $c(j,k) := \infty$ for $j = 0, \ldots, n$ and $k < 0$. While the first term in this recursion corresponds to the situation where item j is not packed into the knapsack, the second term correponds to the situation where j is included (which means that its value c_j is added and the weight capacity for the remaining items is reduced by a_j).

The optimal solution value of the knapsack problem is given by $c(n,b)$ which equals the maximum value that can be achieved by using items from the complete set $\{1, \ldots, n\}$ with weight capacity b. Obviously, all values $c(j,k)$ can be calculated in $O(nb)$ time, i.e. the algorithm is pseudo-polynomial. □

As a last example we consider the traveling salesman problem (TSP) which is one of the most famous combinatorial optimization problems.

Example 2.18: Traveling salesman problem

In the traveling salesman problem we are given a network $G = (V, V \times V)$ where the nodes $j = 1, \ldots, n$ represent cities and the arcs $(i,j) \in V \times V$ are weighted with the distances c_{ij} from i to j. The objective is to find a closed tour through all cities in which each city is visited exactly once and the total distance is minimized.

Since we are searching for a closed tour, an arbitrary node may be fixed as starting node (w.l.o.g. we choose node 1). Then variables $F(S,j)$ are defined for subsets $S \subseteq V$ and nodes $j \in S$, where $F(S,j)$ denotes the length of a shortest path from node 1 to node j visiting each node in S exactly once. If

we initially set $F(\{j\}, j) := c_{1j}$ for all nodes $j = 2, \ldots, n$, these values can be calculated by the recursion

$$F(S, j) = \min_{i \in S} \{f(S \setminus \{j\}, i) + c_{ij}\}.$$

During the initialization the one-element sets $S = \{j\}$ are considered and the lengths of the paths $1 \to j$ are set to c_{1j}. In the recursion all paths from the starting node 1 to a node $i \in S$ visiting all nodes in the set $S \setminus \{j\}$ once are enlarged by the arc (i, j). Finally, the optimal solution value is given by $\min_{j=2}^{n} \{F(\{2, \ldots, n\}, j) + c_{j1}\}$.

The values $F(S, j)$ have to be calculated for all 2^n subsets $S \subseteq \{2, \ldots, n\}$ and all nodes $j \in \{2, \ldots, n\}$, where each calculation can be done in $O(n)$ time. Thus, we get the exponential time complexity $O(n^2 2^n)$. The storage complexity is $O(n2^n)$ if all values are stored. However, if the calculation is organized in stages according to increasing cardinalities of the sets S, during the calculation for sets with cardinality k only values of sets with cardinality $k-1$ are needed. As soon as values for sets with cardinality $k+1$ are calculated, all values in stage $k-1$ can be deleted. Thus, it is sufficient to store only values for two consecutive stages. Since there are $\binom{n}{k}$ subsets with cardinality k and the maximum number is determined by $k = n/2$, the storage complexity can be reduced to $O(n\binom{n}{n/2})$. □

2.7 Local Search and Genetic Algorithms

For NP-hard problems it is in general very difficult to find optimal solutions. While brach-and-bound algorithms are only able to solve small instances exactly, for larger instances approximation algorithms must be applied. Such "heuristic" algorithms only provide feasible solutions which are not guaranteed to be optimal. However, for minimization (maximization) problems, a possible deviation from the optimal objective value can be estimated if lower (upper) bounds are available. In this section we discuss two general techniques for solving optimization problems heuristically: local search and genetic algorithms.

2.7.1 Local search algorithms

One of the most successful methods to deal with hard combinatorial optimization problems is the discrete analogy of "hill climbing", known as **local search** or **neighborhood search**. In this section we consider minimization problems, i.e. we have to find a solution $s^* \in \mathcal{S}$ with $c(s^*) \leq c(s)$ for all $s \in \mathcal{S}$.

Local search is an iterative procedure which moves from one solution in $\mathcal{S}$ to another until a stopping condition is satisfied. In order to move systematically through the solution set, the possible moves from one solution to another are

restricted by so-called **neighborhood structures** $\mathcal{N} : \mathcal{S} \to 2^{\mathcal{S}}$. For each solution $s \in \mathcal{S}$ the set $\mathcal{N}(s)$ describes the subset of solutions which can be reached from s in the next step. The set $\mathcal{N}(s)$ is also called the neighborhood of s.

A neighborhood structure $\mathcal{N}$ may be represented by a directed graph $G = (V, A)$ where $V = \mathcal{S}$ and $(s, s') \in A$ if $s' \in \mathcal{N}(s)$. G is called the **neighborhood graph** of the neighborhood structure. Since in general it is not possible to store the neighborhood graph completely, usually a set OP of **operators** (allowed modifications) $op : \mathcal{S} \to \mathcal{S}$ is introduced. For a given solution s, the neighborhood of s is defined by all solutions which can be obtained by applying an operator op from OP to s, i.e.

$$\mathcal{N}(s) = \{op(s) \mid op \in OP\}.$$

Using these definitions, a **local search method** may be described as follows. In each iteration we start with one solution $s \in \mathcal{S}$ and choose a solution $s' \in \mathcal{N}(s)$ (or an operator $op \in OP$ which provides $s' = op(s)$). Based on the objective function values $c(s)$ and $c(s')$, we choose a starting solution for the next iteration. According to different criteria for the choice of the next solution, different types of local search methods are obtained.

The simplest choice is to take the solution with the smallest objective function value in the neighborhood. This method is called **iterative improvement** or steepest descent and may be formulated as in Figure 2.25.

```
Algorithm Iterative Improvement
1.  Generate an initial solution s ∈ S;
2.  WHILE a solution s' ∈ N(s) with c(s') < c(s) exists DO
3.     Choose the best solution s' ∈ N(s);
4.     s := s'
5.  ENDWHILE
```

Figure 2.25: Iterative improvement

This algorithm terminates with some solution s^*. In general, s^* is only a **local minimum** with respect to the neighborhood $\mathcal{N}$ (i.e. no neighbor is better than this solution) and may differ considerably from the global minimum. To overcome this drawback, one could start the procedure with different starting solutions and take the best local minimum found in this way. Another possibility is to allow also deteriorations of the objective function value during the search process. But, if also non-improving solutions are accepted, solutions may be visited more than once during the search process and therefore the method may cycle. Thus, in such a method additional techniques to avoid cycling have to be integrated.

A method which tries to avoid cycling by randomization is **simulated annealing** which simulates an annealing process in physics. On the one hand, it chooses a solution $s' \in \mathcal{N}(s)$ randomly, on the other hand it accepts such a neighbor solution only with a certain probability.

More precisely, in the i-th iteration s' is accepted with probability

$$\min\left\{1, e^{-\frac{c(s')-c(s)}{t_i}}\right\},$$

where (t_i) is a sequence of positive control values with $\lim_{i\to\infty} t_i = 0$. If $c(s') \leq c(s)$, then this probability is one, i.e. a better solution s' is always accepted. If, on the other hand, $c(s') > c(s)$ holds, then s' is accepted with a probability which decreases with increasing i. This means that a local minimum can be left, but the probability for it gets smaller during the search process.

A general simulated annealing procedure is shown in Figure 2.26. In this procedure, `Rand(0,1)` denotes a number randomly generated from the interval $[0, 1]$.

```
Algorithm Simulated Annealing
 1.  i := 0;
 2.  Generate an initial solution s ∈ S;
 3.  best := c(s);
 4.  s* := s;
 5.  REPEAT
 6.      Generate randomly a solution s' ∈ N(s);
 7.      IF Rand(0,1) < min{1, e^(-(c(s')-c(s))/t_i)} THEN
 8.          s := s';
 9.          IF c(s') < best THEN
10.              s* := s';
11.              best := c(s');
12.          ENDIF
13.      ENDIF
14.      i := i + 1;
15.  UNTIL a stopping condition is satisfied
```

Figure 2.26: Simulated annealing

Often, the sequence (t_i) is defined by $t_{i+1} := \alpha t_i$ with a decreasing factor $0 < \alpha < 1$. The search may be stopped after a certain number of iterations, after a certain number of non-improving solutions, or when a given time-limit is reached.

A variant of simulated annealing is the **threshold acceptance method**. It differs from simulated annealing only by the acceptance rule for the randomly generated solution $s' \in \mathcal{N}(s)$. A neighbor s' is accepted in iteration i if the

difference $c(s') - c(s)$ is smaller than a threshold value $t_i \geq 0$. Again, the values t_i are decreased during the search process.

Another strategy to avoid cycling would be to store all visited solutions in a so-called **tabu list** TL and to accept a neighbor of the current solution only if it is not contained in this list (i.e. it is not tabu). Such procedures are called **tabu search** procedures. In each iteration the current solution is replaced by a non-tabu solution in its neighborhood (for a general tabu search procedure see Figure 2.27).

Due to memory restrictions in general it is not possible to store all visited solutions. Therefore, the list contains only solutions which have been visited in the last $|TL|$ iterations. Then only cycles with a length greater than the tabu list length $|TL|$ may occur and if $|TL|$ is sufficiently large, the probability of cycling becomes very small. Furthermore, not complete descriptions, but only typical properties (attributes) of the solutions are stored in the list. All solutions having one of the properties stored in the tabu list are declared as tabu. The properties are chosen such that it is guaranteed that a visited solution will not be reached again as long as the corresponding property is in the tabu list. A disadvantage of this procedure is that solutions which have never been visited may also be forbidden by the tabu list. To overcome this difficulty, so-called **aspiration criteria** are used which allow to accept neighbors even if they are forbidden due to the tabu status. For example, solutions which improve the best solution found so far should always be accepted.

```
Algorithm Tabu Search
 1.  Generate an initial solution s ∈ S;
 2.  best := c(s);
 3.  s* := s;
 4.  TL := ∅;
 5.  REPEAT
 6.      Cand(s) := {s' ∈ N(s) | the move from s to s' is not
         tabu or s' satisfies the aspiration criterion };
 7.      Choose a solution s' ∈ Cand(s);
 8.      Update the tabu list TL;
 9.      s := s';
10.      IF c(s') < best THEN
11.          s* := s';
12.          best := c(s');
13.      ENDIF
14.  UNTIL a stopping condition is satisfied
```

Figure 2.27: Tabu search

Again, different stopping criteria are possible. Usually, the whole tabu search procedure stops after a certain number of iterations, after a certain number of

non-improving solutions, or when a given time-limit is reached. Other steps which can be varied are step 7 (choosing a neighbor solution from the candidate set) and step 8 (updating the tabu list TL).

If the neighborhood is small, we may always choose the best neighbor $s' \in Cand(s)$ with respect to the cost function c (best-fit). On the other hand, if the neighborhood is larger, such a strategy may be too time-consuming. Then often the first neighbor $s' \in Cand(s)$ with $c(s') < c(s)$ is chosen (first-fit). If no such solution exists, the best solution in $Cand(s)$ is determined.

According to the tabu list management **static** and **dynamic** tabu lists are distinguished. While a static tabu list has a constant size, a dynamic tabu list has a variable length. A dynamic tabu list is often managed according to the following rules:

- If in some iteration a solution is found which improves the best solution found so far, the tabu list is emptied, since it is known that this solution has never been visited before.
- In an improving phase of the search (i.e. the objective value of the current solution is better than the value of the previous iteration) the tabu list length is decreased by one if it is greater than a certain constant $TL_{\min}$.
- In a non-improving phase of the search the tabu list length is increased by one if it is smaller than a certain constant $TL_{\max}$.

Besides the tabu list, which has the function of a short-term memory, often also a long-term memory is kept which is used for **diversification**. In this long-term memory promising solutions or properties of promising solutions which are visited during the search process are stored. If within the search process the best solution found so far is not improved in a certain number of iterations (intensification), the search process is stopped and restarted with a new starting solution (diversification). Since a restart from a randomly generated solution would neglect all information of the previous search process, the long-term memory is used to restart in a hopefully good region. Either a promising solution from the long-term memory list is selected or, in the case where only properties have been stored, a solution having the properties in the long-term memory is generated.

If a local search method is developed for a certain problem, the most important parts are a suitable representation of the solutions and the definition of operators which define an appropriate neighborhood.

Example 2.19: Consider the following NP-hard single-machine scheduling problem: Given are n jobs $j = 1, \ldots, n$ with processing times p_j, due dates d_j and weights w_j. The objective is to determine a schedule S minimizing the total weighted tardiness $c(S) = \sum_{j=1}^{n} w_j T_j = \sum_{j=1}^{n} w_j \max\{0, C_j - d_j\}$.

For a single-machine problem with a regular objective function schedules may be represented by permutations of all jobs. Associated with such a permutation

is a schedule in which all jobs are processed without idle times starting at time zero.

A simple neighborhood is the so-called adjacent pairwise interchange neighborhood $\mathcal{N}_{api}$ containing all permutations which can be obtained by interchanging two adjacent jobs. Thus, each permutation has $n-1$ neighbors. If during a tabu search procedure we move from one solution to another by interchanging the adjacent jobs i, j we store the pair (i, j) in the tabu list. As long as this pair is in the list, it is not allowed to swap jobs i and j again.

Consider the following instance with $n = 4$ jobs.

j	1	2	3	4
p_j	8	6	11	3
d_j	12	7	12	9
w_j	6	8	15	5

We start with the initial permutation $\pi^0 := (4, 1, 3, 2)$ with completion times $C = (C_1, C_2, C_3, C_4) = (11, 28, 22, 3)$ and objective function value $c(\pi^0) = 6 \cdot 0 + 8 \cdot (28 - 7) + 15 \cdot (22 - 12) + 5 \cdot 0 = 318$. This solution has the three neighbors $(1, 4, 3, 2)$, $(4, 3, 1, 2)$, $(4, 1, 2, 3)$ with values 328, 258 and 320. We choose the best neighbor $\pi^1 = (4, 3, 1, 2)$ with $c(\pi^1) = 258$ as the next solution and insert the pair of swapped jobs $(1, 3)$ into the tabu list. Then we proceed as shown in Figure 2.28, where solutions which are tabu are crossed.

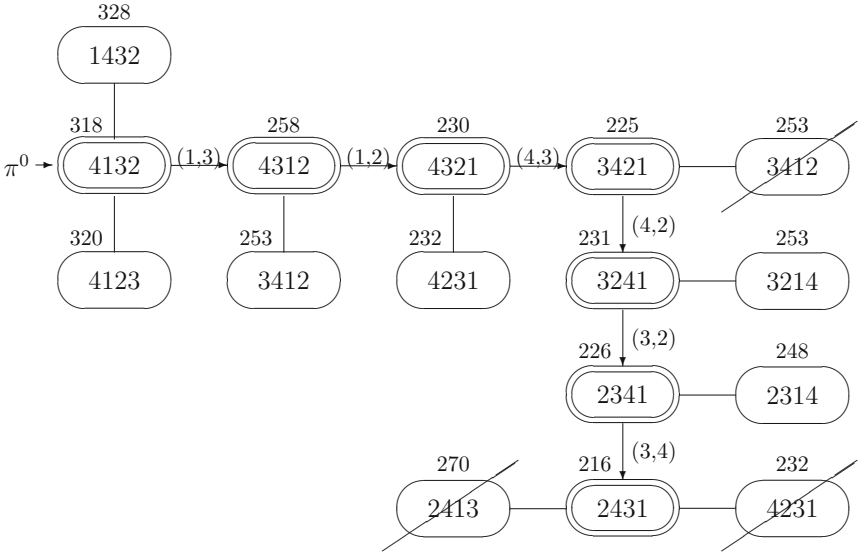


Figure 2.28: Tabu search for example 2.19

The next considered solutions are $\pi^2 = (4, 3, 2, 1)$ with $c(\pi^2) = 230$ and $\pi^3 = (3, 4, 2, 1)$ with $c(\pi^3) = 225$. For these moves the pairs $(1, 2)$ and $(4, 3)$ are

added to the tabu list. While iterative improvement would terminate in the local optimum π^3 since no better neighbor exists, the tabu search procedure moves on to $\pi^4 = (3, 2, 4, 1)$ with $c(\pi^4) = 231$. Note that in this iteration the neighbor $(3, 4, 1, 2)$ is tabu since $(1, 2)$ is contained in the tabu list. When we consider the next solution $\pi^5 = (2, 3, 4, 1)$ with $c(\pi^5) = 226$, its best neighbor $\pi^6 = (2, 4, 3, 1)$ is tabu since $(4, 3)$ is contained in the tabu list. But since $c(\pi^6) = 216$ is better than the best solution found so far, due to the aspiration criterion its tabu status is cancelled. Afterwards, the tabu search terminates in the solution π^6 since all its neighbors are tabu and the aspiration criterion is not satisfied.

Note that for this instance π^6 is also a global optimum (in general a tabu search procedure provides only a local optimum). □

An important property for a neighborhood structure is the so-called **connectivity**. A neighborhood $\mathcal{N}$ is called **connected** if it is possible to transform each solution s into each other solution s' by a finite number of moves in $\mathcal{N}$, i.e. if a sequence of solutions $s_0 = s, s_1, \ldots, s_k = s'$ exists with $s_\lambda \in \mathcal{N}(s_{\lambda-1})$ for $\lambda = 1, \ldots, k$. A weaker form of connectivity is the so-called **opt-connectivity**. A neighborhood $\mathcal{N}$ is called **opt-connected** if it is possible to transform each solution s into an optimal solution s^* by a finite number of moves in $\mathcal{N}$, i.e. if a sequence of solutions $s_0 = s, s_1, \ldots, s_k = s^*$ exists with $s_\lambda \in \mathcal{N}(s_{\lambda-1})$ for $\lambda = 1, \ldots, k$ and s^* is optimal.

If a neighborhood is not opt-connected, it may happen that for bad starting solutions no optimal solution can be reached. Practical experiments in connection with simulated annealing or tabu search have shown that these heuristics generally provide better results if the neighborhood is opt-connected.

2.7.2 Genetic algorithms

A **genetic algorithm** (GA) is a general search technique which mimics the biological evolution and is based on the principle "survival of the fittest". Genetic algorithms have been applied with growing success to combinatorial optimization problems. Unlike the heuristics discussed in the previous subsection (like tabu search or simulated annealing), which consider only a single solution at a time, a GA works on a set POP of solutions, which is called a **population**. A solution is usually encoded as a sequence of symbols which is called a **chromosome**. Associated with an encoding s of a feasible solution is a measure of adaptation, the fitness value $fit(s)$ (which is often related to the objective function value $c(s)$).

Starting from an initial population, some "parent" solutions are selected and new "child" solutions are generated by applying some genetic operators to the parents. While a **crossover** operator usually mixes subsequences of the parent chromosomes, a **mutation** operator perturbates a chromosome. The generated child solutions are added to the population and the population is reduced to

its original size by removing some solutions according to their fitness values. Afterwards, the same process is applied to the new population, etc. (for a description of a general genetic algorithm see Figure 2.29).

```
Genetic Algorithm
1.  Generate an initial population POP;
2.  Calculate a fitness value for each individual s ∈ POP;
3.  REPEAT
4.     Choose two parent solutions s^M, s^F ∈ POP;
5.     Create a child solution s^C from s^M, s^F by crossover;
6.     Mutate s^C with a certain probability;
7.     Calculate the fitness value of s^C;
8.     Add s^C to POP and reduce POP by selection;
9.  UNTIL a stopping condition is satisfied
```

Figure 2.29: Genetic algorithm

Again, different variations of this generic algorithm are possible. For example, in steps 4 and 5 several child solutions may be created simultaneously. If POP contains an even number of individuals, POP can be divided into two sets of "mothers" and "fathers". Then each mother s^M is paired with a father s^F and for each pair (s^M, s^F) two children s^{C_1} and s^{C_2} are created by crossover. After mutation all created child solutions are added to the parent population and the joined population is reduced to its original size by selection (e.g. by removing the solutions with the smallest fitness values). Mutations may also be realized by applying some steps of a local search procedure.

Example 2.20: Consider again Example 2.19. Assume that we have selected the two parent sequences $s^M = (4, 1, 3, 2)$ with $c(s^M) = 318$ and $s^F = (1, 2, 4, 3)$ with $c(s^F) = 336$. For example, the crossover operator may take the first two elements from the mother sequence s^M and the remaining elements from the father sequence s^F (in the same order as they occur in s^F). Thus, the created child sequence is $s^C = (4, 1, 2, 3)$ with $c(s^C) = 320$. Afterwards, s^C may be mutated by swapping two adjacent elements, i.e. s^C may become $s^C = (4, 2, 1, 3)$ with $c(s^C) = 286$. After adding s^C to the population, the sequence s^F may be removed (since it has the largest objective function value). □

2.8 Reference Notes

Basics concerning complexity theory can be found in the book of Garey and Johnson [57] and e.g. in Papadimitriou and Steiglitz [118]. The first NP-completeness proof was given by Cook [35] for the satisfiability problem. Complexity results for problems with resource constraints were firstly derived by Błażewicz et al. [14]. A large number of complexity results and corresponding references for different classes of scheduling problems are collected on the website [28].

While Section 2.3 about linear programming is based on the textbook of Chvátal [34], the sections about shortest path and network flow algorithms are strongly based on the book of Ahuja et al. [3]. Delayed column generation techniques were firstly applied by Gilmore and Gomory [58] to the cutting-stock problem. In 1977 Bland [11] has shown that cycles can be avoided if the smallest index rule is used. The first counterexample for which the simplex algorithm with the largest coefficient rule needs an exponential number of iterations was given by Klee and Minty [81]. The ellipsoid method was proposed by Khachian [79].

Dijkstra presented his algorithm firstly 1959 in [45], the algorithm of Floyd can be found in [55]. The presented pseudo-polynomial labeling algorithm for the maximum flow problem was developed by Ford and Fulkerson [56] in 1956. The cycle-canceling algorithm for the minimum cost flow problem is due to Klein [82]. Detailed references on preflow-push and scaling algorithms can be found in Ahuja et al. [3].

The linear programming formulation of the RCPSP in Section 2.3.4 was firstly proposed by Pritsker et al. [125]. The technique of dynamic programming was invented by Bellman [10].

More about combinatorial optimization problems and general solution methods (like branch-and-bound or dynamic programming) can be found in the old books by Lawler [96], Papadimitriou and Steiglitz [118] or the new books by Korte and Vygen [93], Schrijver [130].

Much research has been done in the area of local search and genetic algorithms. General topics of local search and applications to different combinatorial optimization problems are discussed in the book of Aarts and Lenstra [2]. Simulated annealing was first described by Kirkpatrick et al. [80], genetic algorithms were introduced by Holland [73]. For tabu search see Glover [59], [60] and Glover and Laguna [61].

Chapter 3

Resource-Constrained Project Scheduling

As shown in Section 2.1, even special cases of the RCPSP are NP-hard, i.e. with high probability they cannot be exactly solved in polynomial time. Thus, for larger instances in practice approximation algorithms have to be applied. In this chapter we present different solution methods for the RCPSP with the makespan objective and some of its generalizations (like the multi-mode RCPSP or more general objective functions).

After introducing some basic concepts in Section 3.1, in Section 3.2 we discuss constraint propagation techniques, which are useful to reduce the solution space. Section 3.3 is devoted to methods for calculating lower bounds. Foundations for heuristic procedures like priority-based heuristics, local search or genetic algorithms are described in Section 3.4. In Section 3.5 some branch-and bound algorithms are discussed. Finally, in Section 3.6 more general objective functions for problems without resource constraints are considered.

3.1 Basics

Usually, before solution algorithms are applied to the RCPSP, a temporal analysis is performed. If no generalized precedence relations are given, the activity-on-node network $G = (V, A)$ of an RCPSP-instance must be acyclic (since otherwise no feasible schedule exists). Thus, the activities can be numbered according to a topological ordering, i.e. we may assume $i < j$ for all precedences $i \rightarrow j \in A$.

We assume $S_0 = 0$, i.e. no activity can start before time $t = 0$. Furthermore, an upper bound UB for the $C_{\max}$-value is given, i.e. we have $S_{n+1} \leq UB$. For each activity i we define a

- **head** r_i as a lower bound for the earliest starting time of i, and a
- **tail** q_i as a lower bound for the length of the time period between the completion time of i and the optimal makespan.

Heads and tails may be calculated as follows. The length of a (directed) path P from activity i to j in G is the sum of processing times of all activities in P excluding the processing time of j. A valid head r_i is the length of a longest path from the dummy activity 0 to i. Symmetrically, a valid tail q_i is the length of a longest path from i to the dummy node $n+1$ minus the processing time p_i of i. Given an upper bound UB, a deadline for the completion time of activity i in any schedule with $C_{\max} \leq UB$ may be defined by $d_i := UB - q_i$. Thus, activity i must be completely processed within its **time window** $[r_i, d_i]$ in any feasible schedule with $C_{\max} \leq UB$.

In an acyclic graph $G = (V, A)$ heads r_i and deadlines d_i for the activities $i = 0, 1, \ldots, n, n+1$ may be calculated in a topological ordering by the recursions

$$\begin{array}{lll} r_0 := 0; & r_i := \max\limits_{\{j|j\to i\in A\}} \{r_j + p_j\} & \text{for } i = 1, 2, \ldots, n+1 \text{ and} \\ d_{n+1} := UB; & d_i := \min\limits_{\{j|i\to j\in A\}} \{d_j - p_j\} & \text{for } i = n, n-1, \ldots, 0. \end{array}$$

A longest path CP from 0 to $n+1$ is also called a **critical path**. The length of a critical path r_{n+1} is equal to the minimal makespan if all resource constraints are relaxed. For the problem with resource constraints the critical path length is a lower bound for the optimal makespan. The schedule in which all activities i start as early as possible at $S_i := r_i$ is called **earliest start schedule**, the schedule in which all activities i start as late as possible at $S_i := d_i - p_i$ is called **latest start schedule**. For each activity i the so-called **slack** $s_i := d_i - p_i - r_i$ defines the amount of time by which i can be moved in any feasible schedule with $C_{\max} \leq UB$. Activities with $s_i = 0$ are called **critical** since they cannot be moved in any feasible schedule with $C_{\max} \leq UB$.

Example 3.1: Consider the project with $n = 6$ activities shown in Figure 3.1.

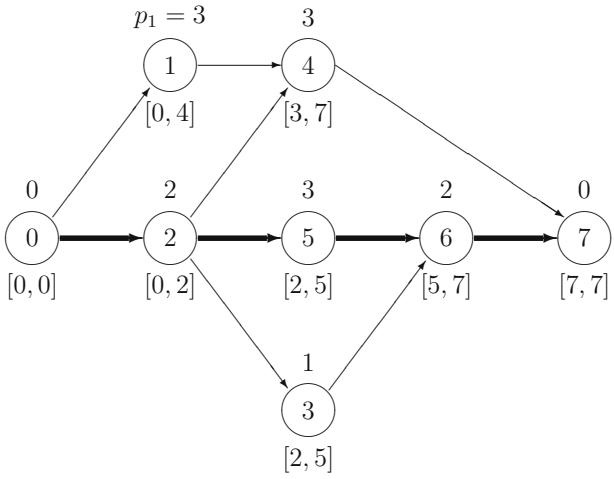


Figure 3.1: Time windows for a project with $n = 6$ and $UB = 7$

For $UB = 7$ we obtain the following heads r_i, tails q_i, deadlines d_i and slacks s_i:

i	1	2	3	4	5	6
r_i	0	0	2	3	2	5
q_i	3	5	2	0	2	0
d_i	4	2	5	7	5	7
s_i	1	0	2	1	0	0

For example, for activity $i = 4$ we get

$$r_4 := \max\{r_1 + p_1, r_2 + p_2\} = \max\{0 + 3, 0 + 2\} = 3,$$

for activity $i = 2$ we get

$$d_2 := \min\{d_5 - p_5, d_4 - p_4, d_3 - p_3\} = \min\{5 - 3, 7 - 3, 5 - 1\} = 2.$$

Activities 2, 5, and 6 are critical, a corresponding critical path is $CP = (0 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 7)$ with length $r_7 = 7$. □

3.2 Constraint Propagation

Constraint propagation is a technique which deduces new constraints from given ones. This method may be used to tighten the search space in connection with branch-and-bound methods or local search heuristics. Also infeasibility of an instance may be detected, which is important for lower bound calculations. In connection with the RCPSP usually additional timing restrictions are derived from given temporal and resource constraints.

3.2.1 Basic relations

Given a schedule $S = (S_i)_{i=0}^{n+1}$, for each pair (i, j) of activities i, j exactly one of the three relations $i \rightarrow j$, $j \rightarrow i$, or $i \parallel j$ holds:

- We have a so-called **conjunction** $i \rightarrow j$ if activity j does not start before i is finished, i.e. if
$$S_i + p_i \leq S_j. \tag{3.1}$$
- We have a so-called **parallelity relation** $i \parallel j$ if activities i and j are processed in parallel for at least one time unit, i.e. if
$$S_i + p_i > S_j \text{ and } S_j + p_j > S_i. \tag{3.2}$$

Furthermore, the negation of a parallelity relation $i \parallel j$ is a so-called **disjunction** $i - j$ which means that either $i \rightarrow j$ or $j \rightarrow i$ must hold.

We denote the set of all conjunctions by C, the set of all disjunctions by D and the set of all parallelity relations by N. The following relations may be immediately derived from the given data of an RCPSP-instance: An initial set C_0 of conjunctions is given by the set of all precedence relations $i \to j \in A$. An initial set D_0 of disjunctions is induced by resource constraints for pairs of activities by setting $i - j \in D_0$ if $r_{ik} + r_{jk} > R_k$ for some renewable resource $k \in \mathcal{K}^\rho$ holds.

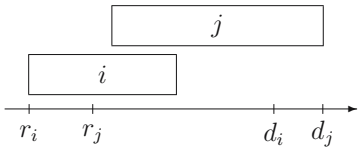


Figure 3.2: Overlapping activities

An initial set N_0 of parallelity relations may be derived by considering two activities i, j with time windows $[r_i, d_i]$ and $[r_j, d_j]$. If

$$p_i + p_j > \max\{d_i, d_j\} - \min\{r_i, r_j\} \tag{3.3}$$

holds, then i and j overlap in any feasible schedule with $C_{\max} \leq UB$ (cf. Figure 3.2). Thus, we may define N_0 as the set of all parallelity relations $i \parallel j$ where i and j satisfy (3.3).

3.2.2 Start-start distance matrix

Let $S = (S_i)_{i=0}^{n+1}$ be a feasible schedule with $S_0 = 0$ and $S_{n+1} \leq UB$. Then we have $i \to j$ if and only if (3.1), i.e. $S_j - S_i \geq p_i$ holds. Furthermore, we have $i \parallel j$ if and only if (3.2) holds which is equivalent to

$$S_j - S_i \geq -(p_j - 1) \text{ and } S_i - S_j \geq -(p_i - 1) \tag{3.4}$$

because all data are integers. Additionally, for arbitrary activities i, j we have

$$S_i + p_i \leq S_{n+1} \leq UB \leq UB + S_j \text{ or equivalently } S_j - S_i \geq p_i - UB$$

because $S_j \geq S_0 = 0$ and $S_{n+1} \leq UB$.

If we define

$$d_{ij} := \begin{cases} 0, & \text{if } i = j \\ p_i, & \text{if } i \to j \\ -(p_j - 1), & \text{if } i \parallel j \\ p_i - UB, & \text{otherwise,} \end{cases} \tag{3.5}$$

then the entries d_{ij} $(i, j = 0, \ldots, n+1)$ of the $(n+2) \times (n+2)$-matrix $d = (d_{ij})$ are lower bounds for the differences $S_j - S_i$, i.e.

$$S_j - S_i \geq d_{ij} \text{ for all } i, j = 0, \ldots, n+1. \tag{3.6}$$

We call d a **start-start distance (SSD)-matrix**.

If additionally generalized precedence constraints (1.1) with time-lags d'_{ij} are given, we have to fulfill $S_i + d'_{ij} \leq S_j$ or equivalently $S_j - S_i \geq d'_{ij}$. Thus, we may incorporate these constraints by setting

$$d_{ij} := \max\{d_{ij}, d'_{ij}\}. \tag{3.7}$$

In this situation all results derived in the following are also valid for the RCPSP with generalized precedence constraints.

The first row of an SSD-matrix represents heads because $S_i = S_i - S_0 \geq d_{0i}$ for each activity. Similarly, the last column of a SSD-matrix contains information on the tails of the activities. More precisely, $d_{i,n+1} - p_i$ is a tail of activity i because

$$S_{n+1} - (S_i + p_i) = (S_{n+1} - S_i) - p_i \geq d_{i,n+1} - p_i.$$

Due to (3.1) and (3.4) we have:

- $i \to j$ if and only if $d_{ij} \geq p_i$ holds, (3.8)
- $i \parallel j$ if and only if both $d_{ij} \geq -(p_j - 1)$ and $d_{ji} \geq -(p_i - 1)$ hold. (3.9)

Some disjunctions may immediately be changed into conjunctions by the following observation. If for activities i, j with $i - j \in D$ the inequality $d_{ij} \geq -(p_j - 1)$ holds,

$$S_j - S_i \geq d_{ij} \geq -(p_j - 1), \text{ i.e. } C_j = S_j + p_j \geq S_i + 1 > S_i, \tag{3.10}$$

which implies $i \to j$. In this case we may set $d_{ij} := p_i$.

The relation $S_j - S_i \geq d_{ij}$ has the following transitivity property:

$$S_j - S_i \geq d_{ij} \text{ and } S_k - S_j \geq d_{jk} \text{ imply } S_k - S_i \geq d_{ij} + d_{jk}. \tag{3.11}$$

Due to (3.11) the matrix $d = (d_{ij})$ can be replaced by its transitive closure $\overline{d} = (\overline{d}_{ij})$. This can be done with complexity $O(n^3)$ by applying the Floyd-Warshall algorithm (cf. Section 2.2.4 and Example 2.15) to d, where for $i, j = 0, \ldots, n+1$ the recursion

$$d^k(i,j) := \max\{d^{k-1}(i,j), d^{k-1}(i,k) + d^{k-1}(k,j)\}$$

is evaluated for $k = 1, \ldots, n+1$.

If $d_{ii} > 0$ for some activity i holds, we get the contradiction $0 = S_i - S_i \geq d_{ii} > 0$. Thus, in this situation no feasible schedule with $C_{\max} \leq UB$ exists and we may state infeasibility. Especially, if generalized precedence relations are given, the test $d_{ii} > 0$ is important. In this case the activity-on-node network $G = (V, A)$ with arc weights d_{ij} contains a positive cycle, which implies that no feasible schedule exists even if all resource constraints are relaxed.

Calculating the transitive closure of the SSD-matrix is the simplest form of constraint propagation. One may start with a SSD-matrix containing basic

information about an RCPSP-instance like the relations in C_0, D_0, N_0, heads, tails and generalized precedence constraints. Then some constraint propagation methods are applied which result in additional relations and increased entries of d. More complicated constraint propagation methods will be discussed in the next subsections.

3.2.3 Symmetric triples and extensions

In this subsection we show how additional relations can be derived from the given ones and some further conditions. A triple (i, j, k) of activities is called a **symmetric triple** if

- $k \parallel i$ and $k \parallel j$, and
- i, j, k cannot be processed simultaneously due to resource constraints.

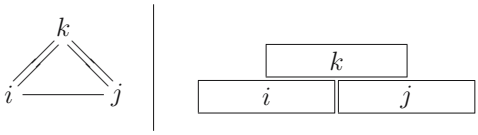


Figure 3.3: Symmetric triple

For a symmetric triple (i, j, k) we can add $i - j$ to D (see Figure 3.3).

Proof: Assume that $i - j$ does not hold. Thus, we have $i \parallel j$, i.e. i and j overlap in some interval $[t, t']$ (see Figure 3.4).

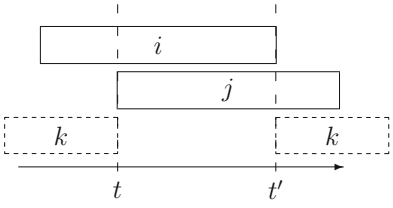


Figure 3.4: Assumption $i \parallel j$

Since i, j, k cannot be processed simultaneously, k must be completed not later than time t or cannot be started before time t'. In both cases k cannot overlap with both i and j, which is a contradiction. □

All symmetric triples can be found in $O(nr|N|)$ time because for each $k \parallel j \in N$ we check for at most $O(n)$ activities i whether $k \parallel i$ holds. If this is the case, we check in $O(r)$ time whether a resource exists such that i, j, k cannot be processed simultaneously.

In connection with symmetric triples (i, j, k) further relations can be deduced:

(1) If $l \parallel i$ and j, k, l cannot be processed simultaneously, then $l - j$ can be added to D (see Figure 3.5). If additionally $i \to j \in C$ $(j \to i \in C)$, then $l \to j \in C$ $(j \to l \in C)$ may be deduced.

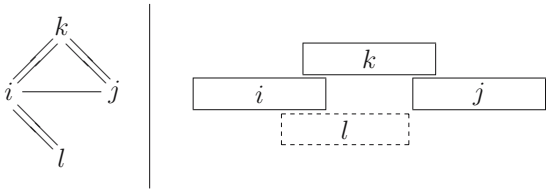


Figure 3.5: Extension of a symmetric triple – Condition (1)

Proof: If $l \parallel j$ would hold, then we have a situation as shown in Figure 3.6 (where the 'roles' of i and j may be interchanged).

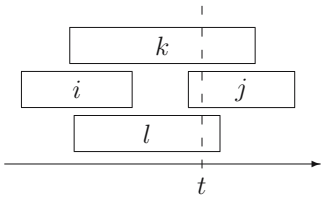


Figure 3.6: Assumption $l \parallel j$ in Condition (1)

Thus, at some time t activities j, k, l must be processed simultaneously, which is a contradiction. Furthermore, $i \to j$ implies $l \to j$ because otherwise $j \to l$ would imply $i \to l$ by transitivity, which contradicts $l \parallel i$. Similarly, the last claim can be proved. □

(2) Suppose that the conditions $p_k - 1 \leq p_i$, $p_k - 1 \leq p_j$ and $p_k - 1 \leq p_l$ hold. If i, k, l and j, k, l cannot be processed simultaneously, then $k - l \in D$ may be deduced (see Figure 3.7).

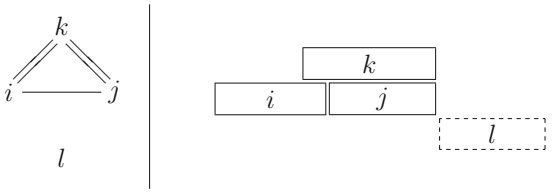


Figure 3.7: Extension of a symmetric triple – Condition (2)

Proof: Because $p_k - 1 \leq p_l$, it is not possible that l is processed between i and j since i and j occupy at least two time units of k.

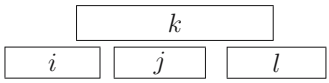


Figure 3.8: Assumption $k \parallel l$ in Condition (2)

Thus, if $k \parallel l$ holds, then we have a situation as shown in Figure 3.8 (where the roles of i and j may be interchanged or l may be processed before i and j). But this contradicts $p_k - 1 \leq p_j$ or $p_k - 1 \leq p_i$. □

Other propagation conditions are listed below. The proofs are similar to the proofs for (1) and (2). All the checks can be done in $O(rn^2|N|)$ time.

(3) Suppose that the conditions $p_k - 1 \leq p_i$ and $p_k - 1 \leq p_l$ hold. Furthermore, assume that the precedence relations $l \rightarrow j$ and $i \rightarrow j$ are given and i, k, l cannot be processed simultaneously. Then the additional conjunction $l \rightarrow k \in C$ can be fixed (see Figure 3.9).

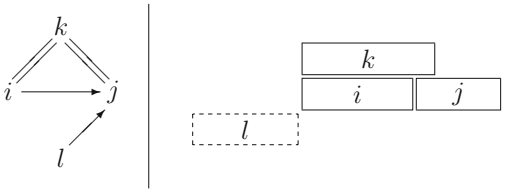


Figure 3.9: Extension of a symmetric triple – Condition (3)

(4) Suppose that the conditions $p_k - 1 \leq p_j$ and $p_k - 1 \leq p_l$ hold. Furthermore, assume that the precedence relations $i \rightarrow j$ and $i \rightarrow l$ are given and j, k, l cannot be processed simultaneously. Then symmetrically to (3) the additional conjunction $k \rightarrow l \in C$ can be fixed (see Figure 3.10).

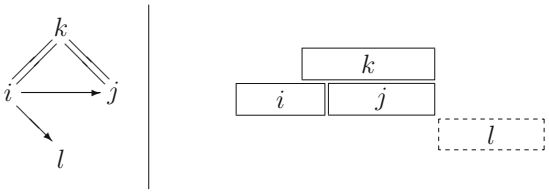


Figure 3.10: Extension of a symmetric triple – Condition (4)

(5) Suppose that the conditions $p_k - 1 \leq p_i$ and $p_k - 1 \leq p_j$ hold. Furthermore, assume that the precedence relations $l \rightarrow i$ and $l \rightarrow j$ ($i \rightarrow l$ and $j \rightarrow l$) are given. Then we may fix the conjunction $l \rightarrow k \in C$ ($k \rightarrow l \in C$) (see Figure 3.11).

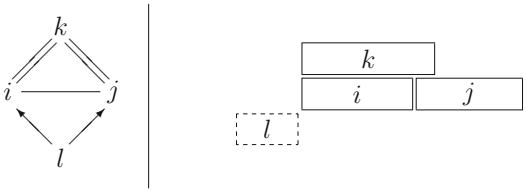


Figure 3.11: Extension of a symmetric triple – Condition (5)

(6) Let the parallelity relations $l \parallel i$ and $l \parallel j$ be given. Then the additional parallelity relation $l \parallel k$ can be fixed (see Figure 3.12).

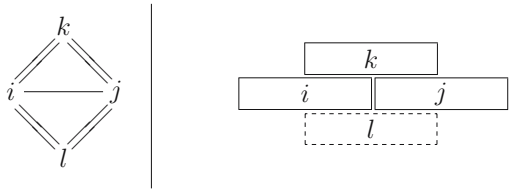


Figure 3.12: Extension of a symmetric triple – Condition (6)

3.2.4 Disjunctive sets

A subset $I \subseteq \{1, \ldots, n\}$ of (non-dummy) activities with $|I| \geq 2$ is called a **disjunctive set** (or **clique**) if $i - j \in D$, $i \to j \in C$ or $j \to i \in C$ for all $i, j \in I$ with $i \neq j$ holds. Disjunctive sets are cliques in the undirected graph defined by all disjunctions and conjunctions.

Examples of disjunctive sets are

- the set of all activities which need a disjunctive resource, i.e. a renewable resource k with capacity $R_k = 1$,
- the set of all jobs in a single machine scheduling problem,
- the set of all operations belonging to the same job in a general-shop problem,
- the set of operations to be processed by the same machine in a general-shop problem,
- the set of multiprocessor-tasks which need the same machine (i.e. all tasks j with $M_i \in \mu_j$ for a fixed machine M_i).

We define the total processing time of such a disjunctive set I by $P(I) := \sum_{i \in I} p_i$. For each activity i let $[r_i, d_i]$ be a time window for i with $r_i + p_i \leq d_i$ (otherwise

no feasible solution exists). Then additional conjunctions $i \to j$ and smaller time windows $[r'_i, d'_i] \subseteq [r_i, d_i]$ for activities of a disjunctive set I may be derived from the following general result:

Theorem 3.1 Let I be a disjunctive set and $J', J'' \subset J \subseteq I$ with $J' \cup J'' \neq \emptyset$. If

$$\max_{\substack{\nu \in J \setminus J' \\ \mu \in J \setminus J'' \\ \nu \neq \mu}} (d_\mu - r_\nu) < P(J), \tag{3.12}$$

then in J an activity from J' must start first or an activity from J'' must end last in any feasible schedule.

Proof: If no activity in J' starts first and no activity in J'' ends last in a feasible schedule, then all activities in J must be processed in a time interval of length

$$\max_{\substack{\nu \in J \setminus J' \\ \mu \in J \setminus J'' \\ \nu \neq \mu}} (d_\mu - r_\nu),$$

which is not possible if (3.12) holds. The condition $\nu \neq \mu$ may be imposed since in a non-preemptive schedule for at least two activities an activity which starts first cannot complete also last. □

Note that Theorem 3.1 also holds if the condition $\nu \neq \mu$ is dropped and only the weaker condition

$$\max_{\substack{\nu \in J \setminus J' \\ \mu \in J \setminus J''}} (d_\mu - r_\nu) = \max_{\mu \in J \setminus J''} d_\mu - \min_{\nu \in J \setminus J'} r_\nu < P(J) \tag{3.13}$$

is satisfied. By choosing different subsets J' and J'', different tests may be derived. The following tests of this type are usually considered in the literature:

- **Input test:** Let $J' = \{i\}$ with $i \in J$ and $J'' = \emptyset$. If we set $\Omega := J \setminus \{i\}$, then condition (3.13) can be rewritten as

  $$\max_{\mu \in \Omega \cup \{i\}} d_\mu - \min_{\nu \in \Omega} r_\nu < P(\Omega \cup \{i\}).$$

  If for some $\Omega \subseteq I$ and $i \in J \setminus \Omega$ this condition holds, then activity i must start first in J, i.e. we conclude that $i \to j$ for all $j \in \Omega$ (writing $i \to \Omega$). In this case i is called **input** of J.

- **Output test:** Let $J' = \emptyset$ and $J'' = \{i\}$ with $i \in J$. If we set $\Omega := J \setminus \{i\}$, then condition (3.13) can be rewritten as

  $$\max_{\mu \in \Omega} d_\mu - \min_{\nu \in \Omega \cup \{i\}} r_\nu < P(\Omega \cup \{i\}).$$

  If for some $\Omega \subseteq I$ and $i \in J \setminus \Omega$ this condition holds, then activity i must end last in J, i.e. we conclude that $j \to i$ for all $j \in \Omega$ (writing $\Omega \to i$). In this case i is called **output** of J.

- **Input-or-Output test:** Let $J' = \{i\}$ and $J'' = \{j\}$ with $i, j \in J$. Then condition (3.13) can be rewritten as

$$\max_{\mu \in J \setminus \{j\}} d_\mu - \min_{\nu \in J \setminus \{i\}} r_\nu < P(J).$$

  If this condition holds, then activity i must start first in J or activity j must end last in J, i.e. i is input for J or j is output for J. If $i \neq j$, the conjunction $i \rightarrow j$ is implied.

- **Input/Output negation test:**

  Let $J' = J \setminus \{i\}$ and $J'' = \{i\}$ with $i \in J$. If we set $\Omega := J \setminus \{i\}$, then condition (3.12) can be rewritten as

$$\max_{\mu \in \Omega} d_\mu - r_i < P(\Omega \cup \{i\}).$$

  If for some $\Omega \subseteq I$ and $i \in J \setminus \Omega$ this condition holds, then i cannot start first in J (**input negation**, writing $i \nrightarrow \Omega$).

  Let $J' = \{i\}$ and $J'' = J \setminus \{i\}$ with $i \in J$. If we set $\Omega := J \setminus \{i\}$, then condition (3.12) can be rewritten as

$$d_i - \min_{\nu \in \Omega} r_\nu < P(\Omega \cup \{i\}).$$

  If for some $\Omega \subseteq I$ and $i \in J \setminus \Omega$ this condition holds, then i cannot end last in J (**output negation**, writing $\Omega \nrightarrow i$).

  Note that with the input/output negation tests no additional conjunctions can be derived but time windows may be strengthened.

In the following we describe in more detail how the different consistency tests can be applied in a systematic and efficient way. For each test we describe a procedure which may be applied iteratively to the data until no more additional constraints are deduced (i.e. a so-called fixed-point is reached).

Input/Output tests

In the input/output tests we are interested in an activity i which must be processed first (or last) in a certain set J of activities, i.e. i is input (or output) of J. To derive an algorithm which performs such input and output tests in a systematic way, we consider for all $i \in I$ and subsets $\Omega \subseteq I$ with $i \notin \Omega$ the following implications:

$$\max_{\mu \in \Omega \cup \{i\}} d_\mu - \min_{\nu \in \Omega} r_\nu < P(\Omega) + p_i \Rightarrow i \rightarrow \Omega, \tag{3.14}$$

i.e. i has to be scheduled first in $\Omega \cup \{i\}$. Symmetrically,

$$\max_{\mu \in \Omega} d_\mu - \min_{\nu \in \Omega \cup \{i\}} r_\nu < P(\Omega) + p_i \Rightarrow \Omega \rightarrow i, \tag{3.15}$$

i.e. i has to be scheduled last in $\Omega \cup \{i\}$.

In the case (3.14) we may introduce the additional conjunctions $i \rightarrow j$ for all $j \in \Omega$, in the case (3.15) we may introduce the additional conjunctions $j \rightarrow i$ for all $j \in \Omega$.

Furthermore, with (3.15) the release date r_i and with (3.14) the deadline d_i of activity i may be improved. In the output situation (3.15) i cannot be started before all activities in the set Ω are completed. Symmetrically, in the input situation (3.14) i cannot be completed later than all activities in the set Ω are started.

In the following we consider the output test (3.15), the input test (3.14) can be treated in a symmetric way. Assume that the considered disjunctive set I contains n activities and that these activities are ordered according to non-decreasing release dates $r_1 \leq r_2 \leq \ldots \leq r_n$.

For each activity i we try to find subsets $\Omega \subseteq I$ with $i \notin \Omega$ such that the inequality in (3.15) is satisfied. In the following we will show that not all possible $O(2^n)$ subsets $\Omega \subseteq I$ have to be considered as candidates in (3.15).

Assume that the inequality in (3.15) holds for some set Ω and an activity $i \notin \Omega$. Let $k \in \Omega$ be an activity with $d_k = \max\limits_{\mu \in \Omega} d_\mu$ and let $l \in \Omega \cup \{i\}$ be the activity with smallest index such that $r_l = \min\limits_{\nu \in \Omega \cup \{i\}} r_\nu$ (which implies $l \leq i$). If the inequality in (3.15) holds for Ω, then this condition is also satisfied for the (possibly larger) set $\Omega_{l,k} \setminus \{i\}$ with

$$\Omega_{l,k} := \{\mu \mid \mu \geq l \text{ and } d_\mu \leq d_k\}$$

because $\Omega \subseteq \Omega_{l,k}$ due to the definition of k and l. Thus, it is sufficient to consider only sets $\Omega_{l,k}$ as candidates.

In the following we assume that activities i and k are fixed. We search for an index $l \leq i$ such that the set $\Omega_{l,k} \setminus \{i\}$ satisfies the inequality in (3.15), i.e.

$$r_l + P(\Omega_{l,k}) + p_i > d_k. \tag{3.16}$$

Let $P_{l,k} := P(\Omega_{l,k})$, $\Delta_{\lambda,k} := \max\limits_{\nu \geq \lambda} \{r_\nu + P_{\nu,k}\}$, and $H_{i,k} := \max\limits_{\nu < i} \{r_\nu + P_{\nu,k}\}$. We distinguish two cases.

- Case 1: $d_i \leq d_k$
  Then we have $i \in \Omega_{l,k}$ for all $l \leq i$, i.e. (3.16) can be written as $r_l + P_{l,k} > d_k$. But, if this condition holds, then no feasible schedule exists since the activities in the set $\Omega_{l,k}$ cannot be scheduled in their time window $[r_l, d_k]$. On the other hand, if $r_l + P_{l,k} > d_k$ for an index $l > i$ holds, also infeasibility can be stated. Thus, we may check the general infeasibility test

$$\Delta_{1,k} = \max_{l \geq 1} \{r_l + P_{l,k}\} > d_k. \tag{3.17}$$

- Case 2: $d_i > d_k$
  Then we have $i \notin \Omega_{l,k}$ for all l. Due to $l \leq i$ two subcases for l are distinguished:

Subcase 2.1: $l = i$
If for $l = i$ the condition

$$r_i + P_{i,k} + p_i > d_k \tag{3.18}$$

is satisfied, then $\Omega_{i,k} \rightarrow i$ is implied. Thus, i cannot start before all activities in the set $\Omega_{i,k}$ are completed, i.e. we may require

$$r_i \geq \max_{\nu \geq i} \{r_\nu + P_{\nu,k}\} = \Delta_{i,k},$$

since due to $\Omega_{\nu,k} \subseteq \Omega_{i,k}$ for all $\nu \geq i$ the right hand side defines a lower bound for the completion time of $\Omega_{i,k}$. Thus, we may update the release date of i to

$$r'_i := \max\{r_i, \Delta_{i,k}\}. \tag{3.19}$$

Subcase 2.2: $l < i$
All cases $l < i$ are covered by checking the condition

$$H_{i,k} + p_i = \max_{l<i} \{r_l + P_{l,k}\} + p_i > d_k. \tag{3.20}$$

If an index $l < i$ of a set $\Omega_{l,k}$ satisfying (3.20) is found, we have $\Omega_{l,k} \rightarrow i$ and may require

$$r_i \geq \max_{\nu \geq l} \{r_\nu + P_{\nu,k}\} = \Delta_{l,k}.$$

We will show that we do not have to determine the index l explicitly, but that we can set

$$r'_i := \max\{r_i, \Delta_{1,k}\}. \tag{3.21}$$

Let $l < i$ be the index for which $H_{i,k} = r_l + P_{l,k}$ holds. Due to $r_l + P_{l,k} \geq r_\lambda + P_{\lambda,k}$ for all $1 \leq \lambda < l$ we have $\Delta_{l,k} = \max_{\nu \geq l} \{r_\nu + P_{\nu,k}\} = \max_{\lambda \geq 1} \{r_\lambda + P_{\lambda,k}\} = \Delta_{1,k}$.

The preceding discussions provide the algorithm shown in Figure 3.13.

In this algorithm condition (3.18) is checked in Step 14, condition (3.20) is checked in Step 16. Note that the infeasibility test (3.17) is also valid in the case $d_i > d_k$. This general check is performed in Step 10.

For fixed k the values $P_{n,k}, \ldots, P_{1,k}$; $\Delta_{n,k}, \ldots, \Delta_{1,k}$ and $H_{1,k}, \ldots, H_{n,k}$ can be computed in $O(n)$ time, since starting with the initial values $P_{n+1,k} = \Delta_{n+1,k} = H_{0k} := 0$ the values

$$P_{\lambda,k} = \sum_{\{\mu | \mu \geq \lambda, d_\mu \leq d_k\}} p_\mu = \begin{cases} P_{\lambda+1,k} + p_\lambda, & \text{if } d_\lambda \leq d_k \\ P_{\lambda+1,k}, & \text{otherwise} \end{cases} \quad \text{for } \lambda = n, \ldots, 1,$$

$$\Delta_{\lambda,k} := \max_{\nu \geq \lambda} \{r_\nu + P_{\nu,k}\} = \max\{\Delta_{\lambda+1,k}, r_\lambda + P_{\lambda,k}\} \quad \text{for } \lambda = n, \ldots, 1,$$

and

$$H_{\lambda,k} := \max_{\nu < \lambda} \{r_\nu + P_{\nu,k}\} = \max\{H_{\lambda-1,k}, r_{\lambda-1} + P_{\lambda-1,k}\} \quad \text{for } \lambda = 1, \ldots, n$$

```
Algorithm Output Test
 1.  Sort the activities such that r_1 ≤ ... ≤ r_n;
 2.  FOR i := 1 TO n DO
 3.      r'_i := r_i;
 4.  FOR k := 1 TO n DO
 5.      Compute P_{n,k}, ..., P_{1,k};
 6.      Compute Δ_{n,k}, ..., Δ_{1,k};
 7.      Compute H_{1,k}, ..., H_{n,k};
 8.  ENDFOR
 9.  FOR k := 1 TO n DO
10.      IF Δ_{1,k} > d_k THEN
11.         EXIT (no feasible schedule exists)
12.      FOR i := 1 TO n DO
13.          IF d_i > d_k THEN
14.              IF r_i + P_{i,k} + p_i > d_k THEN
15.                  r'_i := max{r'_i, Δ_{i,k}};
16.              IF H_{i,k} + p_i > d_k THEN
17.                  r'_i := max{r'_i, Δ_{1,k}};
18.          ENDIF
19.  ENDFOR
20.  FOR i := 1 TO n DO
21.      r_i := r'_i;
```

Figure 3.13: Output Test

can be computed in constant time from the previous values $P_{\lambda+1,k}$, $\Delta_{\lambda+1,k}$ and $H_{\lambda-1,k}$, respectively. Furthermore, sorting the activities according to non-decreasing release dates can be done in $O(n \log n)$ time. Therefore, the presented implementation of the output test runs in $O(n^2)$ time.

Example 3.2: Consider the following example with $n = 5$ activities:

i	1	2	3	4	5
r_i	0	1	2	4	4
p_i	2	1	3	2	1
d_i	4	5	10	7	8

Activity $i = 3$ has to be last in the set $\{1, 2, 3\}$ since for $\Omega := \Omega_{1,2} = \{1, 2\}$ we have

$$\max_{\mu \in \Omega} d_\mu - \min_{\nu \in \Omega \cup \{i\}} r_\nu = d_2 - r_1 = 5 - 0 = 5 < P(\Omega) + p_i = 3 + 3 = 6$$

(cf. also Figure 3.14).

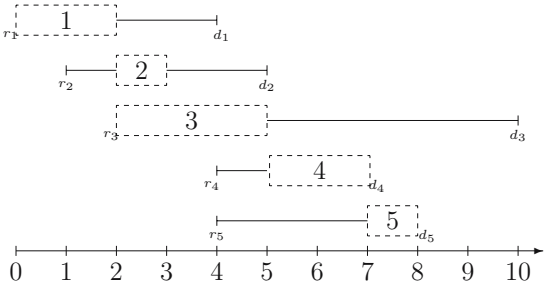


Figure 3.14: Time windows for the activities from Example 3.2

Algorithm `Output Test` detects this situation since for $i = 3$, $k = 2$ we have $d_i = 10 > d_k = 5$ and the check of condition (3.20) in Step 16 gives

$$H_{i,k}+p_i = H_{3,2}+p_3 = \max_{\nu<3}\{r_\nu+P_{\nu,k}\}+p_3 = r_1+P_{1,2}+p_3 = 0+3+3 = 6 > d_k = 5,$$

i.e. it is satisfied for $l = 1 < i = 3$. Thus, the release date of activity 3 may be updated to $r'_3 = \Delta_{1,k} = \Delta_{1,2} = r_1 + P_{1,2} = 3$.

Afterwards, we may also deduce that activity $i = 3$ has to be last in the set $\{3, 4, 5\}$ since for $\Omega := \Omega_{4,5} = \{4, 5\}$ we have

$$\max_{\mu\in\Omega} d_\mu - \min_{\nu\in\Omega\cup\{i\}} r_\nu = d_5 - r'_3 = 8 - 3 = 5 < P(\Omega) + p_i = 3 + 3 = 6.$$

Algorithm `Output Test` detects this situation in a second run since for $i = 3$, $k = 5$ we have $d_i = 10 > d_k = 8$ and the check of condition (3.18) in Step 14 with the new head $r'_3 = 3$ gives

$$r_i + P_{i,k} + p_i = r'_3 + P_{3,5} + p_3 = 3 + 3 + 3 = 9 > d_k = 8,$$

i.e. it is satisfied for $l = 3 = i$. Thus, the release date of activity 3 may be updated to $r''_3 = \Delta_{i,k} = \Delta_{3,5} = r_4 + P_{4,5} = 7$. □

Input/Output negation tests

In the preceding discussions we have focused on determining whether an activity i **must** be processed first (or last) in a certain set J of activities (i.e. i is input or output of J). In the following we consider the converse situation, i.e. we are interested in determining whether an activity i **cannot** be processed first (or last) in a certain set of activities. To derive an algorithm which performs such input negation and output negation tests in a systematic way, we consider for all $i \in I$ and subsets $\Omega \subseteq I$ with $i \notin \Omega$ the following implications:

$$\max_{\mu\in\Omega} d_\mu - r_i < P(\Omega) + p_i \Rightarrow i \nrightarrow \Omega, \tag{3.22}$$

i.e. i cannot be scheduled first in $\Omega \cup \{i\}$. Symmetrically,

$$d_i - \min_{\nu \in \Omega} r_\nu < P(\Omega) + p_i \Rightarrow \Omega \nrightarrow i, \tag{3.23}$$

i.e. i cannot be scheduled last in $\Omega \cup \{i\}$.

In the case (3.22) activity i cannot be started before the job scheduled first in Ω is completed, i.e. for its head r_i we have

$$r_i \geq \min_{\nu \in \Omega} \{r_\nu + p_\nu\}. \tag{3.24}$$

In the case (3.23) activity i cannot be completed later than the job scheduled last in Ω is started, i.e. for its deadline d_i we have

$$d_i \leq \max_{\mu \in \Omega} \{d_\mu - p_\mu\}. \tag{3.25}$$

In the following we consider the input negation test (3.22), the output negation test (3.23), can be treated in a symmetric way. Assume that the considered disjunctive set I contains n activities and that these activities are ordered according to non-decreasing deadlines $d_1 \leq d_2 \leq \ldots \leq d_n$.

For each activity i we try to find subsets $\Omega \subseteq I$ with $i \notin \Omega$ such that the inequality in (3.22) is satisfied. As in the output test we do not have to consider all possible subsets $\Omega \subseteq I$. Assume that the inequality in (3.22) holds for some set Ω with $i \notin \Omega$ and that r_i according to (3.24) may be updated to $r_j + p_j$ for an activity $j \in \Omega$ with $r_j + p_j = \min_{\nu \in \Omega} \{r_\nu + p_\nu\}$. Let $l \in \Omega$ be the activity with largest index such that $d_l = \max_{\nu \in \Omega} d_\nu$. Then the inequality in (3.22) is also satisfied for the set $\Omega_{j,l} \setminus \{i\}$ with

$$\Omega_{j,l} := \{\mu \mid \mu \leq l \text{ and } r_\mu + p_\mu \geq r_j + p_j\}$$

because $\Omega \subseteq \Omega_{j,l}$ due to the definition of j and l.

Let $P_{j,l} := P(\Omega_{j,l})$ if $j \leq l$ and $-\infty$ otherwise. Furthermore for $\lambda = 1, \ldots, n$ let $\Delta_{j,\lambda} := \min_{\mu \leq \lambda} \{d_\mu - P_{j,\mu}\}$.

In the following we assume that activities i and j are fixed. We search for an index l such that the set $\Omega_{j,l} \setminus \{i\}$ satisfies (3.22) or equivalently

$$r_i + p_i > d_l - P_{j,l}, \tag{3.26}$$

and the release date of i may be increased to $r_j + p_j = \min_{\nu \in \Omega_{j,l}} \{r_\nu + p_\nu\}$. We distinguish two cases.

- Case 1: $r_i + p_i < r_j + p_j$

  Then $i \notin \Omega_{j,l}$ for all $l = 1, \ldots, n$ and the best we can do is to check

  $$r_i + p_i > \min_{1 \leq l \leq n} \{d_l - P_{j,l}\} = \Delta_{j,n}. \tag{3.27}$$

  If (3.27) is satisfied, we can update the release date of i to

  $$r_i' := \max\{r_i, r_j + p_j\}.$$

- Case 2: $r_i + p_i \geq r_j + p_j$

  For $l < i$ we have $i \notin \Omega_{j,l}$. We cover all cases with $l < i$ by checking the condition

  $$r_i + p_i > \min_{l \leq i-1} \{d_l - P_{j,l}\} = \Delta_{j,i-1}. \tag{3.28}$$

  For $l \geq i$ we have $i \in \Omega_{j,l}$, i.e. we have to check $r_i > d_l - P_{j,l}$. Instead of testing $r_i > \min_{i \leq l} \{d_l - P_{j,l}\}$, we may check the stronger condition

  $$r_i > \min_{1 \leq l \leq n} \{d_l - P_{j,l}\} = \Delta_{j,n},$$

  which is less time-consuming (since we do not have to find l explicitly). This is correct since if for some $l < i$ the condition $r_i > d_l - P_{j,l}$ holds, then also $r_i + p_i > d_l - P_{j,l}$ is valid, which implies that also condition (3.28) is satisfied.

  Thus, we may cover all cases $l < i$ and $l \geq i$ by checking whether

  $$r_i + p_i > \Delta_{j,i-1} \quad \text{or} \quad r_i > \Delta_{j,n} \tag{3.29}$$

  holds. Again, if this condition holds, we may update the release date of i to $r'_i := \max\{r_i, r_j + p_j\}$.

```
Algorithm Input Negation Test
 1.  Sort the activities such that d_1 ≤ ... ≤ d_n;
 2.  FOR i := 1 TO n DO
 3.      r'_i := r_i;
 4.  FOR j := 1 TO n DO
 5.      Compute Δ_{j,1}, ..., Δ_{j,n};
 6.      FOR i := 1 TO n WITH i ≠ j DO
 7.          IF r_i + p_i < r_j + p_j THEN
 8.              IF r_i + p_i > Δ_{j,n} THEN
 9.                  r'_i := max{r'_i, r_j + p_j};
10.              ENDIF
11.          ELSE
12.              IF r_i + p_i > Δ_{j,i-1} OR r_i > Δ_{j,n} THEN
13.                  r'_i := max{r'_i, r_j + p_j};
14.          ENDIF
15.  ENDFOR
16.  FOR i := 1 TO n DO
17.      r_i := r'_i;
```

Figure 3.15: Input Negation Test

The preceding discussions provide the algorithm shown in Figure 3.15. In this algorithm condition (3.27) is checked in Step 8, condition (3.29) is checked in Step 12.

For fixed j the values $\Delta_{j,1}, \ldots, \Delta_{j,n}$ can be computed in $O(n)$ time, since starting with the initial values $P_{j,0} := 0,\ \Delta_{j,0} := \infty$ for $\lambda = 1, \ldots, n$ the values

$$P_{j,\lambda} = \sum_{\{\mu|\mu\leq\lambda, r_\mu+p_\mu\geq r_j+p_j\}} p_\mu = \begin{cases} P_{j,\lambda-1} + p_\lambda, & \text{if } r_\lambda + p_\lambda \geq r_j + p_j \\ P_{j,\lambda-1}, & \text{otherwise} \end{cases}$$

and

$$\Delta_{j,\lambda} := \min_{\mu\leq\lambda} \{d_\mu - P_{j,\mu}\} = \min\{\Delta_{j,\lambda-1}, d_\lambda - P_{j,\lambda}\}$$

can be computed in constant time from the previous values $P_{j,\lambda-1}$ and $\Delta_{j,\lambda-1}$, respectively. Furthermore, sorting the activities according to non-decreasing deadlines can be done in $O(n \log n)$ time. Therefore, the presented implementation of the input negation test runs in $O(n^2)$ time.

Example 3.3: Consider the following example with $n = 3$ activities:

i	1	2	3
r_i	0	2	1
p_i	2	1	2
d_i	5	5	10

Activity $i = 3$ cannot be first in the set $\{1, 2, 3\}$ since for $\Omega := \Omega_{1,2} = \{1, 2\}$ we have

$$\max_{\mu\in\Omega} d_\mu - r_i = 5 - 1 = 4 < P(\Omega) + p_i = 3 + 2 = 5$$

(cf. also Figure 3.16).

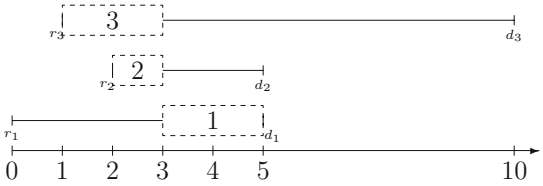


Figure 3.16: Time windows for the activities from Example 3.3

With

$\Omega_{1,1} = \{1\}$, $\Delta_{1,1} = d_1 - p_1 = 3$
$\Omega_{1,2} = \{1, 2\}$, $\Delta_{1,2} = \min\{\Delta_{1,1}, d_2 - P_{1,2}\} = \min\{3, 2\} = 2$
$\Omega_{1,3} = \{1, 2, 3\}$, $\Delta_{1,3} = \min\{\Delta_{1,2}, d_3 - P_{1,3}\} = \min\{2, 5\} = 2$
$\Omega_{2,1} = \Omega_{3,1} = \emptyset$, $\Delta_{2,1} = \Delta_{3,1} = \infty$
$\Omega_{2,2} = \Omega_{3,2} = \{2\}$, $\Delta_{2,2} = \Delta_{3,2} = d_2 - p_2 = 4$
$\Omega_{2,3} = \Omega_{3,3} = \{2, 3\}$, $\Delta_{2,3} = \Delta_{3,3} = \min\{\Delta_{2,2}, d_3 - P_{2,3}\} = \min\{4, 7\} = 4$

Algorithm `Input Negation Test` detects this situation since for $j = 1$ and $i = 3$ we have $r_i + p_i = 3 \geq r_j + p_j = 2$ and the check of condition (3.29) in Step 12 gives

$$r_i + p_i = 3 > \Delta_{j,i-1} = \Delta_{1,2} = 2.$$

Thus, the release date of activity 3 may be updated to $r'_3 = r_1 + p_1 = 2$. □

Input-or-Output test

The input-or-output test is weaker than both the input and the output test (i.e. it is applicable more often, but its conclusion is weaker). At first we consider the test for activities $i, j \in I$ with $i \neq j$. If then a subset $\Omega \subseteq I$ with $i, j \notin \Omega$ exists and the condition

$$\max_{\mu \in \Omega \cup \{i\}} d_\mu - \min_{\nu \in \Omega \cup \{j\}} r_\nu < P(\Omega \cup \{i, j\}) \tag{3.30}$$

holds, then activity i has to be scheduled first in $\Omega \cup \{i, j\}$ or activity j has to be scheduled last in $\Omega \cup \{i, j\}$, i.e. i is input for $\Omega \cup \{i, j\}$ or j is output for $\Omega \cup \{i, j\}$. In this situation the conjunction $i \rightarrow j$ may be introduced. Furthermore, we may improve the head of j according to $r'_j := \max\{r_j, r_i + p_i\}$ and the deadline of i according to $d'_i := \min\{d_i, d_j - p_j\}$.

Assume that the activities are ordered according to non-increasing release dates $r_1 \geq r_2 \geq \ldots \geq r_n$. For each pair of activities i, j with $i \neq j$ we try to find subsets $\Omega \subseteq I$ with $i, j \notin \Omega$ such that the inequality in (3.30) is satisfied.

Assume that (3.30) holds for some set Ω and activities $i, j \notin \Omega$. Let $l \in \Omega \cup \{i\}$ be an activity with $d_l = \max_{\mu \in \Omega \cup \{i\}} d_\mu$ and let $k \in \Omega \cup \{j\}$ be the activity with largest index such that $r_k = \min_{\nu \in \Omega \cup \{j\}} r_\nu$.

We may assume $d_j > d_l$ since otherwise the input test condition

$$\max_{\mu \in \Omega \cup \{i,j\}} d_\mu - \min_{\nu \in \Omega \cup \{j\}} r_\nu < P(\Omega \cup \{i, j\})$$

is satisfied for the set $\Omega' := \Omega \cup \{j\}$ and the activity i. Such situations may be discovered by the input test. Symmetrically, we may assume $r_i < r_k$.

If (3.30) holds for Ω, then this condition is also satisfied for the set $\Omega_{k,l}$ with

$$\Omega_{k,l} := \{\mu \mid \mu \leq k \text{ and } d_\mu \leq d_l\}$$

because $\Omega \subseteq \Omega_{k,l}$ due to the definition of k and l. Thus, it is sufficient to consider only sets $\Omega_{k,l}$ as candidates. Note that due to $r_i < r_k$ and $d_j > d_l$ we have $i, j \notin \Omega_{k,l}$.

In the following we assume that activities i and j with $i \neq j$ are fixed. Let π be a permutation in which the activities are ordered according to non-decreasing deadlines $d_{\pi_1} \leq d_{\pi_2} \leq \ldots \leq d_{\pi_n}$ and let h be the index of j in π, i.e. $j = \pi_h$. We search for an index $k < i$ with $r_k > r_i$ and an index $\lambda < h$ such that for $l := \pi_\lambda$ we have $d_l < d_j$ and the set $\Omega_{k,l}$ satisfies (3.30).

Let $P_{k,l} := P(\Omega_{k,l})$. Then condition (3.30) for the set $\Omega_{k,l}$ is equivalent to

$$\delta_{k,l} := d_l - r_k - P_{k,l} < p_i + p_j. \tag{3.31}$$

We cover all cases for k and l in (3.31) by checking

$$\Delta_{i,h} := \min_{\substack{k<i \\ \lambda<h}} \{\delta_{k,\pi_\lambda} \mid r_k > r_i, d_{\pi_\lambda} < d_{\pi_h}\} < p_i + p_{\pi_h}. \tag{3.32}$$

If this condition holds, we may introduce the conjunction $i \rightarrow j = \pi_h$.

It remains to show how the values $\delta_{k,l}$ and $\Delta_{i,h}$ can be calculated in an efficient way. For fixed l the values $P_{1,l}, \ldots, P_{n,l}$ can be computed in $O(n)$ time, since starting with the initial values $P_{0,l} := 0$ each of the values

$$P_{k,l} = \sum_{\{\mu | \mu \leq k, d_\mu \leq d_l\}} p_\mu = \begin{cases} P_{k-1,l} + p_k, & \text{if } d_k \leq d_l \\ P_{k-1,l}, & \text{otherwise} \end{cases} \quad \text{for } k = 1, \ldots, n$$

can be computed in constant time from the previous values $P_{k-1,l}$. Thus, all values $P_{k,l}$ and all values $\delta_{k,l} := d_l - r_k - P_{k,l}$ for $k, l = 1, \ldots, n$ can be computed in $O(n^2)$.

Starting with the initial values $\Delta_{1,h} := \infty$ for $h = 1, \ldots, n$ and $\Delta_{i,1} := \infty$ for $i = 1, \ldots, n$ due to $r_{i-1} \geq r_i$ and $d_{\pi_{h-1}} \leq d_{\pi_h}$ for $i, h = 2, \ldots, n$ we have

$$\Delta_{i,h} := \min \{\Delta_{i-1,h}, \Delta_{i,h-1}, \Delta'_{i-1,h-1}\} \tag{3.33}$$

with

$$\Delta'_{i-1,h-1} = \begin{cases} \delta_{i-1,\pi_{h-1}}, & \text{if } r_{i-1} > r_i, d_{\pi_{h-1}} < d_{\pi_h} \\ \infty, & \text{otherwise} \end{cases} .$$

Thus, the values $\Delta_{i,h}$ can be calculated diagonalwise by the recursion (3.33) for all $i, h \geq 2$ with $i + h = \beta$ for $\beta = 4, \ldots, 2n$ (cf. Figure 3.17).

	1	2	$\cdots$	$h-1$	h	$\cdots$	n
1	∞	∞	$\cdots$	∞	∞	$\cdots$	∞
2	∞						
$\vdots$	$\vdots$						
$i-1$	∞				$\circ$		
i	∞			$\circ$	$\otimes$		
$\vdots$	$\vdots$						
n	∞						

Figure 3.17: Calculation of the $\Delta_{i,h}$-values

Therefore, also all $\Delta_{i,h}$-values can be computed in $O(n^2)$. Furthermore, sorting the activities according to non-increasing release dates and non-decreasing deadlines can be done in $O(n \log n)$ time. Hence, the presented implementation of the input-or-output test runs in $O(n^2)$ time. The preceding discussions are summarized in the algorithm in Figure 3.18.

Example 3.4: Consider the following example with $n = 4$ activities (cf. also Figure 3.19):

i	1	2	3	4
r_i	2	2	1	0
p_i	3	2	1	3
d_i	10	9	9	8

```
Algorithm Input-or-Output Test
 1.  Sort the activities such that r_1 ≥ ... ≥ r_n;
 2.  Sort the activities such that d_{π_1} ≤ ... ≤ d_{π_n};
 3.  FOR l := 1 TO n DO
 4.      FOR k := 1 TO n DO
 5.          Compute P_{k,l} and δ_{k,l};
 6.  FOR i := 1 TO n DO
 7.      Δ_{i,1} := ∞; Δ_{1,i} := ∞;
 8.  FOR β := 4 TO 2n DO
 9.      FOR ALL i,h = 2,...,n with i + h = β DO
10.          Compute Δ_{i,h};
11.  FOR i := 1 TO n DO
12.      FOR h := 1 TO n DO
13.          IF i ≠ π_h AND Δ_{i,h} < p_i + p_{π_h} THEN
14.              Set i → π_h;
```

Figure 3.18: Input-or-Output Test

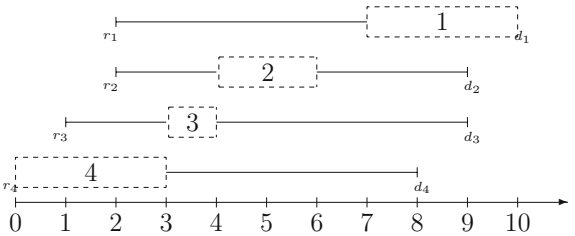


Figure 3.19: Time windows for the activities from Example 3.4

Activity $i = 4$ has to be scheduled before activity $j = 1$ since for the set $\Omega = \{2, 3\}$ we have

$$\max_{\mu \in \Omega \cup \{i\}} d_\mu - \min_{\nu \in \Omega \cup \{j\}} r_\nu = d_2 - r_3 = 9 - 1 = 8 < P(\Omega \cup \{i, j\}) = 9.$$

Algorithm `Input-or-Output Test` calculates the permutation $\pi = (4, 3, 2, 1)$ and the values

$$\begin{aligned}
&\Delta'_{1,1} = \Delta'_{1,2} = \Delta'_{1,3} = \infty \quad \text{due to } r_1 = r_2 \\
&\Delta'_{2,2} = \Delta'_{3,2} = \infty \quad \text{due to } d_{\pi_2} = d_{\pi_3} \\
&\Delta'_{2,1} = \delta_{2,\pi_1} = \delta_{2,4} = d_4 - r_2 - P_{2,4} = 8 - 2 - 0 = 6 \\
&\Delta'_{3,1} = \delta_{3,\pi_1} = \delta_{3,4} = d_4 - r_3 - P_{3,4} = 8 - 1 - 0 = 7 \\
&\Delta'_{2,3} = \delta_{2,\pi_3} = \delta_{2,2} = d_2 - r_2 - P_{2,2} = 9 - 2 - p_2 = 5 \\
&\Delta'_{3,3} = \delta_{3,\pi_3} = \delta_{3,2} = d_2 - r_3 - P_{3,2} = 9 - 1 - p_2 - p_3 = 5.
\end{aligned}$$

Due to the recursion (3.33) we get

$\Delta_{i,h}$	$h=1$	$h=2$	$h=3$	$h=4$
$i=1$	∞	∞	∞	∞
$i=2$	∞	∞	∞	∞
$i=3$	∞	6	6	5
$i=4$	∞	6	6	$\boxed{5}$

$p_i + p_{\pi_h}$	$h=1$	$h=2$	$h=3$	$h=4$
$i=1$	–	–	–	–
$i=2$	–	3	–	5
$i=3$	–	–	3	4
$i=4$	–	4	5	$\boxed{6}$

where in the right table a '–' in row i and column h indicates that $i = \pi_h$. Comparing the entries in the two tables shows that condition (3.32) in Step 13 is satisfied for

$$\Delta_{4,4} = 5 < p_4 + p_{\pi_4} = 6,$$

i.e. it is satisfied for $i = 4$, $h = 4$ (or $j = \pi_4 = 1$) implying the conjunction $4 \to 1$. □

The input-or-output test for $i = j$ can be formulated as follows. If for an activity $i \in I$ and a subset $\Omega \subseteq I$ with $i \notin \Omega$ the condition

$$\max_{\mu\in\Omega} d_\mu - \min_{\nu\in\Omega} r_\nu < P(\Omega \cup \{i\}) \tag{3.34}$$

holds, then activity i has to be scheduled first or last in $\Omega \cup \{i\}$, i.e. i is input or output for $\Omega \cup \{i\}$.

As above, it is sufficient to check (3.34) for sets $\Omega_{k,l}$ with $\Omega_{k,l} := \{\mu \mid \mu \leq k \text{ and } d_\mu \leq d_l\}$ and $r_i < r_k$ and $d_i > d_l$. Then (3.34) is equivalent to

$$\delta_{k,l} := d_l - r_k - P_{k,l} < p_i, \tag{3.35}$$

i.e. these checks can easily be integrated into the above algorithm.

The presented tests can be found with different names in the literature. Since in Theorem 3.1 for disjunctive sets the capacity of certain intervals is considered, all tests of this type are also called "disjunctive interval consistency tests". Since additional conjunctions are derived by the first three tests, they are also termed "edge finding" procedures. In branch-and-bound algorithms where in the branching step disjunctive edges $i - j$ are oriented into $i \to j$ or $j \to i$, the tests may be used to immediately select the orientation of some other disjunctive edges. Therefore, this process has also been called "immediate selection".

All different interval consistency tests are summarized in Table 3.1. Furthermore, the complexities for the best known implementations are listed, where again n denotes the number of activities in the considered disjunctive set I. Note that for a test not only its complexity for a single run is important, but also the number of times it has to be applied. As mentioned before, usually a test is applied in several iterations until no more constraints can be deduced. Thus, an $O(n^2)$-algorithm which uses less iterations may be more effective than an $O(n \log n)$-algorithm which uses more iterations.

Test	$J \setminus J'$	$J \setminus J''$	conclusion	complexity
input	$J \setminus \{i\}$	J	$i \rightarrow J \setminus \{i\}$	$O(n \log n)$
output	J	$J \setminus \{i\}$	$J \setminus \{i\} \rightarrow i$	$O(n \log n)$
input-or-output	$J \setminus \{i\}$	$J \setminus \{j\}$	$i \rightarrow J \setminus \{i\} \vee J \setminus \{j\} \rightarrow j$	$O(n^2)$
input negation	$\{i\}$	$J \setminus \{i\}$	$i \nrightarrow J \setminus \{i\}$	$O(n \log n)$
output negation	$J \setminus \{i\}$	$\{i\}$	$J \setminus \{i\} \nrightarrow \{i\}$	$O(n \log n)$

Table 3.1: Summary of disjunctive interval consistency tests

There are different ways to combine the described constraint propagation techniques in one constraint propagation procedure. In order to apply the interval consistency tests at first disjunctive sets have to be determined.

Unfortunately, the number of all possible disjunctive sets (cliques) may be quite large. Thus, usually only some (maximal) cliques are generated heuristically. A clique I is called maximal if adding an activity $i \notin I$ to I does not lead to another (larger) clique. Maximal cliques $I_1, \ldots, I_q$ which cover the whole set of activities (i.e. with $I_1 \cup I_2 \cup \ldots \cup I_q = \{1, \ldots, n\}$) may be determined as follows.

To build I_1, we start with some activity i_1 and add an activity i_2 which is incompatible to i_1 (i.e. with $i_2 - i_1 \in D$ or $i_1 \rightarrow i_2 \in C$ or $i_2 \rightarrow i_1 \in C$). Then we add some activity i_3 which is incompatible to both i_1 and i_2, etc. This process stops if no activity exists which is incompatible to all $i \in I_1$. To build I_2, we start with some activity not belonging to I_1 and apply the same procedure. Note that $i \in I_1$ may also be added to the clique I_2. If $I_1, \ldots, I_h$ are built and some activity $i \notin I_1 \cup I_2 \cup \ldots \cup I_h$ is left, then we start I_{h+1} with i. The procedure stops if all activities are covered by the constructed cliques. To all these maximal cliques the described tests may iteratively be applied, where only sets I_ν with $|I_\nu| \geq 2$ are used. The tests are stopped if the time windows of the activities are no longer reduced (i.e. a fixpoint is reached).

Starting with the initial set D_0 of disjunctions and with an initial SSD-matrix $d = (d_{ij})$ defined by C_0, N_0 and given heads and tails we repeat the constrained propagation procedure to all disjunctive sets until we detect infeasibility or the SSD-matrix does not change any more. Of course, additionally also symmetric triples and its extensions may be considered in the procedure.

The interval consistency tests can also be seen as tests in which activity sequences with a special property are studied (e.g. an activity which is (or is not) scheduled first or last). Then the objective is to find a contradiction to this hypothesis implying that the reversed property must hold. A similar approach is the so-called **shaving** technique. In this method a hypothetical starting time of an activity is fixed and consequences of this decision are propagated. If afterwards inconsistency is detected, the hypothesis is falsified and the opposite

constraint can be fixed. For example, if the hypothetical constraint $S_i > t_i$ with $t_i \in [r_i, d_i]$ leads to a contradiction, the time window of activity i can be reduced to $[r_i, t_i]$.

Although the presented tests may deduce several new constraints and reduce the search space, in practice one often has to find a tradeoff between the usefulness of the tests and the time spent for them. Especially, the shaving techniques are very time consuming if they are applied to every activity and every possible starting time in its time window.

3.2.5 Cumulative resources

Recall that renewable resources k for which $R_k > 1$ holds, are called **cumulative resources**. For these resources the tests of the previous section are not valid. However, some concepts may be generalized introducing the term "work" or "energy".

For this purpose we consider a fixed resource k. Let I_k be the set of activities i with $r_{ik} > 0$. Then $w_i := r_{ik}p_i$ is the **work** needed to process activity i. For $J \subseteq I_k$ we define $W(J) := \sum_{i \in J} w_i$. Since in an interval $[t_1, t_2]$ with $t_1 < t_2$ the work $R_k(t_2 - t_1)$ is available, similarly to Theorem 3.1 we have

Theorem 3.2 Let $J', J'' \subset J \subseteq I_k$. If

$$R_k \cdot \max_{\substack{\nu \in J \setminus J' \\ \mu \in J \setminus J''}} (d_\mu - r_\nu) = R_k \cdot (\max_{\mu \in J \setminus J''} d_\mu - \min_{\nu \in J \setminus J'} r_\nu) < W(J), \tag{3.36}$$

then in J an activity from J' must start first or an activity from J'' must end last. □

In contrast to (3.12), in (3.36) we cannot impose $\nu \neq \mu$ because in the non-disjunctive case an activity which starts first may also complete last.

This theorem can be used to derive tests similar to those listed in Table 3.1. The meaning of conclusions such as $J \setminus \{i\} \to i$ or $i \to J \setminus \{i\}$ is that i must complete after (start before) activities in $J \setminus \{i\}$. In contrast to the disjunctive case this does not imply that it must also start after (complete before) $J \setminus \{i\}$.

3.2.6 Constraint propagation for the multi-mode case

In this subsection we shortly describe how some constraint propagation techniques for the RCPSP can be extended to the multi-mode case.

At first some superfluous modes and resources may be eliminated in a preprocessing step. A mode $m \in \mathcal{M}_i$ of an activity i is called **non-executable** if its resource requirements cannot be fulfilled in any feasible schedule, i.e. if

$r^{\rho}_{ikm} > R^{\rho}_k$ for a renewable resource $k \in \mathcal{K}^{\rho}$ or if $r^{\nu}_{ikm} + \sum_{\substack{j=1 \\ j \neq i}}^{n} \min_{\mu \in \mathcal{M}_j} \{r^{\nu}_{jk\mu}\} > R^{\nu}_k$ for a non-renewable resource $k \in \mathcal{K}^{\nu}$ holds.

A mode $m \in \mathcal{M}_i$ of an activity i is called **inefficient** if another mode $\mu \in \mathcal{M}_i$ for the same activity exists in which i does not need more resource units from each resource and does not have a larger duration than in mode m, i.e. $p_{i\mu} \leq p_{im}$, $r^{\rho}_{ik\mu} \leq r^{\rho}_{ikm}$ for all renewable resources $k \in \mathcal{K}^{\rho}$ and $r^{\nu}_{ik\mu} \leq r^{\nu}_{ikm}$ for all non-renewable resources $k \in \mathcal{K}^{\nu}$.

Finally, a non-renewable resource k is called **redundant** if $\sum_{j=1}^{n} \max_{\mu \in \mathcal{M}_j} \{r^{\nu}_{jk\mu}\} \leq R^{\nu}_k$ holds, i.e. resource k is sufficiently available independent of the assigned modes.

Obviously, non-executable and inefficient modes as well as redundant resources may be deleted from a multi-mode instance without changing the optimal solution.

Example 3.5: Consider a multi-mode project with $n = 4$ activities, where each activity may be processed in two different modes $m = 1, 2$. We are given three resources: one renewable resource 1 with capacity $R^{\rho}_1 = 4$ and two non-renewable resources 2,3 with $R^{\nu}_2 = 13$ and $R^{\nu}_3 = 14$. Furthermore, the activities $i = 1, \ldots, 4$ in modes $m = 1, 2$ have the following processing times p_{im} and resource requirements $r^{\rho}_{i1m}, r^{\nu}_{ikm}$ for $k = 2, 3$:

(i,m)	$(1,1)$	$(1,2)$	$(2,1)$	$(2,2)$	$(3,1)$	$(3,2)$	$(4,1)$	$(4,2)$
p_{im}	2	4	3	5	2	3	3	4
r^{ρ}_{i1m}	5	2	3	1	2	1	2	2
r^{ν}_{i2m}	2	4	3	2	8	2	3	1
r^{ν}_{i3m}	1	1	3	4	3	3	2	7

Due to $r^{\rho}_{111} = 5 > R^{\rho}_1 = 4$ mode 1 of activity 1 is non-executable with respect to the renewable resource 1 and can therefore be deleted from the input data. This induces that mode 1 of activity 3 becomes non-executable with respect to non-renewable resource 2 since

$$r^{\nu}_{321} + \sum_{j \neq 3} \min_{\mu \in \mathcal{M}_j} \{r_{j2\mu}\} = 8 + 4 + 2 + 1 = 15 > R^{\nu}_2 = 13.$$

After removing this mode, resource 2 becomes redundant and can also be deleted. Then mode 2 of activity 4 becomes inefficient and eleminating this mode causes that resource 3 also becomes redundant. Thus, the given instance may be reduced to the following instance with one renewable resource 1 with $R^{\rho}_1 = 4$ and no non-renewable resource (the mode numbers are adjusted):

(i,m)	$(1,1)$	$(2,1)$	$(2,2)$	$(3,1)$	$(4,1)$
p_{im}	4	3	5	3	3
r^{ρ}_{i1m}	2	3	1	1	2

□

The example shows that removing a non-executable or inefficient mode may cause a redundant resource, and deleting a redundant resource may lead to a new inefficient mode. Thus, the elimination steps may have to be iterated as follows:

1. Eliminate all non-executable modes (several passes through the activities may be necessary).
2. Eliminate redundant resources.
3. Eliminate all inefficient modes. If a mode was eliminated, go to Step 2.

In order to reduce a multi-mode situation to the classical single-mode case, for each activity i we define its minimal processing time and minimal resource requirements by

$$p_i := \min_{m\in\mathcal{M}_i}\{p_{im}\},\ r_{ik}^{\rho} := \min_{m\in\mathcal{M}_i}\{r_{ikm}^{\rho}\}\ \forall\ k\in\mathcal{K}^{\rho},\ r_{ik}^{\nu} := \min_{m\in\mathcal{M}_i}\{r_{ikm}^{\nu}\}\ \forall\ k\in\mathcal{K}^{\nu}.$$

Since these values are lower bounds for the actual mode-dependent values in any feasible schedule, all lower bound and constraint propagation methods for the single-mode case can be applied to these data. Furthermore, we define a distance matrix $d = (d_{ij})_{i,j\in V}$, where the entries d_{ij} define lower bounds for the differences of starting times $S_j - S_i$. For example, if the precedence relation $i \to j$ exists, we set $d_{ij} := \min_{m\in\mathcal{M}_i}\{p_{im}\}$. If additionally, mode-dependent start-start time-lags $d_{ij}^{m\mu}$ for activity i in mode m and activity j in mode μ are given, we may define d by

$$d_{ij} = \begin{cases} 0, & \text{if } i = j \\ \min\limits_{m\in\mathcal{M}_i}\min\limits_{\mu\in\mathcal{M}_j}\{d_{ij}^{m\mu}\}, & \text{if a time-lag between } i \text{ and } j \text{ exists} \\ p_i - T, & \text{otherwise} \end{cases} \tag{3.37}$$

where T denotes a given time horizon for the project. Let A be the set of all pairs (i,j) for which a time-lag $d_{ij}^{m\mu}$ for some modes $m\in\mathcal{M}_i, \mu\in\mathcal{M}_j$ exists.

Again, this matrix may be transitively adjusted by the Floyd-Warshall algorithm and heads and tails are given by d_{0i} and $d_{i,n+1}$, respectively. In order to strengthen these heads and tails, the modes may be taken into account more carefully.

For this purpose let r_i^m be a lower bound for the starting time of activity i if it is processed in mode $m\in\mathcal{M}_i$ (independent of the modes of the other activities). We must have $r_i^m \geq \min_{\mu\in\mathcal{M}_j}\{r_j^{\mu} + d_{ji}^{\mu m}\}$ for all other activities j since each j has to be processed in some mode $\mu\in\mathcal{M}_j$. Thus, the values r_i^m can be calculated by the recursion

$$r_i^m = \max_{\{j|(j,i)\in A\}}\ [\min_{\mu\in\mathcal{M}_j}\{r_j^{\mu} + d_{ji}^{\mu m}\}]. \tag{3.38}$$

Initially, we set $r_i^m := r_i = d_{0i}$ for all activities $i\in V$ and all modes $m\in\mathcal{M}_i$. Iteratively we scan all activities $i\in V$ in the order $1,2,\ldots,n,n+1,1,2,\ldots$ and

update the values r_i^m for all $m \in \mathcal{M}_i$ according to (3.38). This label-correcting procedure (cf. Section 2.2.2) stops if during one scan of all activities $i \in V$ no value is changed any more. Then $r_i := \min_{m \in \mathcal{M}_i} \{r_i^m\}$ provides a new head for activity i.

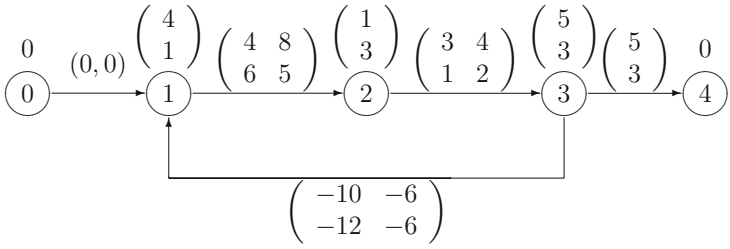
Calculating all these values can be done in pseudo-polynomial time. One scan through all activities needs $O(\sum_{i \in V} \sum_{\{j|(j,i) \in A\}} |\mathcal{M}_i||\mathcal{M}_j|)$ time and it can be shown that at most $O(\sum_{i \in V} |\mathcal{M}_i|T)$ iterations are needed.

Symmetrically, let q_i^m be a lower bound for the time we need after starting activity i if it is processed in mode m. These values can be calculated by the recursion

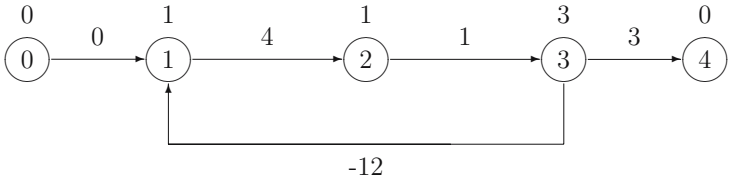
$$q_i^m = \max_{\{j|(i,j) \in A\}} [\min_{\mu \in \mathcal{M}_j} \{d_{ij}^{m\mu} + q_j^\mu\}], \tag{3.39}$$

where initially $q_i^m := q_i = d_{i,n+1}$ for all $i \in V$ and $m \in \mathcal{M}_i$. After no value is changed any more, $q_i := \min_{m \in \mathcal{M}_i} \{q_i^m\}$ defines a new tail for activity i. The values r_i and q_i may be used to adjust the corresponding entries d_{0i} and $d_{i,n+1}$ in the distance matrix d. Furthermore, for a given value T a deadline d_i for the completion time of activity i is given by $d_i := \max_{m \in \mathcal{M}_i} \{T - q_i^m + p_{im}\}$.

After the calculation of time windows $[r_i, d_i]$ additional methods of constraint propagation may be applied like in the single-mode case (e.g. by using the concept of symmetric triples or the disjunctive set tests).



(a) Multi-mode instance with time-lags and $n = 3$ activities



(b) Corresponding single-mode instance

Figure 3.20: Multi-mode instance and the corresponding single-mode instance

Example 3.6: Consider the multi-mode instance with time-lags $d_{ij}^{m\mu}$ and $n = 3$ activities shown in Figure 3.20(a), where each activity can be processed in two different modes. In the figure for each activity i the corresponding processing time vector $(p_{im}) = \begin{pmatrix} p_{i1} \\ p_{i2} \end{pmatrix}$ is given, furthermore, each arc $i \to j$ is weighted with the matrix $(d_{ij}^{m\mu})$ of start-start time-lags $\begin{pmatrix} d_{ij}^{11} & d_{ij}^{12} \\ d_{ij}^{21} & d_{ij}^{22} \end{pmatrix}$.

Associated with this multi-mode instance is the single-mode instance shown in Figure 3.20(b), where each node i is weighted with the minimal processing time p_i and each arc $i \to j$ is weighted with the minimal distance d_{ij}.

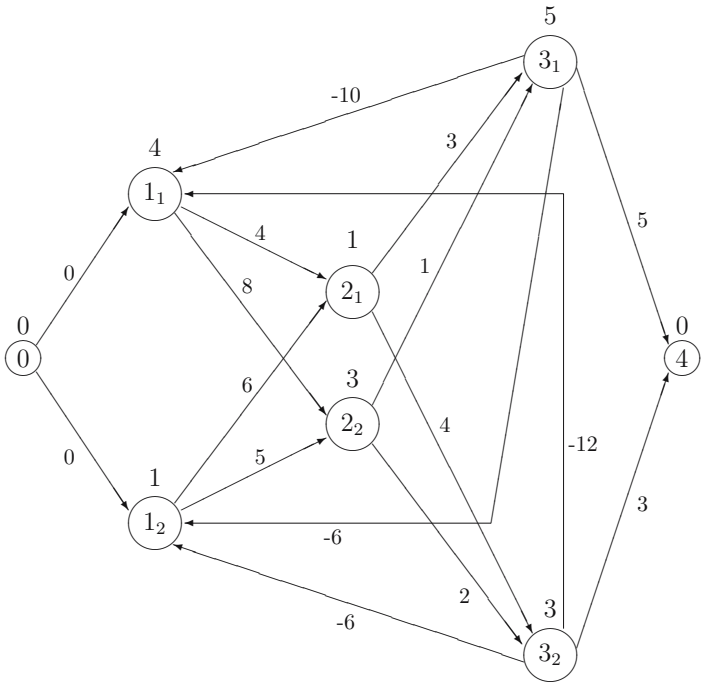


Figure 3.21: Expanded network for the multi-mode instance

In order to calculate the values r_i^m according to (3.38) we consider the network shown in Figure 3.21 with nodes i_m for all activity-mode combinations (i, m) and arcs $i_m \xrightarrow{d_{ij}^{m\mu}} j_\mu$.

During the first scan of the activities $1, 2, 3$ we calculate the values r_1^1, r_1^2, r_2^1, r_2^2, r_3^1, r_3^2. For the combinations $(1, m)$ with $m = 1, 2$ we get $r_1^m = r_0 + 0 = 0$, for

$(2, m)$ we get

$$\begin{aligned} r_2^1 &= \min\{r_1^1 + d_{12}^{11}, r_1^2 + d_{12}^{21}\} = \min\{4, 6\} = 4, \\ r_2^2 &= \min\{r_1^1 + d_{12}^{12}, r_1^2 + d_{12}^{22}\} = \min\{8, 5\} = 5, \end{aligned}$$

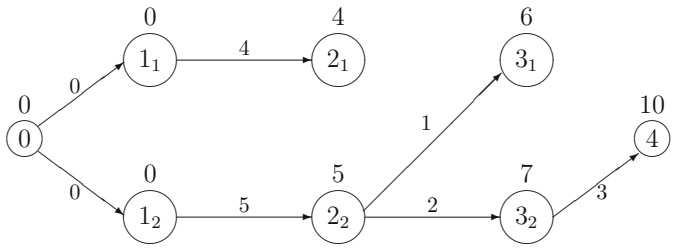
and for $(3, m)$ we get

$$\begin{aligned} r_3^1 &= \min\{r_2^1 + d_{23}^{11}, r_2^2 + d_{23}^{21}\} = \min\{4 + 3, 5 + 1\} = 6, \\ r_3^2 &= \min\{r_2^1 + d_{23}^{12}, r_2^2 + d_{23}^{22}\} = \min\{4 + 4, 5 + 2\} = 7. \end{aligned}$$

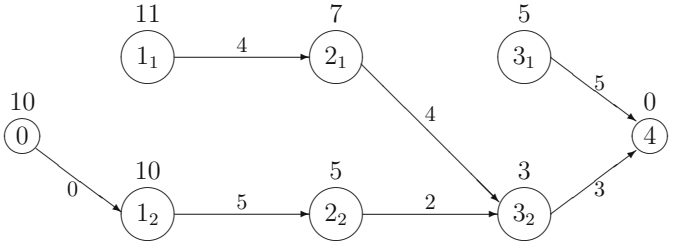
During the second scan of the activities we get

$$\begin{aligned} r_1^1 &= \max\{r_0 + 0, \min\{r_3^1 + d_{31}^{11}, r_3^2 + d_{31}^{21}\}\} = \max\{0, \min\{-4, -5\}\} = 0, \\ r_1^2 &= \max\{r_0 + 0, \min\{r_3^1 + d_{31}^{12}, r_3^2 + d_{31}^{22}\}\} = \max\{0, \min\{0, 1\}\} = 0, \end{aligned}$$

i.e. the values r_1^m are not changed in the second iteration. Also the other values are not changed any more and we get the heads r_i^m as shown in Figure 3.22(a) and the tails q_i^m as in Figure 3.22(b).



(a) Calculated heads r_i^m



(b) Calculated tails q_i^m

Figure 3.22: Calculation of heads r_i^m and tails q_i^m

Thus, improved heads r_i and tails q_i for the activities are given by

$$r_1 = 0, r_2 = 4, r_3 = 6, r_4 = 10, q_3 = 3, q_2 = 5, q_1 = 10, q_0 = 10.$$

□

3.2.7 Reference notes

The calculation of time windows for activities and the notion of critical paths are the oldest techniques proposed in project management. They have their origin in the methods called PERT (Program Evaluation and Review Technique, cf. Malcolm et al. [99]) and CPM (Critical Path Method, cf. Kelley [78]) which were invented in 1958 in connection with military projects.

General aspects of constraint propagation can for example be found in Kumar [95], Tsang [138] and van Hentenryck [71]. The book of Baptiste et al. [7] and the survey on disjunctive scheduling problems by Dorndorf et al. [49] contain the most important constraint propagation techniques in connection with scheduling problems.

A survey on the different disjunctive interval consistency tests can be found in Dorndorf et al. [48], Dorndorf [47], and Phan Huy [121]. In connection with a branch-and-bound algorithm for the job-shop problem, Carlier and Pinson [31] were the first to present an algorithm which finds all inputs and outputs for all subsets $J \subseteq I$. It is based on calculating Jackson's preemptive schedule for the considered set and updates all release dates and deadlines in $O(n^2)$. The $O(n^2)$-implementation of the input/output test presented in Section 3.2.4 is due to Nuijten [115]. Improved algorithms with complexity $O(n \log n)$ were later proposed by Carlier and Pinson [32] and Brucker et al. [20] using more complex data structures.

The $O(n^2)$-implementation of the input/output negation test presented in Section 3.2.4 is due to Baptiste and Le Pape [6]. An algorithm where one iteration needs $O(n \log n)$ time, but in general more iterations are necessary, was proposed by Vilim [140]. The input-or-output test has been suggested by Dorndorf et al. [48] and an $O(n^3)$-algorithm was given by Phan Huy [121]. The presented $O(n^2)$-algorithm is due to Brucker and Knust [27].

Usually, in the literature only implementations for the weaker tests (3.13), in which the condition $\nu \neq \mu$ is dropped, are considered. Phan Huy [121] developed an $O(n^3)$-algorithm for the stronger input/output tests based on (3.12) and an $O(n^4)$-algorithm for the stronger input-or-output test, where additionally $\nu \neq \mu$ is imposed.

Interval consistency tests for cumulative resources are also called "energetic reasoning" and are summarized in Baptiste et al. [7]. The concept of shaving has been proposed by Martin and Shmoys [101].

The concept of symmetric triples and its extensions was introduced by Brucker et al. [22] in connection with a branch-and-bound algorithm for the RCPSP.

The preprocessing procedures reducing modes and resources in the multi-mode case are presented in Sprecher et al. [135], the calculation of improved heads and tails taking into account the modes can be found in Heilmann [70].

3.3 Lower Bounds

In this section methods to calculate lower bounds for the makespan of an RCPSP will be discussed. If LB is a lower bound for an instance of the RCPSP and UB is the solution value given by some heuristic, then $UB - LB$ is an upper bound for the distance between the optimal solution value and UB. Furthermore, $\frac{UB-LB}{UB}$ is an upper bound for the relative error. Thus, good lower bounds may be used to estimate the quality of heuristic solutions. They are also needed in connection with branch-and-bound algorithms.

Usually, two types of lower bounds can be distinguished for a combinatorial optimization problem: constructive and destructive bounds. **Constructive** lower bounds are usually provided by solving relaxations of the problem, which are less complex than the original problem. In a **relaxation** of an optimization problem certain restrictions (which make the problem hard) are eliminated. For example, if we relax the resource constraints of the RCPSP, the optimal makespan of the relaxed problem is equal to the length of a longest path in the activity-on-node network, which can be calculated efficiently.

Destructive lower bounds are based on a different idea. To derive lower bounds with this technique we consider the decision version (feasibility problem) of the optimization problem: Given a threshold value T, does a feasible schedule exist with an objective value smaller than or equal to T? If we can prove that such a schedule does not exist, then $T + 1$ is a valid lower bound for the optimization problem supposing that all data are integral. To contradict (destruct) a threshold value T, again relaxations may be used. If we can state infeasibility for a relaxed problem, obviously the original problem is also infeasible. To find the best lower bound we search for the largest T, where infeasibility can be proved.

This can be organized in incremental steps or by binary search. An incremental procedure starts with a valid lower bound value $T = LB$ and increases T in each step by an appropriate value $\Delta \geq 1$ until it is not possible to state infeasibility. When applying binary search, in each step we consider an interval $[L, U]$ in which we search for a valid lower bound value. Initially, we start with a valid upper bound U^0 and a valid lower bound L^0. In each iteration we solve the feasibility problem for $T = \lfloor \frac{U+L}{2} \rfloor$. If we can prove that no feasible solution with an objective value smaller than or equal to T exists, we increase the lower bound L to the value $T+1$ and repeat the procedure with the interval $[T+1, U]$. Otherwise, if we do not succeed in proving infeasibility for the threshold T, we replace U by T and repeat the procedure in the interval $[L, T]$. The binary search procedure terminates as soon as $L \geq U$ holds after at most $O(\log(U^0 - L^0))$ steps. Then L equals the largest lower bound value which can be calculated in this way.

In the next subsections we discuss several methods to calculate lower bounds for the optimal makespan $C_{\max}$. While in Sections 3.3.1 and 3.3.2 constructive lower bounds are described, a destructive lower bound can be found in Section 3.3.3. This approach is generalized to the multi-mode case in Section 3.3.4.

3.3.1 Combinatorial constructive lower bounds

In Section 3.2.1 we already considered a simple constructive lower bound: if we relax all resource constraints and only take into account the precedence constraints, the length of a critical path in the activity-on-node network defines a lower bound for the optimal makespan, denoted by LB_0. If no generalized precedence relations are given, the graph is acyclic, and the longest path can be calculated in $O(n^2)$ time.

Another simple lower bound can be determined in $O(nr)$ time by considering each resource separately. For each renewable resource $k = 1, \ldots, r$ the value $\lceil \sum_{i=1}^{n} r_{ik}\, p_i / R_k \rceil$ defines a lower bound for the optimal makespan because the total work $\sum_{i=1}^{n} r_{ik}\, p_i$ for all activities cannot be greater than the work $R_k \cdot C_{\max}$ which is available for resource k in the interval $[0, C_{\max}]$. Thus, the maximum among all resources gives the resource-based lower bound

$$LB_1 := \max_{k=1}^{r} \left\lceil \sum_{i=1}^{n} r_{ik}\, p_i / R_k \right\rceil .$$

The longest path length LB_0 of a critical path CP_0 can be strengthened by partially taking into account some resource conflicts as follows. If a schedule with $C_{\max} = LB_0$ exists, each activity i has to be completed before the deadline $d_i := LB_0 - q_i$ (recall that the tail q_i is a lower bound for the length of the time period between the completion time of i and the optimal makespan).

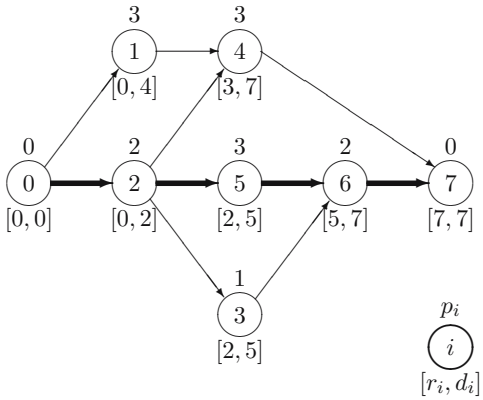
For each activity i not belonging to the critical path CP_0, let e_i be the maximal length of an interval contained in $[r_i, d_i]$ in which i can be processed with the activities from CP_0 simultaneously without violating the resource constraints. If $e_i < p_i$ holds, no feasible schedule with $C_{\max} = LB_0$ exists and in order to get a feasible schedule the time window of activity i has to be enlarged by at least $p_i^+ := \max\{p_i - e_i, 0\}$ time units. Thus,

$$LB_S := LB_0 + \max_{i \notin CP_0} \{p_i^+\}$$

defines a valid lower bound value. For each activity not in CP_0 we have to check, whether the resources left by the activities from CP_0 are sufficient or not. This can be done in $O(nr|CP_0|)$ time, where $|CP_0|$ denotes the number of activities on CP_0.

Example 3.7: Consider the instance with $n = 6$ activities and $r = 2$ resources with capacities $R_1 = 3, R_2 = 1$ shown in Figure 3.23.

The critical path $CP_0 = (0 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 7)$ has the length $LB_0 = 7$. In Figure 3.24 the partial schedule for all critical activities from CP_0 is drawn, furthermore, the time windows for the remaining activities $i \notin CP_0$ are shown. For these activities we have $e_1 = 2$ since activity 1 can be processed in parallel



i	1	2	3	4	5	6
p_i	3	2	1	3	3	2
r_{i1}	2	2	0	2	1	2
r_{i2}	1	1	1	1	0	1
r_i	0	0	2	3	2	5
q_i	3	5	2	0	2	0
d_i	4	2	5	7	5	7

Figure 3.23: A project with $n = 6, r = 2$ and time windows for $LB_0 = 7$

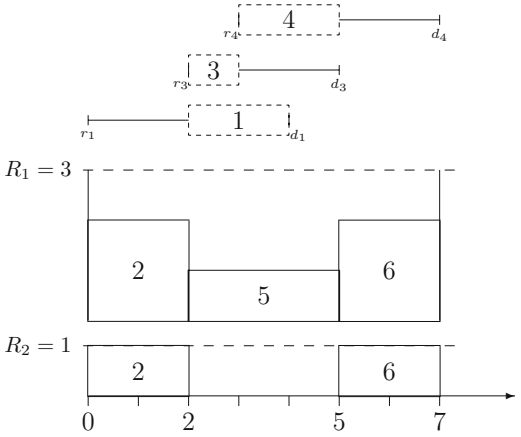


Figure 3.24: Partial schedule for CP_0

with CP_0 in the interval $[2, 4]$, $e_3 = 3$ since 3 can be processed in the interval $[2, 5]$ and $e_4 = 2$.

Thus, $p_1^+ = 3 - 2 = 1$, $p_3^+ = \max\{1 - 3, 0\} = 0$, $p_4^+ = 3 - 2 = 1$ and

$$LB_S = LB_0 + \max_{i \notin CP_0} \{p_i^+\} = 7 + 1 = 8.$$

□

Note that in LB_S only conflicts between activities from CP_0 and activities $i \notin CP_0$ are taken into account, conflicts among activities $i \notin CP_0$ are not considered. More complicated extensions of the critical path lower bound LB_0 may be obtained by considering a critical path together with a second node-disjoint path in the network. Taking into account the fixed positions of the activities from the critical path, the minimum completion time of the activities in the second path provides a lower bound.

3.3.2 An LP-based constructive lower bound

In the following we consider a linear programming formulation partially relaxing the precedence constraints and allowing preemption. The columns of this LP correspond to so-called non-dominated **feasible subsets**. Feasible sets X are sets of activities which may be processed simultaneously, i.e. there are no conjunctions or disjunctions between any pair of activities $i, j \in X$ and all resource constraints are respected (i.e. $\sum_{i \in X} r_{ik} \leq R_k$ for $k = 1, \ldots, r$). Such a set is called **non-dominated** if it is not a proper subset $X \subset Y$ of another feasible set Y.

We consider all non-dominated feasible sets and additionally the one-element sets $\{i\}$ for $i = 1, \ldots, n$. We denote all these sets by $X_1, X_2, \ldots, X_f$ and associate with each set X_j an incidence vector $a^j \in \{0,1\}^n$ defined by

$$a_i^j := \begin{cases} 1, & \text{if} \quad i \in X_j \\ 0, & \text{otherwise.} \end{cases}$$

Furthermore, let x_j be a variable denoting the number of time units where all activities in X_j are processed simultaneously. Then the following linear program provides a lower bound for the RCPSP. It relaxes the conjunctions by treating them as disjunctions and allows preemption.

$$\min \quad \sum_{j=1}^{f} x_j \tag{3.40}$$

$$\text{s.t.} \quad \sum_{j=1}^{f} a_i^j x_j \geq p_i \qquad (i = 1, \ldots, n) \tag{3.41}$$

$$x_j \geq 0 \qquad (j = 1, \ldots, f). \tag{3.42}$$

In (3.40) the makespan $C_{\max} = \sum_{j=1}^{f} x_j$ is minimized subject to the constraints (3.41) which ensure that all activities are processed for at least p_i time units. Note that in this formulation we must allow that an activity i may be processed longer than its processing time p_i since only non-dominated subsets are considered (this does not change the optimal makespan).

Example 3.8: Consider the instance in Figure 3.25 with $n = 6$ activities, one resource with capacity $R_1 = 4$ and $LB_0 = 4$.

For this instance we have 14 feasible sets $\{1\}$, $\{1,2\}$, $\{1,2,3\}$, $\{1,3\}$, $\{1,6\}$, $\{2\}$, $\{2,3\}$, $\{2,5\}$, $\{3\}$, $\{3,4\}$, $\{3,6\}$, $\{4\}$, $\{5\}$, $\{6\}$. Among these sets we have 5 non-dominated sets: $X_1 = \{1,2,3\}$, $X_2 = \{1,6\}$, $X_3 = \{2,5\}$, $X_4 = \{3,4\}$, $X_5 = \{3,6\}$.

An optimal preemptive schedule for the relaxation with makespan 5 is shown in Figure 3.25. Note that all activities $i \neq 2$ are processed for exactly p_i time units, only activity 2 is processed longer for $p_2 + 1 = 3$ time units. This schedule corresponds to the LP solution $x_1 = 1, x_2 = 1, x_3 = 2, x_4 = 1, x_5 = 0$. □

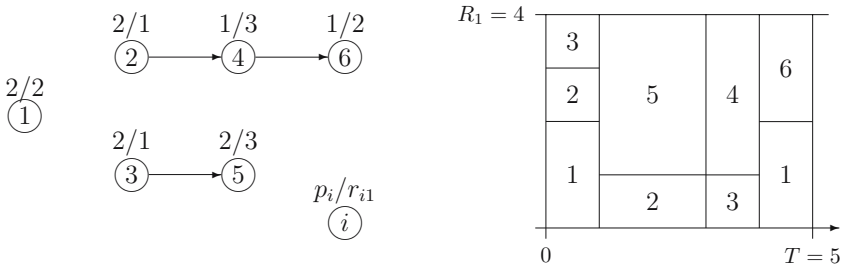


Figure 3.25: A project and a corresponding optimal preemptive schedule

Unfortunately, the cardinality of all non-dominated feasible sets grows exponentially with the number n of activities. For $n = 60$ we have approximately $f = 300\,000$, for $n = 90$ even $f = 8\,000\,000$ columns. However, it is not necessary to generate and store all these columns.

As described in Section 2.3.5, we can work with only a few of them at a time and only generate new feasible sets when they are really needed. In the following we apply such a **delayed column generation approach** to (3.40) to (3.42) and describe the problem-specific details. As shown in the generic column generation algorithm in Figure 2.11, besides the problem-independent subprocedure `Optimize` we have the problem-specific subprocedures `Initialize` providing a basic starting solution, `InsertDeleteColumns` organizing the insertion and deletion of columns, and the pricing procedure `CalculateColumns`.

`Initialize`: As a starting basis for the first iteration of the simplex method we may use the one-element sets $X_i := \{i\}$ for $i = 1, \ldots, n$ and set $x_i := p_i$ for the corresponding variables x_i (corresponding to the unit-vectors $e^i \in \mathbb{R}^n$). Furthermore, we add the columns corresponding to the slack variables in (3.41) to the working set.

`InsertDeleteColumns`: We use a delete-strategy which never deletes the slack variables from the working set, i.e. $-e^i$ is always in the working set.

`CalculateColumns`: After solving the corresponding LP to optimality by the procedure `Optimize`, we search for feasible columns which are able to improve the objective value when entering the basis. If y_i $(i = 1, \ldots, n)$ are the values of the dual variables associated with the current basic solution, we have to find a feasible column $a \in \{0, 1\}^n$ with $ya > 1$, i.e. a has to satisfy

$$\sum_{i=1}^{n} y_i \cdot a_i \;>\; 1 \tag{3.43}$$

$$\sum_{i=1}^{n} r_{ik} \cdot a_i \;\leq\; R_k \quad (k = 1, \ldots, r) \tag{3.44}$$

$$a_i \cdot a_j \;=\; 0 \qquad (i - j \in D, i \to j \in C \text{ or } j \to i \in C). \tag{3.45}$$

Conditions (3.44) and (3.45) ensure that a column a corresponds to a feasible

subset (respecting the resource constraints, the disjunctions D, and the conjunctions C).

Columns which satisfy conditions (3.43) to (3.45) are generated by an enumeration procedure, which works as follows: Since we have solved the current LP to optimality and the slack-variables are kept in the working set, we have for all dual variables $y_i \geq 0$ because

- $-y_i = -ye^i = c_i = 0$, if $-e^i$ is in the basis, or
- $-y_i = -ye^i \leq c_i = 0$, since $-e^i$ is in the working set.

We sort the activities such that

$$y_1 \geq y_2 \geq \ldots \geq y_{n_0} > 0,\ y_{n_0+1} = \ldots = y_n = 0,$$

where n_0 denotes the last index of a positive dual variable.

Due to (3.43) we only have to consider activities i with $y_i > 0$, i.e. it is sufficient to generate non-dominated vectors $(a_1, \ldots, a_{n_0})$ of length n_0. They can be extended to non-dominated vectors $(a_1, \ldots, a_n)$ with length n by adding some activities $i \in \{n_0 + 1, \ldots, n\}$ which maintain feasibility.

To calculate incidence vectors of relevant non-dominated feasible sets, we define an enumeration tree in which the vertices represent partial vectors $(a_1, \ldots, a_l)$ with $1 \leq l \leq n_0$. A vector $(a_1, \ldots, a_l)$ has at most two sons: $(a_1, \ldots, a_l, 0)$ is the right son, the left son $(a_1, \ldots, a_l, 1)$ exists only if $(a_1, \ldots, a_l, 1, 0, \ldots, 0)$ corresponds to a feasible set.

We apply a depth-first search in the enumeration tree where leaves correspond to feasible subsets. The sets are generated in a lexicographic order.

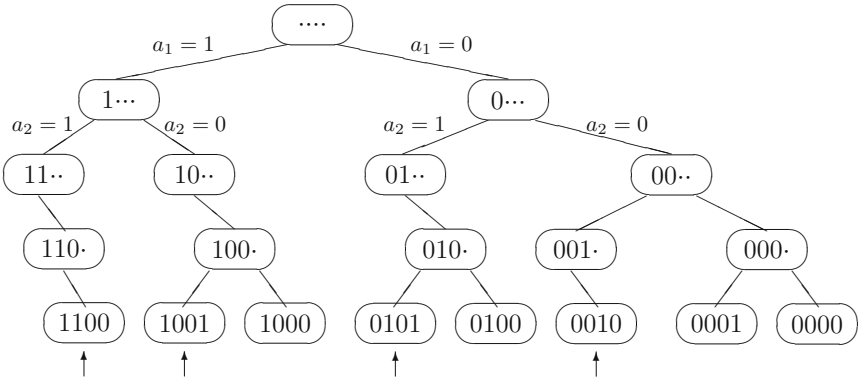


Figure 3.26: Enumeration tree for the calculation of non-dominated sets

Example 3.9: Consider a small example with $n = 4$ activities, $r = 2$ resources with capacities $R_1 = 2$, $R_2 = 3$, and a precedence relation $2 \rightarrow 3$. The resource

requirements r_{ik} of the activities are given by

i	1	2	3	4
r_{i1}	1	1	2	0
r_{i2}	1	1	2	2

i.e. the pairs $1-3$ and $3-4$ are incompatible with respect to the resources.

The calculation of the non-dominated sets is shown in Figure 3.26, where each partial set is characterized by a 4-tuple $(a_1, a_2, a_3, a_4) \in \{0,1\}^4$. We obtain 8 feasible subsets, 4 of them are non-dominated (indicated by an arrow under the corresponding leaf). □

Instead of checking for each subset whether it is non-dominated or not, we only generate a subset of non-dominated feasible columns satisfying (3.43) to (3.45), which are automatically non-dominated. Each time a column a^* satisfying (3.43) to (3.45) has been found, we only search for further columns a with

$$\sum_{i=1}^{n_0} y_i a_i > \sum_{i=1}^{n_0} y_i a_i^*. \tag{3.46}$$

This ensures that all further sets during the computation are non-dominated for the following reason: An incidence vector a is dominated if at least one component a_k can be changed from 0 to 1 without violating feasibility. But then a dominating column a' with $a'_k = 1$ has already been considered (either generated as a leaf or eliminated due to its objective value), since we scan the columns in lexicographic order. Let a^* be the best column found when we consider a. Since a' has been considered before, we have

$$\sum_{i=1}^{n_0} y_i a_i^* \geq \sum_{i=1}^{n_0} y_i a'_i \geq \sum_{i=1}^{n_0} y_i a_i + y_k \geq \sum_{i=1}^{n_0} y_i a_i,$$

due to $y \geq 0$, i.e. column a is not generated by our procedure. Thus, we do not have to check dominance explicitly.

When we have found a suitable leaf a^*, we fill this column successively with all activities $n_0+1, \ldots, n$ which do not violate feasibility. Then the corresponding column is also non-dominated and we backtrack to depth n_0. Note that if we search the enumeration tree completely, then a non-dominated column a which maximizes $\sum_{i=1}^{n} y_i a_i$ is found.

In order to reduce the enumeration tree we may include the following simple dominance rule into our enumeration procedure: If we set $a_k := 0$ in step k and if

$$\sum_{\lambda=1}^{k-1} y_\lambda \cdot a_\lambda + \sum_{\lambda=k+1}^{n_0} y_\lambda \leq \sum_{i=1}^{n_0} y_i \cdot a_i^*$$

holds, the best column a^* found so far cannot be improved and therefore we can backtrack.

We stop the search process if one of the following conditions holds:

- we have found a given number of columns, or
- we have found at least one column and the ratio between the number μ of generated leaves and the number μ^u of leaves satisfying (3.46) exceeds a certain limit s (i.e. $\mu \geq \mu^u \cdot s$).

After applying the procedure `CalculateColumns`, the generated columns are added to the working set by the procedure `InsertDeleteColumns`. Furthermore, if the number of columns in the current working set exceeds a given size, some arbitrary non-basic columns are deleted.

Unfortunately, for problems with a larger number of activities the quality of the presented lower bound worsens. The reason is that the possibility of preemption allows us to split the activities into many pieces and to put them appropriately into different feasible subsets yielding low objective values. Therefore, in the next subsection we strengthen the lower bound by additionally taking into account time windows. Furthermore, we use a destructive approach.

3.3.3 An LP-based destructive method

In this section we consider an extension of the previous linear programming formulation where additionally time windows for the activities are taken into account. We use a destructive approach which combines constraint propagation and linear programming techniques.

Given a hypothetical upper bound T, one first tries to prove infeasibility by constraint propagation. Constraint propagation also provides time windows $[r_i, d_i]$ for the activities $i = 1, \ldots, n$. In case that we cannot prove infeasibility, we use the time windows and try to prove that no preemptive schedule with $C_{\max} \leq T$ exists such that all activities are processed within their time windows and all resource constraints are respected.

For the LP-formulation let $z_0 < z_1 < \ldots < z_\tau$ be the ordered sequence of all different r_i- and d_i-values. For $t = 1, \ldots, \tau$ we consider the intervals $I_t := [z_{t-1}, z_t]$ of length $z_t - z_{t-1}$. With each interval I_t we associate a set A_t of all activities i which can partially be scheduled in this interval, i.e. with $r_i \leq z_{t-1} < z_t \leq d_i$. Let X_{jt} $(j = 1, \ldots, f_t)$ be the feasible subsets of A_t and denote by a^{jt} the incidence vector corresponding to X_{jt}. Furthermore, let x_{jt} be a variable denoting the number of time units where all activities in X_{jt} are processed simultaneously. Then the preemptive feasibility problem may be written as follows:

$$\sum_{t=1}^{\tau} \sum_{j=1}^{f_t} a_i^{jt} x_{jt} \geq p_i \qquad (i = 1, \ldots, n) \tag{3.47}$$

$$\sum_{j=1}^{f_t} x_{jt} \leq z_t - z_{t-1} \qquad (t = 1, \ldots, \tau) \tag{3.48}$$

$$x_{jt} \geq 0 \qquad (t = 1, \ldots, \tau;\ j = 1, \ldots, f_t) \tag{3.49}$$

Due to restrictions (3.47) all activities i are processed for at least p_i time units. Conditions (3.48) ensure that the number of time units scheduled in interval I_t does not exceed the length $z_t - z_{t-1}$ of this interval.

By introducing artificial variables u_t for $t = 1, \ldots, \tau$ in conditions (3.48) the feasibility problem can be formulated as a linear program:

$$\min \quad \sum_{t=1}^{\tau} u_t \tag{3.50}$$

$$\text{s.t.} \quad \sum_{t=1}^{\tau} \sum_{j=1}^{f_t} a_i^{jt} x_{jt} \geq p_i \qquad (i = 1, \ldots, n) \tag{3.51}$$

$$-\sum_{j=1}^{f_t} x_{jt} + u_t \geq -z_t + z_{t-1} \qquad (t = 1, \ldots, \tau) \tag{3.52}$$

$$x_{jt} \geq 0 \qquad (t = 1, \ldots, \tau;\ j = 1, \ldots, f_t) \tag{3.53}$$

$$u_t \geq 0 \qquad (t = 1, \ldots, \tau) \tag{3.54}$$

A solution exists for the preemptive feasibility problem if and only if the linear program has the optimal solution value zero, i.e. if all values of the artifical variables become zero.

Example 3.10: Consider again Example 3.8 from the previous subsection. In Figure 3.27 additionally the time windows $[r_i, d_i]$ for $T = 6$ are shown.

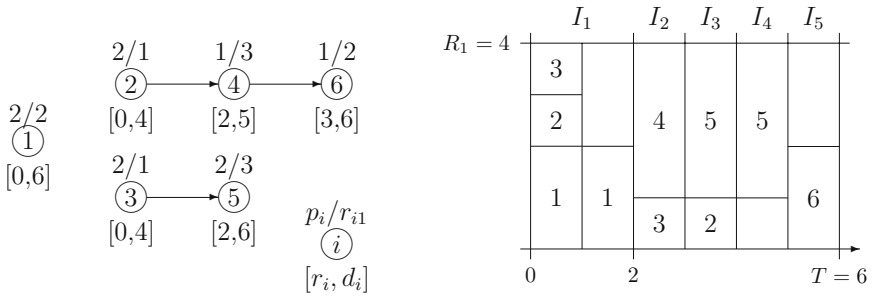


Figure 3.27: A feasible preemptive schedule for $T = 6$

For this instance we have $\tau = 5$ intervals I_t with the following sets A_t:

$$\begin{aligned}
I_1 &= [0, 2]: \quad A_1 = \{1, 2, 3\},\\
I_2 &= [2, 3]: \quad A_2 = \{1, 2, 3, 4, 5\},\\
I_3 &= [3, 4]: \quad A_3 = \{1, 2, 3, 4, 5, 6\},\\
I_4 &= [4, 5]: \quad A_4 = \{1, 4, 5, 6\},\\
I_5 &= [5, 6]: \quad A_5 = \{1, 5, 6\}.
\end{aligned}$$

A feasible preemptive schedule corresponding to the solution

$$x_{\{1,2,3\},1} = 1,\ x_{\{1\},1} = 1,\ x_{\{3,4\},2} = 1,\ x_{\{2,5\},3} = 1,\ x_{\{5\},4} = 1,\ x_{\{6\},5} = 1,$$

(where $x_{X,t}$ denotes that subset X is processed in interval I_t) can be found in Figure 3.27. Furthermore, it is easy to see that for $T = 5$ no feasible schedule respecting the time windows exists. Thus, we get a lower bound value of 6, which improves the lower bound from the previous subsection by one unit. □

Again the linear programming formulation contains an exponential number of variables, but again it can be solved efficiently with column generation techniques. In the following we describe how the procedures from the previous subsection have to be modified:

- `Initialize`: For each activity $i \in \{1\dots,n\}$ let $I_{t(i)} \subseteq [r_i, d_i]$ be the first interval in which i can be processed. Then the starting working set consists of the columns corresponding to the one-element sets $\{i\}$ $(i = 1,\dots,n)$ in the interval $I_{t(i)}$, the artificial variables u_t $(t = 1,\dots,\tau)$ and the slack variables according to (3.52).

- `InsertDeleteColumns`: We use a delete-strategy which never deletes the slack variables according to (3.52) from the working set.

- `CalculateColumns`: We have to calculate new entering columns or to show that no improving column exists. If we denote the dual variables corresponding to conditions (3.51) by y_i $(i = 1,\dots,n)$ and the dual variables belonging to (3.52) by w_t $(t = 1,\dots,\tau)$, then the constraints in the associated dual problem have the form:

$$\sum_{i=1}^{n} a_i^{jt} y_i - w_t \le 0 \quad (t = 1,\dots,\tau;\ j = 1,\dots,f_t) \tag{3.55}$$

$$w_t \le 1 \quad (t = 1,\dots,\tau) \tag{3.56}$$

$$y_i \ge 0 \quad (i = 1,\dots,n) \tag{3.57}$$

$$w_t \ge 0 \quad (t = 1,\dots,\tau) \tag{3.58}$$

  Since the slack variables are kept in the working set, conditions (3.56) to (3.58) are always satisfied. Thus, we have to find a column $a^{jt} \in \{0,1\}^n$ violating (3.55), i.e. satisfying

$$\sum_{i=1}^{n} a_i^{jt} y_i - w_t > 0$$

  for an index $t \in \{1,\dots,\tau\}$. This can be done by an enumeration procedure as in Section 3.3.2, where we restrict the search for improving columns to activities from the set A_t.

The LP-formulation may be strengthened by adding additional valid inequalities. In the following three different types of inequalities are described. Since these inequalities "cut off" solutions for the relaxation which are infeasible for the original problem, they are also called **cuts**.

Energetic cuts

For an interval $[a, b] \subseteq [0, T]$ and an activity i we may calculate a lower bound $P_i(a, b)$ for the amount of time where i has to be processed in $[a, b]$ in any feasible schedule. The value $P_i(a, b)$ is given by the minimum of

- the interval length $b - a$,
- $p_i^+ := \max\{0, p_i - \max\{0, a - r_i\}\}$, which equals the required processing time in $[a, d_i]$ if i is started at time r_i, and
- $p_i^- := \max\{0, p_i - \max\{0, d_i - b\}\}$, which equals the required processing time in $[r_i, b]$ if i is completed at time d_i.

Obviously, the minimum of these three values is a lower bound for the processing time of activity i in the interval $[a, b] \cap [r_i, d_i]$.

For the LP-formulation we may consider each interval $[z_{t_1}, z_{t_2}]$ with $0 \leq t_1 < t_2 \leq \tau$ and add the inequalities

$$\sum_{t=t_1+1}^{t_2} \sum_{j=1}^{f_t} a_i^{jt} x_{jt} \geq P_i(z_{t_1}, z_{t_2}) \quad (i = 1, \ldots, n;\ 0 \leq t_1 < t_2 \leq \tau). \tag{3.59}$$

Example 3.11: Consider the example with $n = 2$ activities and a single resource with capacity $R_1 = 1$ shown in Figure 3.28. A feasible preemptive schedule for $T = 3$ respecting the time windows is also shown in the figure.

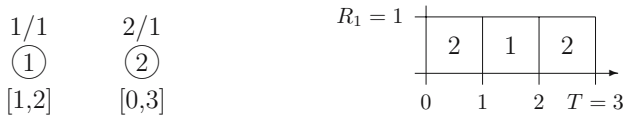


Figure 3.28: A feasible preemptive schedule for $T = 3$

If the inequalities (3.59) are added, this schedule is no longer feasible since for activity 2 in the interval $[1, 2]$ we have $P_2(1, 2) = 1$ but in the schedule no part of activity 2 is scheduled in $[1, 2]$. Thus, adding the inequalities (3.59) may lead to stronger lower bounds. □

Non-preemptive cuts

Some other valid inequalities my be derived from the observation that in a non-preemptive schedule (where additionally all starting times are integral), an activity cannot be processed simultaneously in two intervals $[z_{t_\nu - 1}, z_{t_\nu}[$ $(\nu = 1, 2)$ which have a distance of at least $p_i - 1$ time units. For example, an activity i with processing time $p_i = 5$ cannot overlap with the intervals $[1, 4[$ and $[8, 10[$

simultaneously. Furthermore, we may state that activity i can be processed for at most $\max\{4-1, 10-8\} = 3$ time units in $[1,4[\cup[8,10[$.

More generally, for each activity i we consider subsets of the given intervals $I_t = [z_{t-1}, z_t[$ in which i can be processed and which have a distance of at least $p_i - 1$ time units. For activity i let $\Psi_i \subseteq \{1, \ldots, \tau\}$ be a subset of interval indices with $I_t = [z_{t-1}, z_t[\subseteq [r_i, d_i]$ for all $t \in \Psi_i$ and $z_{t'-1} - z_t \geq p_i - 1$ for all indices $t < t' \in \Psi_i$. Then we may state that i can be processed in at most one interval of Ψ_i, i.e. the maximal length of an interval in Ψ_i is a lower bound for the total processing time of i in all intervals belonging to Ψ_i.

Thus, we may add the inequalities

$$\sum_{t \in \Psi_i} \sum_{j=1}^{f_t} a_i^{jt} x_{jt} \leq \max_{t \in \Psi_i} \{z_t - z_{t-1}\} \quad (i = 1, \ldots, n;\ \Psi_i \subseteq \{1, \ldots, \tau\}). \tag{3.60}$$

There are many possible subsets $\Psi_i \subseteq \{1, \ldots, \tau\}$ for an activity i representing non-overlapping intervals with distance at least $p_i - 1$. For each activity i and each $t \in \{1, \ldots, \tau\}$ with $I_t \subseteq [r_i, d_i]$ we construct a set $\Psi_{i,t}$ containing t where I_t has the maximal length in $\Psi_{i,t}$, i.e. $z_t - z_{t-1} = \max\limits_{\psi \in \Psi_{i,t}} \{z_\psi - z_{\psi-1}\}$.

Such a set $\Psi_{i,t}$ may be constructed as follows.

1. Set $\Psi_{i,t} := \{t\}$.
2. Let ψ be the maximal index in the current set $\Psi_{i,t}$. Determine the smallest $\psi' > \psi$ with $z_{\psi'-1} - z_\psi \geq p_i - 1$ and $z_{\psi'} - z_{\psi'-1} \leq z_t - z_{t-1}$. If no such ψ' exists, go to Step 3. Otherwise, add ψ' to $\Psi_{i,t}$ and iterate Step 2.
3. Let ψ be the minimal index in the current set $\Psi_{i,t}$. Determine the largest $\psi' < \psi$ with $z_{\psi-1} - z_{\psi'} \geq p_i - 1$ and $z_{\psi'} - z_{\psi'-1} \leq z_t - z_{t-1}$. Add ψ' to $\Psi_{i,t}$ and iterate Step 3 until no further index ψ' exists.

Example 3.12: Consider one activity i with $p_i = 3$, $[r_i, d_i] = [0, 27]$ and time points $z_0 = 0$, $z_1 = 4$, $z_2 = 6$, $z_3 = 8$, $z_4 = 11$, $z_5 = 14$, $z_6 = 16$, $z_7 = 20$, $z_8 = 23$, $z_9 = 25$, $z_{10} = 27$ (cf. also Figure 3.29).

For $t = 5$ the construction of $\Psi_{i,5}$ proceeds as follows.

- Step 1: We set $\Psi_{i,5} := \{5\}$ and have $z_t - z_{t-1} = z_5 - z_4 = 14 - 11 = 3$.
- Step 2: Starting with the maximal index $\psi = 5$, the first $\psi' > \psi = 5$ with the required properties is $\psi' = 8$ (for $\psi' = 6$ we have $z_{\psi'-1} - z_\psi = z_5 - z_5 = 0 < 2 = p_i - 1$, for $\psi' = 7$ we have $z_{\psi'} - z_{\psi'-1} = z_7 - z_6 = 4 > 3 = z_t - z_{t-1}$, and for $\psi' = 8$ we have $z_{\psi'-1} - z_\psi = z_7 - z_5 = 6 \geq 2 = p_i - 1$ and $z_{\psi'} - z_{\psi'-1} = z_8 - z_7 = 3 \leq 3 = z_t - z_{t-1}$). After adding $\psi' = 8$ to $\Psi_{i,5}$ the new maximal index is $\psi = 8$. Since for $\psi' = 10$ we get $z_{\psi'-1} - z_\psi = z_9 - z_8 = 2 \geq 2 = p_i - 1$ and $z_{\psi'} - z_{\psi'-1} = z_{10} - z_9 = 2 \leq 3 = z_t - z_{t-1}$, we add $\psi' = 10$ to $\Psi_{i,5}$.

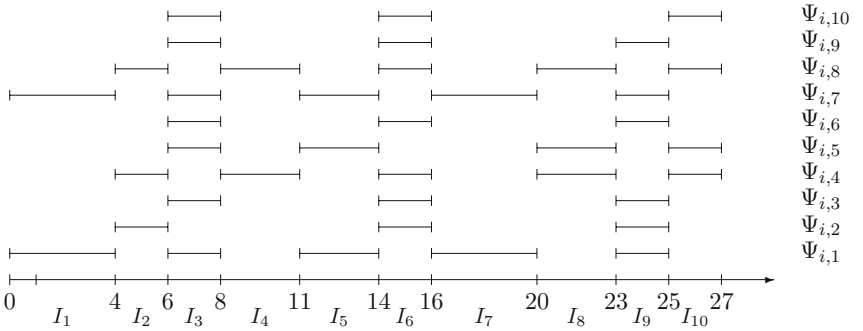


Figure 3.29: Construction of the sets $\Psi_{i,t}$

- Step 3: Starting with the minimal index $\psi = 5$, the first $\psi' < \psi = 5$ with the required properties is $\psi' = 3$ (for $\psi' = 4$ we have $z_{\psi-1} - z_{\psi'} = z_4 - z_4 = 0 < 2 = p_i - 1$, and for $\psi' = 3$ we have $z_{\psi-1} - z_{\psi'} = z_4 - z_3 = 3 \geq 2 = p_i - 1$ and $z_{\psi'} - z_{\psi'-1} = z_3 - z_2 = 2 \leq 3 = z_t - z_{t-1}$). After adding $\psi' = 3$ to $\Psi_{i,5}$ the new minimal index is $\psi = 3$. Since for $\psi' = 2$ we have $z_{\psi-1} - z_{\psi'} = 0 < 2 = p_i - 1$ and for $\psi' = 1$ we have $z_{\psi'} - z_{\psi'-1} = 4$, no further suitable index ψ' exists. Thus, the final set is $\Psi_{i,5} = \{3, 5, 8, 10\}$.

The other sets $\Psi_{i,t}$ are constructed similarly (cf. Figure 3.29). As a result we get $\Psi_{i,1} = \Psi_{i,7} = \{1, 3, 5, 7, 9\}$, $\Psi_{i,2} = \{2, 6, 9\}$, $\Psi_{i,3} = \Psi_{i,6} = \Psi_{i,9} = \{3, 6, 9\}$, $\Psi_{i,4} = \Psi_{i,8} = \{2, 4, 6, 8, 10\}$, $\Psi_{i,10} = \{3, 6, 10\}$. □

Precedence cuts

Finally, the third type of inequalities tries to take into account the precedence constraints a little bit more. For each activity $i = 1, \ldots, n$ we introduce an additional variable M_i representing the **midpoint** of activity i (i.e. the average of its starting and its completion time, $M_i = \frac{S_i + C_i}{2}$). Then the precedence relations $i \to j \in C$ may be expressed as $S_j \geq C_i$, which implies $C_j - C_i \geq p_j$ and $S_j - S_i \geq p_i$. By adding the last two inequalities and dividing the result by 2 we get

$$M_j - M_i \geq \frac{p_j + p_i}{2} \quad \text{for all } i \to j \in C. \tag{3.61}$$

For a given schedule if activity i is processed in interval I_t, let S_{it} be the starting time, C_{it} be the completion time and $p_{it} := C_{it} - S_{it} > 0$ be the processing time of the part of i processed in interval I_t. In a non-preemptive schedule each activity i starts in some interval I_{t_1}, is continuously processed in some intervals I_t and completes in some interval I_{t_2} with $t_1 \leq t \leq t_2$. Thus, we have

$$S_i = S_{it_1} \geq z_{t_1-1},\ C_i = C_{it_2} \leq z_{t_2} \text{ and } z_t = C_{it} = S_{i,t+1} \text{ for } t_1 \leq t \leq t_2 - 1.$$

This implies

$$\begin{aligned} M_i\, p_i &= \frac{C_i + S_i}{2}(C_i - S_i) = \frac{C_i^2 - S_i^2}{2} = \sum_{t=t_1}^{t_2}(C_{it}^2 - S_{it}^2)/2 \\ &= \sum_{t=t_1}^{t_2}(C_{it} - S_{it})(C_{it} + S_{it})/2 = \sum_{t=t_1}^{t_2} p_{it} M_{it} = \sum_{t=1}^{\tau} p_{it} M_{it}, \end{aligned}$$

where $M_{it} := (C_{it} + S_{it})/2$ defines the midpoint of i in the interval I_t.

For each interval I_t with $p_{it} > 0$ we have $p_{it} \geq 1$, i.e.

$$M_{it} = S_{it} + \frac{p_{it}}{2} \geq z_{t-1} + \frac{1}{2} \quad \text{and} \quad M_{it} = C_{it} - \frac{p_{it}}{2} \leq z_t - \frac{1}{2}.$$

This implies

$$\sum_{t=1}^{\tau}(z_{t-1} + \frac{1}{2})\, p_{it} \leq \sum_{t=1}^{\tau} M_{it} p_{it} \leq \sum_{t=1}^{\tau}(z_t - \frac{1}{2})\, p_{it}.$$

Using the variables x_{jt}, the processing time of i in I_t my be expressed as $p_{it} := \sum_{j=1}^{f_t} a_i^{jt} x_{jt}$. Thus, the midpoint variables M_i may be linked to the x_{jt}-variables by the additional inequalities

$$\sum_{t=1}^{\tau}(z_{t-1} + \frac{1}{2}) \sum_{j=1}^{f_t} a_i^{jt} x_{jt} \leq M_i p_i \leq \sum_{t=1}^{\tau}(z_t - \frac{1}{2}) \sum_{j=1}^{f_t} a_i^{jt} x_{jt} \quad (i = 1, \ldots, n). \tag{3.62}$$

If these inequalities as well as conditions (3.61) are added to the LP, additional (infeasible) solutions are excluded, i.e. the lower bound is strengthened.

Example 3.13: Consider again Example 3.10. The schedule shown in Figure 3.27 is no longer feasible if the inequalities (3.61) and (3.62) for the precedence relation $2 \to 4$ are added to the LP. For this conjunction (3.61) becomes $M_4 - M_2 \geq \frac{p_4 + p_2}{2} = 1.5$. In the schedule activity 2 is processed in the intervals $[0, 1]$ and $[3, 4]$. Thus, (3.62) for $i = 2$ reads

$$4 = (0 + \frac{1}{2}) + (3 + \frac{1}{2}) \leq M_2\, p_2 \leq (1 - \frac{1}{2}) + (4 - \frac{1}{2}) = 4,$$

which implies $M_2 = 2$. Furthermore, activity 4 is processed in the interval $[2, 3]$, i.e. (3.62) for $i = 4$ reads

$$2.5 = 2 + \frac{1}{2} \leq M_4\, p_4 \leq 3 - \frac{1}{2} = 2.5$$

implying $M_4 = 2.5$. But then $M_4 - M_2 = 0.5 < 1.5$, i.e. (3.61) is violated for the conjunction $2 \to 4$. □

```
Algorithm Destructive LB (incremental search)
 1.  Calculate an upper bound UB and set T := UB − 1;
 2.  infeasible := ConstraintPropagation(T);
 3.  WHILE infeasible = FALSE DO
 4.        infeasible := LP(T);
 5.        IF infeasible = FALSE THEN
 6.           T := T − 1;
 7.           infeasible := ConstraintPropagation(T);
 8.        ENDIF
 9.  ENDWHILE
10.  LB := T + 1;
```

Figure 3.30: Incremental destructive lower bound procedure

To finish this subsection, the previously described methods are summarized in an incremental destructive lower bound procedure combining constraint propagation techniques and linear programming (cf. Figure 3.30). In Steps 2 and 7 the subprocedure `ConstraintPropagation`(T) is called, which applies certain reduction techniques to the restricted problem where $C_{\max} \leq T$ is imposed. It may find additional constraints which strengthen the problem data (i.e. additional precedence relations and reduced time windows) before the LP is solved. Furthermore, it returns a boolean variable `infeasible` which is `TRUE` if infeasibility could be detected for the threshold value T. If infeasibility could not be proved by constraint propagation (i.e. `ConstraintPropagation`(T) returns `FALSE`), in Step 4 another feasibility check is performed by solving the LP.
Since in each step of the procedure T is reduced by one, the first value T for which the boolean flag `infeasible` equals `TRUE` corresponds to the largest infeasible T, i.e. $LB := T + 1$ is the best possible lower bound value calculated in this way.

```
Algorithm Destructive LB (binary search)
 1.  Calculate an upper bound U and a lower bound L;
 2.  WHILE L < U DO
 3.        T := ⌊(U+L)/2⌋;
 4.        infeasible := CheckInfeasibility(T);
 5.        IF infeasible = TRUE THEN
 6.           L := T + 1;
 7.        ELSE
 8.           U := T;
 9.  ENDWHILE
```

Figure 3.31: Binary search destructive lower bound procedure

As already mentioned at the beginning of Section 3.3, a binary search procedure is more efficient. Such a procedure can be found in Figure 3.31, where the procedure `CheckInfeasibility`(T) does all infeasibility tests for a threshold value T. Iteratively we consider an interval $[L, U]$ with $L < U$ in which we search for a valid lower bound value. If we can prove that no feasible solution with an objective value smaller than or equal to $T = \lfloor \frac{U+L}{2} \rfloor$ exists, we increase the lower bound L to the value $T+1$ and repeat the procedure with the interval $[T+1, U]$. Otherwise, we replace U by T and repeat the procedure in the interval $[L, T]$. As soon as $L \geq U$ holds, the procedure terminates. Then L equals the largest lower bound value which can be calculated in this way.

3.3.4 A destructive method for the multi-mode case

In this subsection we describe how the linear programming formulation from the previous subsection can be extended to the multi-mode case. In the corresponding preemptive relaxation we also allow that an activity may be processed in different modes. As described in Section 3.2.6, for each activity i and each mode $m \in \mathcal{M}_i$ we have a mode-dependent head r_i^m and a deadline $d_i^m := T - q_i^m + p_{im}$. Furthermore, mode-dependent time-lags $d_{ij}^{m\mu}$ may be given.

The time windows $[r_i^m, d_i^m]$ for activity-mode combinations (i, m) are first sharpened by constraint propagation as described in Section 3.2.6 and then used in the LP.

In a preprocessing step we eliminate all time windows $[r_i^m, d_i^m]$ with $d_i^m - r_i^m < p_{im}$ which are too small to process p_{im} time units in it. Let $z_0 < z_1 < \ldots < z_\tau$ be the ordered sequence of all other different r_i^m- and d_i^m-values with $d_i^m - r_i^m \geq p_{im}$. For $t = 1, \ldots, \tau$ we consider the intervals $I_t := [z_{t-1}, z_t]$ of length $z_t - z_{t-1}$. With each interval I_t we associate a set A_t of feasible activity-mode combinations (i, m) with $i = 1, \ldots, n$ and $m \in \mathcal{M}_i$ which can partially be executed in this interval, i.e. with $r_i^m \leq z_{t-1}$ and $z_t \leq d_i^m$.

A subset X of activity-mode combinations $(i, m) \in A_t$ is called **feasible** for the interval I_t if

- $(i, m) \in X$ implies $(i, m') \notin X$ for all $m' \in \mathcal{M}_i$ with $m' \neq m$, i.e. activity i is not processed in two different modes in X,
- for all pairs $(i, m), (j, \mu) \in X$ no precedence relation exists between activity i and activity j (or $d_{ij}^{m\mu} < p_{im}$ in the case that additionally mode-dependent time-lags are given),
- the renewable resource constraints are respected for the activities in X, i.e. we have

$$\sum_{(i,m) \in X} r_{ikm}^{\rho} \leq R_k^{\rho} \text{ for all } k \in \mathcal{K}^{\rho},$$

- the non-renewable resource constraints are respected for the activities in X if the remaining activities not contained in X only need their minimal resource requirements, i.e. we have

$$\sum_{(i,m)\in X} r^{\nu}_{ikm} + \sum_{j\in\overline{X}} \min_{\mu\in\mathcal{M}_j} \{r^{\nu}_{jk\mu}\} \leq R^{\nu}_k \text{ for all } k \in \mathcal{K}^{\nu}$$

  where $\overline{X} := \{ j \mid (j,\mu) \notin X \text{ for all } \mu \in \mathcal{M}_j\}$ denotes the set of activities which are not contained in X.

Let X_{lt} for $l = 1, \ldots, f_t$ be all these feasible subsets for the interval I_t and denote the set of all feasible subsets in I_t by F_t. With each set X_{lt} we associate an incidence vector a^{lt} defined by

$$a^{lt}_{im} = \begin{cases} 1, & \text{if } (i,m) \in X_{lt} \\ 0, & \text{otherwise} \end{cases} \quad \text{for activities } i = 1, \ldots, n \text{ and modes } m \in \mathcal{M}_i.$$

Let ξ_{im} $(i = 1, \ldots, n; m \in \mathcal{M}_i)$ be a 0-1 variable with

$$\xi_{im} = \begin{cases} 1, & \text{if activity } i \text{ is processed in mode } m \\ 0, & \text{otherwise.} \end{cases}$$

Furthermore, let x_{lt} be a variable denoting the number of time units where all activity-mode combinations in X_{lt} are executed simultaneously. Then the preemptive feasibility problem may be written as follows:

$$\sum_{m\in\mathcal{M}_i} \xi_{im} = 1 \qquad (i = 1, \ldots, n) \tag{3.63}$$

$$\sum_{i=1}^{n} \sum_{m\in\mathcal{M}_i} r^{\nu}_{ikm}\xi_{im} \leq R^{\nu}_k \qquad (k \in \mathcal{K}^{\nu}) \tag{3.64}$$

$$\sum_{t=1}^{\tau} \sum_{l=1}^{f_t} a^{lt}_{im} x_{lt} \geq p_{im}\xi_{im} \qquad (i = 1, \ldots, n;\ m \in \mathcal{M}_i) \tag{3.65}$$

$$\sum_{l=1}^{f_t} x_{lt} \leq z_t - z_{t-1} \qquad (t = 1, \ldots, \tau) \tag{3.66}$$

$$x_{lt} \geq 0 \qquad (t = 1, \ldots, \tau;\ l = 1, \ldots, f_t) \tag{3.67}$$

$$\xi_{im} \in \{0, 1\} \qquad (i = 1, \ldots, n;\ m \in \mathcal{M}_i) \tag{3.68}$$

Conditions (3.63) ensure that each activity i is assigned to exactly one mode $m \in \mathcal{M}_i$ and conditions (3.64) take care of the non-renewable resource constraints. In (3.65) the variables ξ_{im} are linked with the x_{lt}-variables by requiring that all activities i are processed in their assigned modes m for at least p_{im} time units. Finally, conditions (3.66) ensure that the number of time units scheduled in interval I_t does not exceed the length $z_t - z_{t-1}$ of this interval.

Since the integrality constraints (3.68) make the problem hard, we relax them and replace them by $\xi_{im} \geq 0$ (i.e. in this relaxation an activity may be processed

in different modes). Furthermore, we introduce artificial variables v_k for $k \in \mathcal{K}^\nu$ in conditions (3.64) which allow that the non-renewable resource contraints may be violated. We also introduce artificial variables u_t for $t = 1, \ldots, \tau$ in conditions (3.66) which allow that the capacities of the time intervals I_t may be exceeded. Then the feasibility problem can be formulated as the following linear program:

$$\min \quad \sum_{k \in \mathcal{K}^\nu} v_k + \sum_{t=1}^{\tau} u_t \tag{3.69}$$

$$\text{s.t.} \quad \sum_{m \in \mathcal{M}_i} \xi_{im} = 1 \qquad (i = 1, \ldots, n) \tag{3.70}$$

$$-\sum_{i=1}^{n} \sum_{m \in \mathcal{M}_i} r^\nu_{ikm} \xi_{im} + v_k \geq -R^\nu_k \qquad (k \in \mathcal{K}^\nu) \tag{3.71}$$

$$\sum_{t=1}^{\tau} \sum_{l=1}^{f_t} a^{lt}_{im} x_{lt} - p_{im} \xi_{im} \geq 0 \qquad (i = 1, \ldots, n;\ m \in \mathcal{M}_i) \tag{3.72}$$

$$-\sum_{l=1}^{f_t} x_{lt} + u_t \geq -z_t + z_{t-1} \qquad (t = 1, \ldots, \tau) \tag{3.73}$$

$$x_{lt} \geq 0 \qquad (t = 1, \ldots, \tau;\ l = 1, \ldots, f_t) \tag{3.74}$$

$$\xi_{im} \geq 0 \qquad (i = 1, \ldots, n;\ m \in \mathcal{M}_i) \tag{3.75}$$

$$v_k \geq 0 \qquad (k \in \mathcal{K}^\nu) \tag{3.76}$$

$$u_t \geq 0 \qquad (t = 1, \ldots, \tau) \tag{3.77}$$

A solution exists for the continuous relaxation of the preemptive feasibility problem if and only if the linear program (3.69) to (3.77) has the optimal solution value zero, i.e. if all values of the artifical variables become zero.

Example 3.14: Consider again Example 3.6. We assume that no renewable resources but one non-renewable resource k with $R^\nu_k = 5$ is given and that $r^\nu_{ikm} = m$ for all activity-mode combinations (i, m) holds. The longest path length $d_{04} = r_4 = 10$ (after improving heads and tails) provides a first simple lower bound for the given instance. For the threshold value $T = 11$ we get the following time windows $[r^m_i, d^m_i]$ with deadlines $d^m_i = T - q^m_i + p_{im}$:

$$\begin{aligned} &r^1_1 = 0, \quad r^2_1 = 0, \quad r^1_2 = 4, \quad r^2_2 = 5, \quad r^1_3 = 6, \quad r^2_3 = 7, \\ &d^1_1 = 4, \quad d^2_1 = 2, \quad d^1_2 = 5, \quad d^2_2 = 9, \quad d^1_3 = 11, \quad d^2_3 = 11, \end{aligned}$$

which lead to the feasible activity-mode combinations A_t in the time intervals I_t for $t = 1, \ldots, \tau = 7$ as shown in Figure 3.32.

Since a feasible subset X cannot contain the combinations $(1, 1)$ and $(1, 2)$ or $(3, 1)$ and $(3, 2)$ simultaneously (because an activity may not be processed in two different modes in X), we obtain 2+1+1+1+3+5+2=15 feasible subsets X_{lt}.

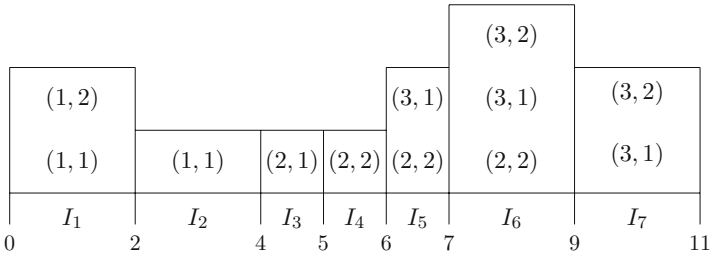


Figure 3.32: Feasible activity-mode combinations A_t in the intervals I_t

For example, in the interval $I_6 = [7, 9]$ we have 5 feasible subsets: $\{(2,2),(3,1)\}$, $\{(2,2),(3,2)\}$, $\{(2,2)\}$, $\{(3,1)\}$ and $\{(3,2)\}$. The subset $\{(2,2),(3,1)\}$ is feasible since $d_{23}^{21} = 1 < p_{22} = 3$ (i.e. no precedence relation $(2,2) \rightarrow (3,1)$ is induced by the time-lags) and $r_{2k2}^{\nu} + r_{3k1}^{\nu} + \min\limits_{m \in \mathcal{M}_1} \{r_{1km}^{\nu}\} = 2 + 1 + 1 = 4 \leq R_k^{\nu} = 5$ holds.

The LP-formulation (3.63) to (3.68) contains 6 variables ξ_{im} ($i = 1, 2, 3; m = 1, 2$) and 15 variables x_{lt} corresponding to the feasible subsets X_{lt}. It is easy to check that we get a feasible solution for (3.63) to (3.68) by setting $\xi_{12} = \xi_{21} = \xi_{32} = 1$ and processing the activity-mode combination $(1,2)$ in $[0,2]$ for one time unit, the combination $(2,1)$ in $[4,5]$ for one time unit and the combination $(3,2)$ in $[7,9]$ for two and in $[9,11]$ for one time unit.

For the threshold value $\tilde{T} = 10$ we get the deadlines $\tilde{d}_i^m = d_i^m - 1$, i.e.

$$\tilde{d}_1^1 = 3, \tilde{d}_1^2 = 1, \tilde{d}_2^1 = 4, \tilde{d}_2^2 = 8, \tilde{d}_3^1 = 10, \tilde{d}_3^2 = 10.$$

Due to $\tilde{d}_1^1 - r_1^1 = 3 < p_{11} = 4$, $\tilde{d}_2^1 - r_2^1 = 0 < p_{21} = 1$ and $\tilde{d}_3^1 - r_3^1 = 4 < p_{31} = 5$ only the activity-mode combinations $(1,2),(2,2),(3,2)$ remain feasible. But due to $r_{1k2}^{\nu} + r_{2k2}^{\nu} + r_{3k2}^{\nu} = 6 > R_k^{\nu} = 5$, no feasible mode assignment satisfying (3.64) exists. Thus, we may state infeasibility for $\tilde{T} = 10$, which leads to the improved lower bound $\tilde{T} + 1 = 11$. □

Again the linear program can be solved by column generation techniques. Since in general the number of columns corresponding to the variables ξ_{im} is not very large, all these columns may be generated at the beginning and kept in the working set during the whole procedure. Thus, only columns corresponding to the variables x_{lt} have to be generated by column generation techniques.

We solve the linear program with a two-phase method. At first we try to get a feasible fractional mode assignment (i.e. variables $0 \leq \xi_{im} \leq 1$ satisfying (3.70) and (3.71) with $v_k = 0$ for all $k \in \mathcal{K}^{\nu}$). If in the set of all variables ξ_{im} no such feasible fractional mode assignment exists, we can state infeasibility. Otherwise, we proceed with the second phase in which we perform column generation for the variables x_{lt} and try to get rid of the artificial variables u_t.

In the following we describe how the procedures from the previous subsection have to be modified:

`Initialize`: In order to get a feasible basic starting solution we determine for each activity $i = 1, \ldots, n$ an index $t(i)$ of an interval $I_{t(i)}$ with $I_{t(i)} \subseteq [r_i, d_i]$, where again $r_i := \min_{m \in \mathcal{M}_i} r_i^m$ and $d_i := \max_{m \in \mathcal{M}_i} d_i^m$. Besides all columns corresponding to the variables ξ_{im}, all artifical variables u_t, and all slack variables according to (3.71), (3.72) and (3.73) we add all columns corresponding to the one-element sets $\{(i, m)\}$ with $m \in \mathcal{M}_i$ to the initial working set of columns. Note that modes $m \in \mathcal{M}_i$ with $r_i^m > z_{t(i)-1}$ or $d_i^m < z_{t(i)}$ may not be executed in $I_{t(i)}$. For this reason, variables corresponding to infeasible activity-mode combinations $(i, m) \notin A_{t(i)}$ are treated as additional artificial variables and penalized with a positive coefficient in the objective function (3.69). Then in a basic starting solution we may process activity i in the interval $I_{t(i)}$ according to the fractional mode assignment ξ for $\max_{m \in \mathcal{M}_i} \{p_{im}\xi_{im}\}$ time units. If these processing times exceed the capacities of the intervals I_t, this is compensated by the artificial variables u_t in (3.73).

`InsertDeleteColumns`: We work iteratively with a restricted working set of columns which contains a basis, the columns corresponding to all variables ξ_{im}, all artifical variables u_t, and all slack variables according to (3.71), (3.72) and (3.73). We again use a delete-strategy which never deletes slack variables from the working set. Thus, only columns corresponding to the variables x_{lt} have to be calculated by column generation techniques.

`CalculateColumns`: We have to calculate new entering columns a^{lt} or to show that no improving column exists. If we denote the dual variables corresponding to conditions (3.72) by y_{im} $(i = 1, \ldots, n;\ m \in \mathcal{M}_i)$ and the dual variables belonging to (3.73) by w_t $(t = 1, \ldots, \tau)$, then the dual constraints corresponding to the variables x_{lt} have the form

$$\sum_{i=1}^{n} \sum_{m \in \mathcal{M}_i} a_{im}^{lt} y_{im} - w_t \leq 0 \qquad (t = 1, \ldots, \tau;\ l = 1, \ldots, f_t).$$

Thus, in order to find entering columns we have to determine columns a^{lt} with

$$\sum_{i=1}^{n} \sum_{m \in \mathcal{M}_i} a_{im}^{lt} y_{im} > w_t \tag{3.78}$$

for an index $t \in \{1, \ldots, \tau\}$. Feasible columns satisfying (3.78) are calculated by scanning iteratively through the intervals I_t $(t = 1, \ldots, \tau)$ and searching in each interval I_t for sets in F_t which correspond to improving columns. This can be done by an enumeration procedure as in Section 3.3.3, where the nodes of the enumeration tree correspond to feasible activity-mode combination $(i, m_i) \in A_t$.

3.3.5 Reference notes

The lower bound LB_S was proposed by Stinson et al. [137], the LP-based constructive lower bound by Mingozzi et al. [105]. To calculate the LP lower bound values, in [105] the dual linear program was considered and the corresponding weighted node packing problem was solved heuristically (which gives weaker bounds than the original LP formulation). The column generation procedure for the constructive LP bound is due to Baar et al. [5]. Brucker and Knust [23] strengthened the LP by taking into account time windows and solved the resulting LP by column generation in a destructive approach combined with constraint propagation. This approach was later improved by Baptiste and Demassey [8] who considered further constraint propagation techniques and introduced additional valid inequalities in the LP formulation. The extension of the destructive linear programming lower bound to the situation with multiple modes is due to Brucker and Knust [26].

Other constructive and destructive lower bounds for the RCPSP can be found in Klein and Scholl [83] who compared many bounds in computational experiments. Fisher [54] was the first who derived a lower bound based on a Lagrangian relaxation of the resource constraints. A similar bound was introduced by Christofides et al. [33] where the main difficulty consists in finding good Lagrangian multipliers. In this paper also two other lower bounds were proposed: the first is based on the LP relaxation of an integer programming formulation of the RCPSP with several additional valid inequalities to strengthen the LP relaxation. The second is based on a disjunctive graph by adding disjunctive arcs (representing resource conflicts) to the precedence constraints of the activity-on-node network.

Lower bounds for the RCPSP based on Lagrangian relaxation can also be found in Möhring et al. [108]. The idea of Stinson et al. was improved by Demeulemeester [39] who augmented a critical path by a second node-disjoint path and solved the resulting two-path relaxation by dynamic programming.

3.4 Heuristic Methods

In this section we discuss methods which calculate heuristic solutions for the RCPSP or more generally, for the RCPSP with time-dependent resource profiles. After giving a classification of different schedules in Section 3.4.1, we present so-called schedule generation schemes, which are used to generate a schedule from a given sequence of activities. Priority-based heuristics are discussed in Section 3.4.3. Foundations for local search and genetic algorithms can be found in Sections 3.4.4 and 3.4.5, the multi-mode case is reviewed in Section 3.4.6.

3.4.1 A classification of schedules

In the following we give a short classification of schedules for the RCPSP, which is useful to compare different exact and heuristic algorithms. Usually, the set of considered schedules is restricted to the set of **feasible** schedules, which are feasible with respect to the precedence and resource constraints. Moreover, if a regular objective function has to be minimized, activities may be shifted to the left without increasing the objective function value.

More specifically, a left shift of an activity i in a schedule S transforms S into a feasible schedule S' with $S'_i < S_i$ and $S'_j = S_j$ for all other activities $j \neq i$. If $S'_i = S_i - 1$ holds, the shift is called an **one-period left shift**. A **local left shift** is a shift which can be obtained by successive one-period left shifts, i.e. all intermediate schedules (in which the starting time of i is successively decreased by one time unit) have to be feasible. A left shift which is not a local left shift is called a **global left shift**. Then at least one intermediate schedule must be infeasible with respect to the resource constraints.

According to the notion of left shifts, schedules may be classified as follows. A feasible schedule is called **semi-active** if for all activities no local left shift is feasible. A feasible schedule is called **active** if for all activities no local or global left shift can be performed. A feasible schedule is called a **non-delay** schedule if for all activities no local or global left shift can be performed even if activities can be preempted at integer time points (i.e. only parts of activities are shifted).

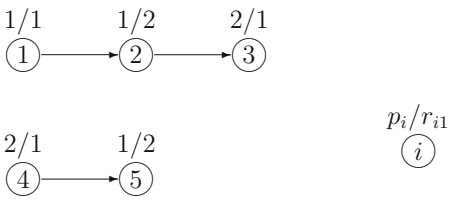
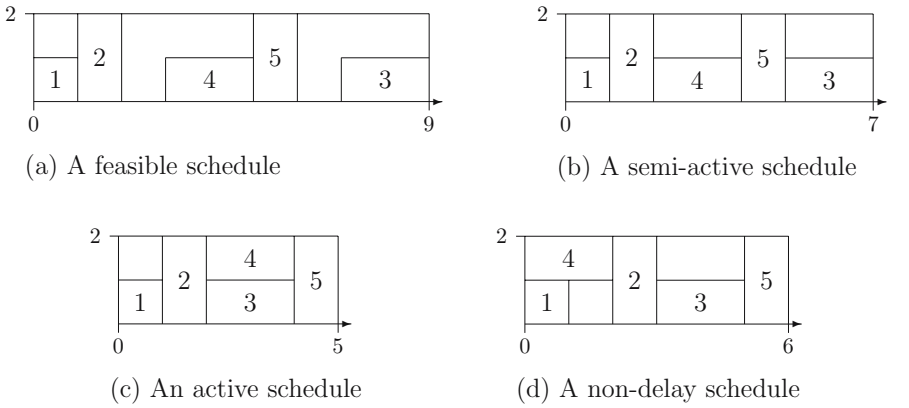


Figure 3.33: An RCPSP instance with $n = 5$ activities

Example 3.15: Consider the instance with $n = 5$ activities and one resource with constant capacity $R_1 = 2$ shown in Figure 3.33.



(a) A feasible schedule (b) A semi-active schedule

(c) An active schedule (d) A non-delay schedule

Figure 3.34: Four different schedules

A feasible schedule for this instance is depicted in Figure 3.34(a). This schedule is not semi-active since activities 4, 5 and 3 can be locally shifted to the left. By performing one-period left shifts for activities 4, 5 and two one-period left shifts of activity 3 (note that all intermediate schedules are feasible) the semi-active schedule shown in (b) is derived. For this schedule no further local left shift is possible, but activity 3 can be globally shifted to the left by three time periods. As a result the active schedule in (c) is obtained (which is the unique optimal solution for the given instance). This schedule is not a non-delay schedule, since the first unit of activity 4 can be feasibly shifted to the left starting at time 0. On the other hand, the schedule in (d) is a non-delay schedule, since no part of activity 3 can be feasibly shifted to the left due to the precedence relation $2 \rightarrow 3$. □

According to the previous definitions the non-delay schedules are a subset of all active schedules and the active schedules are a subset of the semi-active schedules. Moreover, each feasible schedule can be transformed into a semi-active schedule by a series of local left shifts. Furthermore, it can be proved that for any regular objective function (not only for the makespan) always an optimal active schedule exists.

Theorem 3.3 For the RCPSP with a regular objective function always an optimal active schedule exists.

Proof: Consider an optimal schedule S which is not active. Then an activity i exists which can be feasibly shifted to the left without violating the precedence or resource constraints. Let S' be the resulting schedule which is obtained from S by the left shift of i. We have $S'_i < S_i$ and $S'_j = S_j$ for all other activities $j \neq i$, i.e. for any regular objective function the schedule S' must also be optimal. By

repeating these arguments and shifting further activities to the left an optimal schedule is obtained for which no further left shift is possible, i.e. this schedule is an optimal active schedule. □

3.4.2 Schedule generation schemes

In order to apply an optimization algorithm to a problem, at first a suitable representation of solutions has to be chosen. Often, a schedule S is not directly represented by the corresponding starting times S_i of the activities since this representation has two drawbacks. On the one hand, the corresponding solution space is very large (if for each activity every starting time in the interval $[0, T]$ has to be considered, we get $O(T^n)$ possible schedules), on the other hand, it is often difficult to check feasibility of a given starting time vector. For this reason, schedules are often represented by sequences of the activities (so-called **activity lists**). From these lists feasible starting times are derived by appropriate decoding procedures (so-called **schedule generation schemes**). In the following we will present different schedule generation schemes for the RCPSP. Since it is not much more difficult to handle also time-dependent resource profiles, we dicuss this more general situation.

We assume that the time-dependent resource profile $R_k(t)$ of resource k is represented by pairs (t_k^μ, R_k^μ) for $\mu = 1, \ldots, m_k$, where $0 = t_k^1 < t_k^2 < \ldots < t_k^{m_k} = T$ are the jump points of the availability function and R_k^μ denotes the resource capacity in the time interval $[t_k^\mu, t_k^{\mu+1}[$ for $\mu = 1, \ldots, m_k - 1$. Note that it is already NP-complete to decide whether a feasible schedule satisfying the resource constraints exists. Since we ask for a feasible schedule with $C_{\max} \leq T$ for a given time horizon T, the decision version of the RCPSP (cf. Example 2.2) is a special case of this problem. In order to guarantee that a feasible solution exists, we assume that the last interval $[t^{m_k-1}, T]$ is chosen in such a way that enough space and sufficient resources are available to process all activities in this interval.

We represent solutions by lists of all activities which are compatible with the precedence relations, i.e. i is placed before j in the list if $i \to j \in C$ holds. With each list $L = (i_1, \ldots, i_n)$ of all activities we associate an "earliest start schedule" by planning the activities in the order induced by the list. This is done by the procedure `Earliest Start Schedule` $(i_1, \ldots, i_n)$ shown in Figure 3.35. For each activity j the procedure calculates the earliest time t where all predecessors of j are finished and sufficient resources are available. After scheduling j in the interval $[t, t + p_j[$ the resource profiles are updated and the next activity is considered.

In each iteration we have to check all jump points of the resource profiles at most once. Furthermore, scheduling an activity creates at most two new jump points where the resource profile changes. Thus, procedure `Earliest Start Schedule` can be implemented such that it needs at most $O(\sum_{\lambda=1}^{n} \sum_{k=1}^{r} (m_k + 2(\lambda - 1)) + |C|) =$

```
Procedure Earliest Start Schedule (i_1, ..., i_n)
1. FOR λ := 1 TO n DO
2.     j := i_λ;
3.     t := max_{i→j∈C} {S_i + p_i};
4.     WHILE a resource k with r_jk > R_k(τ) for a τ ∈ [t, t + p_j[
       exists DO
5.        Calculate the smallest time t_k^μ > t such that j
          can be scheduled in the interval [t_k^μ, t_k^μ + p_j[ if
          only resource k is considered and set t := t_k^μ;
6.     ENDWHILE
7.     Schedule j in the interval [S_j, C_j[ := [t, t + p_j[;
8.     Update the current resource profiles by setting
       R_k(τ) := R_k(τ) − r_jk for all k = 1, ..., r and τ ∈ [t, t + p_j[;
9. ENDFOR
```

Figure 3.35: Calculation of an earliest start schedule for a given list

$O(n^2 r + n \sum_{k=1}^{r} m_k)$ time, i.e. it runs in polynomial time. For the RCPSP with constant resource capacities (i.e. $m_k = 1$ for all resources k) the running time reduces to $O(n^2 r)$.

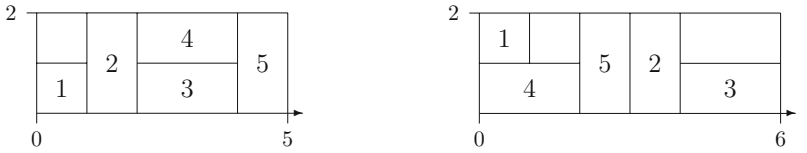
It remains to show that with this procedure a dominant set of schedules is achieved, i.e. that in the set of schedules which can be generated by applying the list scheduling procedure to all possible lists, always an optimal solution exists. By construction of the procedure `Earliest Start Schedule` each activity is processed as early as possible (respecting the precedence and resource constraints). Thus, in the constructed schedules no activity can be (locally or globally) shifted to the left, i.e. the list schedules are active.

Theorem 3.4 For the RCPSP with a regular objective function a list of all activities exists such that the procedure `Earliest Start Schedule` provides an optimal schedule.

Proof: Let S be an optimal schedule and order the activities according to non-decreasing starting times in S. If we apply procedure `Earliest Start Schedule` to this list, a feasible schedule S' is produced. Furthermore, we have $S'_j \leq S_j$ for all activities $j = 1, \ldots, n$ since each activity is processed as early as possible in S'. Thus, for any regular objective function the schedule S' must also be optimal. □

Example 3.16: Consider again the instance from Example 3.15 with $n = 5$ activities and one resource with constant capacity $R_1 = 2$.

In Figure 3.36 the corresponding list schedules for the lists $(1, 2, 3, 4, 5)$ and $(4, 5, 1, 2, 3)$ are shown. For the lists $(1, 2, 4, 3, 5)$ and $(4, 1, 5, 2, 3)$ also the sched-



(a) List schedule for $(1, 2, 3, 4, 5)$ (b) List schedule for $(4, 5, 1, 2, 3)$

Figure 3.36: Two list schedules for Example 3.15

ules in (a) and (b), respectively, are generated. This shows that in general the scheduling procedure does not produce different schedules for different lists. □

Another list scheduling procedure, which is often used for scheduling problems, also produces a dominant set of schedules for the described situation. This procedure generates schedules where the starting times are ordered according to the given list, i.e. $S_{i_1} \leq S_{i_2} \leq \ldots \leq S_{i_n}$ holds. The resulting schedules are in general not active, but with the same arguments as in the proof of Theorem 3.4 it can be shown that for any regular objective function an optimal schedule is contained in the set of all schedules generated by this procedure.

In order to calculate such schedules in Step 3 of the procedure `Earliest Start Schedule` we only have to set t equal to the maximum of $\max\limits_{i \to j \in C} \{S_i + p_i\}$ and the starting time $S_{i_{\lambda-1}}$ of the previous activity. The resulting complexity is $O(\sum\limits_{k=1}^{r}(m_k + n) + |C|) = O(nr + n^2 + \sum\limits_{k=1}^{r} m_k)$ since during the check of the resource profiles we never have to go back to previous time periods.

The first list scheduling procedure described above may be seen as a special case of the so-called "serial schedule generation scheme" for the RCPSP. A **schedule generation scheme** (SGS) iteratively generates a feasible schedule by extending a partial schedule in each iteration (in general not always a list is needed). In the literature two schemes are distinguished:

- the activity-oriented **serial SGS**, where in each iteration one activity is scheduled, and
- the time-oriented **parallel SGS**, where in each iteration a new decision point on the time axis is considered and a subset of activities is scheduled at this time point.

Schedule generation schemes are a basic component of heuristics based on priority rules. In such heuristics the activities are iteratively planned and in each iteration an eligible activity is chosen according to some priority rule. More details on priority-based heuristics will be described in Subsection 3.4.3.

In the **serial schedule generation scheme** a schedule is generated in n stages. With each stage $\lambda \in \{1, \ldots, n\}$ two disjoint activity sets are associated: the set of scheduled activities and the set E_λ of all eligible activities (i.e. all activities

for which all predecessors are scheduled). In each stage one eligible activity $j \in E_\lambda$ is chosen and scheduled at the earliest precedence- and resource-feasible time. Afterwards, the resource profiles of the partial schedule and the set of eligible activities are updated. The serial SGS is summarized in Figure 3.37. Similar to the procedure `Earliest Start Schedule` it can be implemented in such a way that it runs in $O(n^2 r + n \sum_{k=1}^{r} m_k)$ time.

```
Procedure Serial Schedule Generation Scheme
 1. Let E_1 be the set of all activities without predecessor;
 2. FOR λ := 1 TO n DO
 3.    Choose an activity j ∈ E_λ;
 4.    t := max_{i→j∈C} {S_i + p_i};
 5.    WHILE a resource k with r_jk > R_k(τ) for a τ ∈ [t, t + p_j[
       exists DO
 6.       Calculate the smallest time t_k^μ > t such that j
          can be scheduled in the interval [t_k^μ, t_k^μ + p_j[ if
          only resource k is considered and set t := t_k^μ;
 7.    ENDWHILE
 8.    Schedule j in the interval [S_j, C_j[ := [t, t + p_j[;
 9.    Update the current resource profiles by setting
       R_k(τ) := R_k(τ) − r_jk for all k = 1, ..., r and τ ∈ [t, t + p_j[;
10.    Let E_{λ+1} := E_λ \ {j} and add to E_{λ+1} all successors
       i ∉ E_λ of j for which all predecessors are scheduled;
11. ENDFOR
```

Figure 3.37: Serial schedule generation scheme

Example 3.17: Consider again the instance from Example 3.15. If in Step 3 we always choose an eligible activity with smallest index, then the serial SGS generates the schedule shown in Figure 3.36(a).

In the first iteration we choose activity 1 from the set of eligible activities $E_1 = \{1, 4\}$ and schedule it at time 0. Afterwards, we have $E_2 = \{2, 4\}$ and schedule activity 2 at time 1. In stage $\lambda = 3$ we get $E_3 = \{3, 4\}$ and schedule activity 3 at time 2. Then $E_4 = \{4\}$ and activity 4 is scheduled also at time 2. Finally, activity 5 is scheduled at time 4. □

While the serial SGS is activity-oriented, the **parallel schedule generation scheme** is time-oriented. With each stage λ a time point t_λ and three disjoint activity sets are associated: the set of finished activities, the set A_λ of all active activities (i.e. activities which are already scheduled in the partial schedule, but finish after time t_λ), and the set E_λ of all eligible activities (i.e. all unscheduled activities i for which all predecessors are completed up to time t_λ and for which sufficient resources are available when i is started at time t_λ).

In each stage a maximal resource-feasible subset of eligible activities in E_λ is chosen and scheduled at time t_λ. Afterwards, the resource profiles of the partial schedule and the sets of active and eligible activities are updated. The next decision point $t_{\lambda+1}$ is given by the minimum of the next value t_k^μ where a resource profile changes and the minimal completion time of all active activities.

The parallel SGS is summarized in Figure 3.38. As the serial SGS, it generates feasible schedules in $O(n^2r + n\sum_{k=1}^{r} m_k)$ time.

```
Procedure Parallel Schedule Generation Scheme
 1. λ := 1; t_1 := 0; A_1 := ∅;
 2. Let E_1 be the set of all activities i without predecessor
    which can be scheduled in [0, p_i] respecting the resources;
 3. WHILE not all activities are scheduled DO
 4.   WHILE E_λ ≠ ∅ DO
 5.     Choose an activity j ∈ E_λ;
 6.     Schedule j in the interval [S_j, C_j[ := [t_λ, t_λ + p_j[;
 7.     Update the current resource profiles and add j to A_λ;
 8.     Update the set E_λ by eliminating j and all
        activities i which cannot be resource-feasibly
        scheduled in [t_λ, t_λ + p_i];
 9.   ENDWHILE
10.   Let t_{λ+1} be the minimum of the smallest value t_k^μ > t_λ
      and min_{i∈A_λ} {S_i + p_i};
11.   λ := λ + 1;
12.   Calculate the new sets A_λ and E_λ;
13. ENDWHILE
```

Figure 3.38: Parallel schedule generation scheme

Example 3.18: Consider again the instance from Example 3.15. If in Step 5 we always choose an eligible activity with smallest index, then the parallel SGS generates the schedule shown in Figure 3.34(d).

In the first iteration we consider time $t_1 = 0$ and schedule activity 1 and then activity 4 from the set of eligible activities $E_1 = \{1, 4\}$. The next decision point is given by the completion time of activity 1, i.e. $t_2 = 1$. Since at this time no activity can be scheduled, we proceed with $t_3 = 2$. Then we have $E_3 = \{2, 5\}$ and schedule activity 2. Since afterwards activity 5 cannot be processed, we go to $t_4 = 3$ and $E_4 = \{3, 5\}$. After scheduling activity 3 at time 3, in the last stage activity 5 is scheduled at the next decision point $t_5 = 5$. □

For the RCPSP with constant resource capacities contrary to the serial SGS (which may generate all active schedules), the parallel SGS generates only a

subset of all active schedules, namely the set of non-delay schedules (cf. Section 3.4.1).

Theorem 3.5 For the RCPSP with constant resource capacities the parallel SGS generates non-delay schedules.

Proof: To prove that the parallel SGS only generates non-delay schedules, assume to the contrary that it generates a schedule S which is not a non-delay schedule. Thus, in S some part of an activity j can be feasibly shifted to the left to some time $t < S_j$. W.l.o.g. we may assume $t = t_\lambda$ for some index λ since due to the constant resource capacities the starting times of activities may be restricted to completion times of other activities. Because the left shift is feasible, all predecessors must be completed at time t_λ and sufficient resources have to be available in the interval $[t_\lambda, t_\lambda + 1]$. Since constant resource profiles are given, in stage λ of the parallel SGS the resource availabilities from interval $[t_\lambda, t_\lambda + 1]$ to $[t_\lambda, t_\lambda + p_i]$ do not decrease. Thus, at the end of stage λ activity j was contained in the set E_λ of eligible activities. Although it could be scheduled in $[t_\lambda, t_\lambda + p_i]$, it was not scheduled which contradicts Step 4 of the parallel SGS, where activities are scheduled until the set E_λ becomes empty. □

Note that this property does not hold for the RCPSP with time-dependent resource profiles since in Step 8 activities i are eliminated from E_λ if they cannot be resource-feasibly scheduled in the interval $[t_\lambda, t_\lambda + p_i]$. However, parts of i may be processed in an interval $[t_\lambda, t_\lambda + x]$ for some $x \geq 1$.

Since the set of non-delay schedules is a subset of the set of active schedules, in general the solution space of the parallel SGS is smaller than the solution space of the serial SGS. But unfortunately, it may happen that the set of considered schedules does not contain an optimal solution. This can be seen at the instance from Example 3.15. Its unique optimal schedule is the active schedule shown in Figure 3.34(c) with makespan 5. Since this schedule is not a non-delay schedule, for this instance the set of non-delay schedules does not contain an optimal solution.

However, from this observation it cannot be deduced that in general the serial generation scheme is superior to the parallel one. Computational experiments of various authors have shown that for some instances the serial SGS produces better schedules, for other instances the parallel SGS is more suitable.

Another variant of the proposed schemes are their **backward** counterparts. While in the described (forward) schemes, the schedules are generated from left to right, in the corresponding backward schemes the schedules are constructed in the reverse direction from right to left. Finally, in a **bidirectional scheduling** procedure the forward and the backward scheme are combined. In each iteration an activity is scheduled either in a forward or in a backward schedule. At the end both schedules are concatenated and transformed into an active schedule by moving activities to the left.

The idea of scheduling activities from both directions is also used in so-called **forward-backward improvement** procedures. After applying a forward SGS

to a given list, a new list is created by sorting the activities according to non-increasing completion times in the forward schedule. This new list is used for a backward scheduling step and after sorting the activities according to non-decreasing starting times in the backward schedule again the forward SGS is applied. This process may be iterated until no new schedules are created. Finally, the best schedule generated during the process is taken as output for the original list.

3.4.3 Priority-based heuristics

As described in the previous subsection, the main two components of priority-based heuristics for the RCPSP are schedule generation schemes and priority rules. In a schedule generation scheme in each stage an activity is selected which is scheduled next in the partial schedule. For this purpose, a priority value is calculated for each activity and an activity with smallest (or largest) value is chosen. In case of ties (i.e. if several activities have the same priority value), an additional rule is used to choose a unique activity.

activity-based:	
SPT	choose an activity with the smallest processing time
LPT	choose an activity with the largest processing time

network-based:	
MIS	choose an activity with the most immediate successors
LIS	choose an activity with the least immediate successors
MTS	choose an activity with the most total successors
LTS	choose an activity with the least total successors
GRPW	choose an activity with the greatest rank positional weight, i.e. with the largest total processing time of all successors

critical path-based:	
EST	choose an activity with the smallest earliest starting time
ECT	choose an activity with the smallest earliest completion time
LST	choose an activity with the smallest latest starting time
LCT	choose an activity with the smallest latest completion time
MSLK	choose an activity with a minimum slack

resource-based:	
GRR	choose an activity with the greatest resource requirements

Figure 3.39: Priority rules for the RCPSP

Many different priority rules for scheduling problems have been proposed in the literature. They may be classified as activity-based, network-based, critial

path-based or resource-based (depending on the information which is used by the rule). In Figure 3.39 some often used rules are listed.

Another distinction of priority rules is based on the dynamics of a rule. For example, the SPT-rule is always static since the processing time of an activity does not change over time. On the other hand, the EST-rule may be used in a static or dynamic way: in the static case the earliest starting times are calculated once (only based on the project network), in the dynamic case they are recalculated after each scheduling step (taking into account the current partial schedule).

Additionally, priority-based heuristics may be classified as single- or multi-pass methods. While in **single-pass methods** only one schedule is generated (using a single priority rule and a single SGS), in **multi-pass methods** several schedules are generated (for example, by using several priority rules, different generation schemes or different scheduling directions).

Priority-based heuristics are often used in practice since they have small running times and are easy to implement. Furthermore, they are often used to calculate a starting solution for improvement procedures like local search or genetic algorithms, which will be discussed next.

3.4.4 Local search algorithms

In this subsection we describe some problem-specific parts of local search algorithms for the RCPSP with time-dependent resource profiles. As discussed in Section 2.7, the problem-specific parts of local search algorithms are the representation of solutions and the definition of appropriate neighborhoods (which are usually defined by operators).

As described in Subsection 3.4.2, a solution for the RCPSP with time-dependent resource profiles may be represented by an activity list (sequence of the activities), which is compatible with the precedence constraints. In order to change such a list $L = (i_1, \ldots, i_n)$ in a local search procedure, the following neighborhoods may be defined:

Adjacent pairwise interchange-neighborhood

The adjacent pairwise interchange (api)-neighborhood $\mathcal{N}_{api}$ is defined by operators api_λ for $\lambda = 1, \ldots, n-1$, where api_λ interchanges the elements i_λ and $i_{\lambda+1}$ in L.

For example, by applying api_λ with $\lambda = 4$ to $L = (2, 1, 5, 3, 7, 4, 6)$, we get $L' = (2, 1, 5, 7, 3, 4, 6)$ (cf. Figure 3.40).

Since we only consider lists which are compatible with the precedence constraints, such an interchange is only feasible if no precedence relation $i_{\lambda+1} \rightarrow i_\lambda$ exists. Since we have at most $n-1$ feasible *api*-operators for a list, the size of the neighborhood $\mathcal{N}_{api}$ is bounded by $O(n)$.

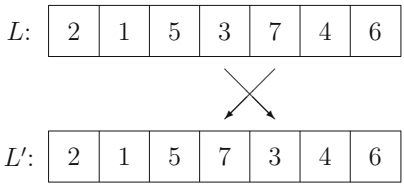


Figure 3.40: Operator api_λ for $\lambda = 4$

Swap-neighborhood

The swap-neighborhood $\mathcal{N}_{swap}$ generalizes the neighborhood $\mathcal{N}_{api}$ and is defined by operators $swap_{\lambda,\mu}$ for $\lambda, \mu = 1, \ldots, n; \lambda < \mu$, where $swap_{\lambda,\mu}$ interchanges the elements i_λ and i_μ in L.

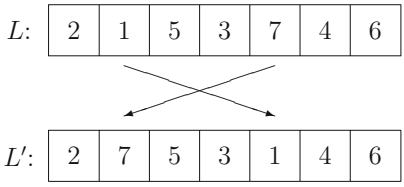


Figure 3.41: Operator $swap_{\lambda,\mu}$ with $\lambda = 2, \mu = 5$

For example, by applying $swap_{\lambda,\mu}$ with $\lambda = 2, \mu = 5$ to $L = (2, 1, 5, 3, 7, 4, 6)$, we get $L' = (2, 7, 5, 3, 1, 4, 6)$ (cf. Figure 3.41).

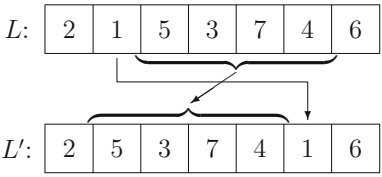
Such an interchange is only feasible if for all $\nu \in \{\lambda+1, \ldots, \mu-1\}$ no precedence relation $i_\lambda \to i_\nu$ or $i_\nu \to i_\mu$ exists. Since we have at most $n(n-1)/2$ feasible *swap*-operators for a permutation, the size of the neighborhood $\mathcal{N}_{swap}$ is bounded by $O(n^2)$.

Shift-neighborhood

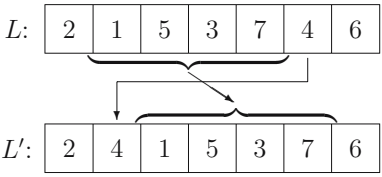
The shift-neighborhood $\mathcal{N}_{shift}$ also generalizes the neighborhood $\mathcal{N}_{api}$ and is defined by operators $shift_{\lambda,\mu}$ for $\lambda, \mu = 1, \ldots, n; \lambda \neq \mu$, where $shift_{\lambda,\mu}$ shifts the element i_λ to position μ in L. For $\lambda < \mu$ we have a right shift, symmetrically, for $\lambda > \mu$ we have a left shift.

For example, by applying $shift_{\lambda,\mu}$ with $\lambda = 2, \mu = 6$ to $L = (2, 1, 5, 3, 7, 4, 6)$, we get $L' = (2, 5, 3, 7, 4, 1, 6)$ (cf. Figure 3.42(a)). By applying $shift_{\lambda,\mu}$ with $\lambda = 6, \mu = 2$ to $L = (2, 1, 5, 3, 7, 4, 6)$, we get $L' = (2, 4, 1, 5, 3, 7, 6)$ (cf. Figure 3.42(b)).

Such shifts are only feasible if for no $\nu \in \{\lambda + 1, \ldots, \mu\}$ a precedence relation $i_\lambda \to i_\nu$ and for no $\nu \in \{\mu, \ldots, \lambda - 1\}$ a relation $i_\nu \to i_\lambda$ exists. Since we have at most $n(n-1)/2$ feasible *shift*-operators for a permutation, the size



(a) Operator $shift_{\lambda,\mu}$ for $\lambda = 2 < \mu = 6$



(b) Operator $shift_{\lambda,\mu}$ for $\lambda = 6 > \mu = 2$

Figure 3.42: Operators $shift_{\lambda,\mu}$

of the neighborhood $\mathcal{N}_{shift}$ is bounded by $O(n^2)$. Furthermore, note that a swap-operator can be performed by two consecutive shift-operators.

3.4.5 Genetic algorithms

In this subsection we describe some problem-specific parts of genetic algorithms for the RCPSP with time-dependent resource profiles. For genetic algorithms the population POP may be defined as a set of precedence-feasible sequences (activity lists). Mutations may be achieved by performing some iterations of simulated annealing or tabu search using one of the neighborhoods from the previous subsection. In the following we describe three commonly used basic crossover operators.

One-point crossover

Given two parent sequences, a mother sequence $L^M = (i_1^M, \ldots, i_n^M)$ and a father sequence $L^F = (i_1^F, \ldots, i_n^F)$, two child sequences, a daughter sequence $L^D = (i_1^D, \ldots, i_n^D)$ and a son sequence $L^S = (i_1^S, \ldots, i_n^S)$ are created by the following procedure. Let $q \in \{1, \ldots, n\}$ be a random number. Then the daughter sequence L^D inherits the positions $\lambda = 1, \ldots, q$ from the mother sequence L^M, i.e.

$$i_\lambda^D := i_\lambda^M \text{ for } \lambda = 1, \ldots, q.$$

The elements in positions $\lambda = q+1, \ldots, n$ are taken from the father sequence L^F in such a way that

$$i_\lambda^D := i_k^F \text{ where } k \text{ is the smallest index with } i_k^F \notin \{i_1^D, \ldots, i_{\lambda-1}^D\}.$$

The son sequence L^S is symmetrically constructed by interchanging the roles of L^M and L^F.

For example, with $L^M = (7, 1, 3, 2, 5, 8, 4, 6, 9)$, $L^F = (1, 4, 2, 6, 3, 9, 8, 7, 5)$, and $q = 4$ we get $L^D = (7, 1, 3, 2, 4, 6, 9, 8, 5)$ (cf. Figure 3.43).

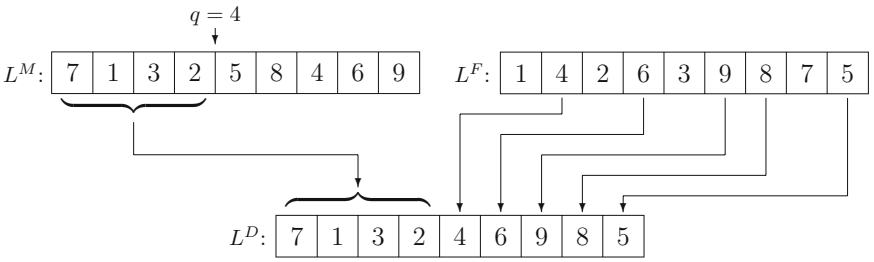


Figure 3.43: One-point crossover

If L^M and L^F are compatible with the precedence constraints, then L^D also has this property. For the first subsequence $I_1 = (i_1^D, \ldots, i_q^D)$ of L^D this property is inherited from L^M, for the second subsequence $I_2 = (i_{q+1}^D, \ldots, i_n^D)$ from L^F. Furthermore, it is not possible that $i \rightarrow j$ with $i \in I_2,\ j \in I_1$ holds, since this would contradict the fact that L^M is compatible with the precedence constraints.

Two-point crossover

Firstly, we draw two random numbers $q_1, q_2 \in \{1, \ldots, n\}$ with $q_1 < q_2$. The daughter sequence L^D inherits positions $1, \ldots, q_1$ and $q_2 + 1, \ldots, n$ from the mother L^M. The remaining positions $q_1 + 1, \ldots, q_2$ are taken from the father sequence L^F. They are ordered in L^D in the same way as they are ordered in L^F. The son sequence L^S is constructed by changing the roles of L^M and L^F.

For example, with $L^M = (7, 1, 3, 2, 5, 8, 4, 6, 9)$, $L^F = (1, 4, 2, 6, 3, 9, 8, 7, 5)$, and $q_1 = 2, q_2 = 6$ we get $L^D = (7, 1, 2, 3, 8, 5, 4, 6, 9)$ (cf. Figure 3.44).

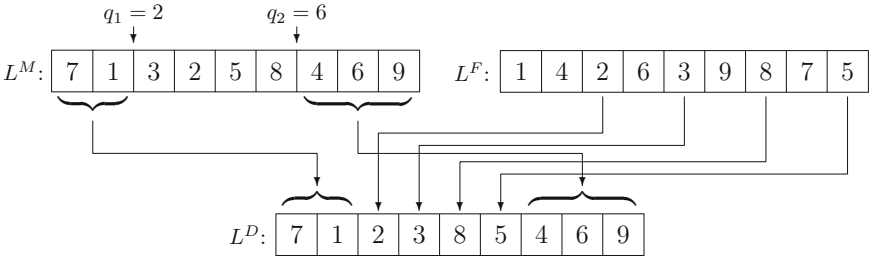


Figure 3.44: Two-point crossover

Again, it is easy to show that the daughter sequence is compatible with the precedence constraints if L^M and L^F have this property.

Another possibility for a two-point crossover would be the following. Again, two random numbers $q_1, q_2 \in \{1, \ldots, n\}$ with $q_1 < q_2$ are chosen. The daughter sequence L^D inherits positions $q_1, \ldots, q_2$ from the mother L^M. The remaining positions $1, \ldots, q_1 - 1$ and $q_2 + 1, \ldots, n$ are filled with those elements from L^F which are not contained in $i^M_{q_1}, \ldots, i^M_{q_2}$ (in the same order as in L^F).

For example, with $L^M = (1, 2, 3, 4, 5, 6, 7)$, $L^F = (3, 4, 5, 6, 1, 2, 7)$, and $q_1 = 3, q_2 = 4$ we get $L^D = (5, 6, 3, 4, 1, 2, 7)$ (cf. Figure 3.45). This example shows that in general such a crossover does not provide a sequence which is compatible with the precedence constraints. For example, the precedences $3 \to 6$ and $4 \to 6$ are satisfied in both parents L^M and L^F, but they are not respected in L^D.

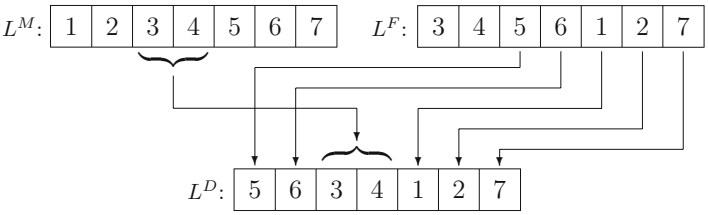


Figure 3.45: A non-compatible two-point crossover

Uniform crossover

For $\lambda = 1, \ldots, n$ we randomly draw values $\xi_\lambda \in \{0, 1\}$ and fill the positions in L^D in the following way:

If $\xi_\lambda = 1$, then we choose from L^M the activity which has the smallest index and is not yet inserted in positions $1, \ldots, \lambda - 1$ in L^D, i.e.

$$i^D_\lambda := i^M_k \text{ where } k \text{ is the smallest index with } i^M_k \notin \{i^D_1, \ldots, i^D_{\lambda-1}\}.$$

If $\xi_\lambda = 0$, then we set

$$i^D_\lambda := i^F_k \text{ where } k \text{ is the smallest index with } i^F_k \notin \{i^D_1, \ldots, i^D_{\lambda-1}\}.$$

For the construction of L^S we again change the roles of L^M and L^F.

For example, with $L^M = (7, 1, 3, 2, 5, 8, 4, 6, 9)$, $L^F = (1, 4, 2, 6, 3, 9, 8, 7, 5)$, and $\xi = (1, 1, 0, 1, 0, 0, 1, 0, 1)$ we get $L^D = (7, 1, 4, 3, 2, 6, 5, 9, 8)$.

Also for this crossover the children are compatible with the precedence constraints if the parents have this property. This can be proved as follows. Assume that in L^D an activity i with $j \to i$ is placed before j. Then in L^M and L^F activity j must be placed before i. Is i inherited from L^F, then we have a contradiction to the fact that j is placed after i in L^D, since then j had to be scheduled in L^D instead of i. We get a similar contradiction if i is inherited from L^M.

In connection with genetic algorithms and schedule generation schemes for the RCPSP also so-called **self-adapting genetic algorithms** were proposed. In such algorithms the activity list representation is extended by an additional gene which determines whether the serial or parallel SGS is used as decoding procedure. Then during the selection process the algorithm tries to "learn" which schedule generation scheme is better.

3.4.6 Heuristics for the multi-mode case

In order to solve the multi-mode RCPSP heuristically, often a procedure with two stages is applied. While in one stage the mode assignment is determined, in the other stage a corresponding schedule is calculated. If this is done in a hierarchical way (first fixing the modes), in the second stage a classical single-mode RCPSP has to be solved.

The genetic algorithm based on activity lists has been extended to the multi-mode RCPSP by introducing an additional mode list M which assigns a mode to each activity. From such a solution (M, L) a schedule is derived by fixing the modes according to M and applying a schedule generation scheme to the activity list L afterwards. Different mutation and crossover operators have been proposed for M. For example, in a mutation the mode of a single activity may be changed. In the crossover operators mode sequences are inherited from the parents (e.g. the modes for the first q activities are taken from the mother, the remaining ones from the father).

3.4.7 Reference notes

The classification into semi-active, active and non-delay schedules is due to Sprecher et al. [136]. The first heuristic for the RCPSP based on priority rules was proposed by Kelley [78] in 1963. A review of heuristics developed before 1973 can be found in Davies [37], a review of priority-based heuristics up to 1989 in Alvarez-Valdes and Tamarit [4]. Different methods based on the serial and the parallel schedule generation scheme are reviewed and computationally tested by Kolisch [85]. The described genetic algorithms are due to Hartmann [64], [66], [67]. Recent overviews about different heuristics (also including local search and genetic algorithms) and a computational comparison of them can be found in Hartmann and Kolisch [69] as well as in Kolisch and Hartmann [86], [87].

3.5 Branch-and-Bound Algorithms

In this section we will discuss different exact methods to solve the RCPSP with constant resource profiles. They are all based on the general branch-and-bound principle, i.e. they partition the problem into subproblems and try to eliminate subproblems by calculating lower bounds or using dominance rules.

A branch-and-bound algorithm based on so-called precedence trees is presented in Section 3.5.1. In Sections 3.5.2 and 3.5.3 algorithms based on extension and delaying alternatives can be found. In Section 3.5.4 so-called schedule schemes are introduced, in Section 3.5.5 briefly the multi-mode case is reviewed.

3.5.1 An algorithm based on precedence trees

In this subsection we present a branch-and-bound algorithm which is based on the list scheduling techniques introduced in Section 3.4.2. The nodes of the enumeration tree correspond to precedence-feasible subsequences of activities which are extended by an unscheduled activity in each branching step. Associated with each node is the earliest start schedule for the corresponding subsequence in which all activities are scheduled according to the subsequence and no activity i is started before the starting time of the previously scheduled activity (i.e. for a list $(i_1, \ldots, i_n)$ we have $S_{i_1} \leq S_{i_2} \leq \ldots \leq S_{i_n}$). As mentioned in Section 3.4.2, for a regular objective function always an optimal schedule exists in the set of such schedules. Since in the enumeration tree all sequences which are compatible with the precedence relations are enumerated, the resulting tree is also called **precedence tree**. This tree has $n+2$ levels because in each stage one activity is scheduled (including the dummy activities 0 and $n+1$) and the leafs correspond to complete schedules.

With each node of the enumeration tree in stage $\lambda \in \{1, \ldots, n+1\}$ two disjoint activity sets are associated: the set of scheduled activities F_λ and the set E_λ of all eligible activities (i.e. all unscheduled activities for which all predecessors are scheduled). Then an eligible activity $j \in E_\lambda$ is chosen and scheduled at the earliest precedence- and resource-feasible time t which is not smaller than the starting time of the previously scheduled activity in stage $\lambda - 1$. Afterwards, the resource profiles of the partial schedule are updated and the set of eligible activities $E_{\lambda+1}$ is determined. Then the next activity is chosen in stage $\lambda + 1$.

If stage $\lambda = n + 1$ is reached, all activities are scheduled and the makespan of the corresponding complete schedule is compared to the best upper bound value found so far. Afterwards, backtracking to the previous stage occurs where the next unconsidered eligible activity is chosen. If all eligible activities have been considered as branching candidates in a stage, a further backtracking step to the previous stage is performed.

Example 3.19: Consider the instance with $n = 4$ activities and one resource with capacity $R_1 = 3$ shown in Figure 3.46. Due to the precedence relation

$2 \rightarrow 4$ only sequences in which 2 is placed before 4 are feasible. The resulting precedence tree can be found in Figure 3.47, where the level of the dummy activity $n + 1 = 5$ is not drawn. In the leafs of this tree the six different schedules $S_1, \ldots, S_6$ shown in Figure 3.48 are obtained.

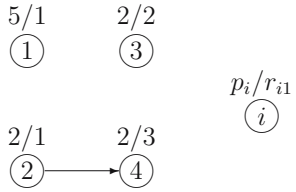


Figure 3.46: An instance with $n = 4$ activities

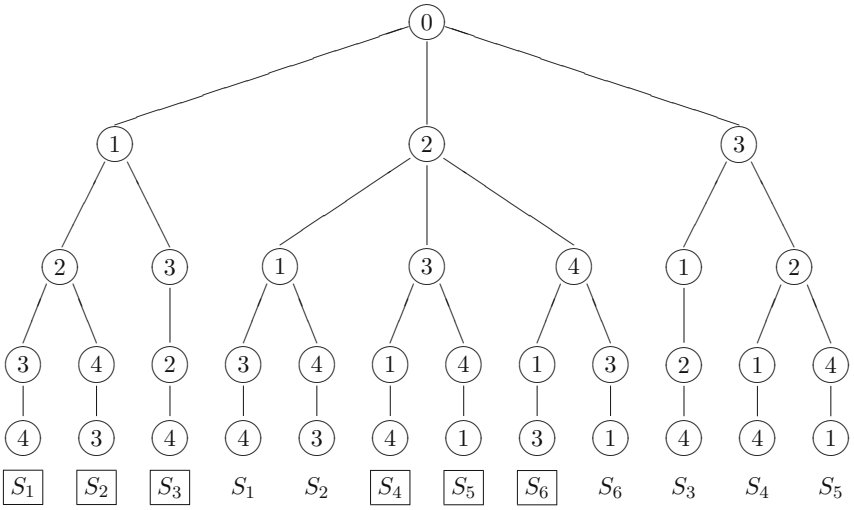


Figure 3.47: Precedence tree for the instance in Example 3.19

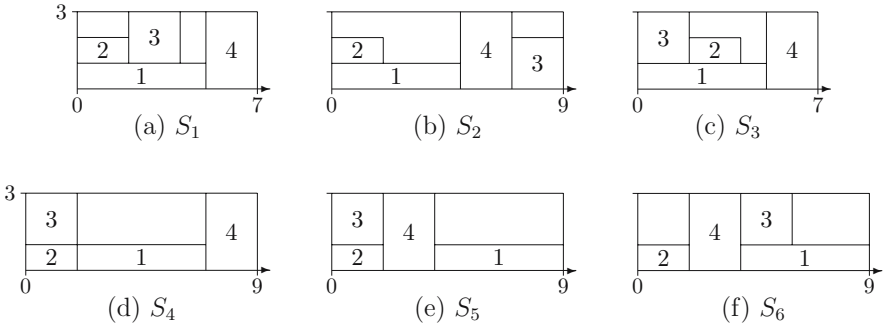


Figure 3.48: Schedules obtained by the precedence tree algorithm

□

The preceding discussions are summarized in the algorithm shown in Figure 3.49.

```
Algorithm Branch-and-Bound Precedence Tree
 1.  Calculate an initial upper bound UB for the instance;
 2.  S_0 := 0; λ := 1; F_1 := {0}; s_0 := 0;
 3.  Let E_1 be the set of all activities without predecessor;
 4.  B&B (λ, F_1, E_1, S, s_0);

Procedure B&B (λ, F_λ, E_λ, S, s_{λ-1})
 1.  WHILE E_λ ≠ ∅ DO
 2.     Choose an activity j ∈ E_λ;
 3.     Calculate the smallest time t ≥ s_{λ-1} such that j
        can be scheduled in the interval [t, t + p_j[ without
        violating resource or precedence constraints;
 4.     Schedule j in the interval [t, t + p_j[ and set S_j := t;
 5.     F_λ := F_λ ∪ {j};   E_λ := E_λ \ {j};   s_λ := t;
 6.     IF λ = n + 1 THEN
 7.        UB := min {UB, S_{n+1}};
 8.        RETURN
 9.     ELSE
10.        Let E_{λ+1} := E_λ and add to E_{λ+1} all successors
           i ∉ E_λ of j for which all predecessors are in F_λ;
11.        Calculate a lower bound LB(S) for all extensions
           of the current partial schedule S;
12.        IF LB(S) < UB THEN
13.           B&B (λ + 1, F_λ, E_{λ+1}, S, s_λ);
14.        Unschedule j and set F_λ := F_λ \ {j};
15.     ENDIF
16.  ENDWHILE
```

Figure 3.49: A branch-and-bound algorithm based on precedence trees

In this algorithm the recursive procedure B&B $(\lambda, F_\lambda, E_\lambda, S, s_{\lambda-1})$ is called, where λ denotes the current stage, and F_λ, E_λ are the sets of scheduled and eligible activities, respectively. S is the starting time vector of the current partial schedule which is successively filled by setting one component S_j to some time period t in Step 4 and by removing the S_j-value again in Step 14 during backtracking. Initially, we set $S_0 := 0$. Finally, $s_{\lambda-1}$ denotes the starting time of the previously scheduled activity in stage $\lambda - 1$.

In Step 11 additionally a lower bound $LB(S)$ for the makespan of all extensions of the current partial schedule S is calculated. If this value is not smaller than the best makespan UB found so far, the corresponding branch in the tree does not have to be inspected and backtracking can be performed. Such a lower bound

may for example be obtained by adding for the currently scheduled activity j its processing time p_j and its tail q_j to its starting time S_j, i.e. $LB(S) = S_j + p_j + q_j$. Better lower bounds may additionally take into account unscheduled activities. Since each unscheduled activity $i \notin F_\lambda$ cannot start before time S_j, it must satisfy $S_i \geq S_j$, i.e. a simple lower bound is $LB(S) = \max_{i \notin F_\lambda} \{S_j + p_i + q_i\}$. More complicated lower bounds may be obtained by calculating lower bounds for the whole set of unscheduled activities (e.g. LB_S).

As observed in Example 3.16 in Section 3.4.2, the list scheduling procedure may generate the same earliest start schedule for different lists (cf. also the precedence tree in Figure 3.47 where only six different schedules are generated in the 12 leafs of the tree). Thus, in order to reduce the size of the enumeration tree, it is desirable to eliminate sequences leading to the same earliest start schedule. The following dominance rule ensures that all generated schedules are different.

Swapping rule: If two activities i, j are started at the same time, they can be swapped in the sequence without affecting the resulting schedule. Thus, all sequences can be eliminated in which an activity i starts at the same time as its predecessor j in the sequence and i has a smaller number (i.e. $i < j$).

If the swapping rule is applied in the precedence tree algorithm for Example 3.19, the sequences $(2, 1, ., .), (3, 1, ., .), (3, 2, ., .)$ and $(2, 4, 3, 1)$ are eliminated. Then, all generated schedules in the six remaining leafs are different.

In Theorem 3.3 in Section 3.4.1 we proved that for a regular objective function always an optimal active schedule exists. Thus, additionally the following dominance criterion can be applied:

Local left shift rule: If an activity which is scheduled in the current stage of the branch-and-bound algorithm can be locally shifted to the left, the corresponding partial schedule does not have to be completed.

This rule may be extended to the **global left shift rule**, where additionally global left shifts of the current activity are considered. While using the first rule the search space is reduced to the set of semi-active schedules, with the second rule only active schedules are enumerated.

If the global left shift rule is applied in the precedence tree algorithm for Example 3.19, the schedules S_2 and S_6 are eliminated (since activity 3 can be globally shifted to the left). On the other hand, the local left shift rule is not applicable in this example.

If dominance rules are integrated into the algorithm, additionally the lower bound values may be improved. For example, if the swapping rule is applied, we know that each unscheduled activity $i \notin F_\lambda$ with a smaller number than the currently scheduled activity j cannot start before time $S_j + 1$, i.e. we may set

$$LB(S) = \max_{\{i \notin F_\lambda, i<j\}} \{S_j + 1 + p_i + q_i\}.$$

Since all additional dominance criteria need more computation time, in general a tradeoff between their effort and their effectiveness has to be found.

3.5.2 An algorithm based on extension alternatives

In this subsection we present a branch-and-bound algorithm which is based on a different idea. Contrary to the activity-oriented precedence tree algorithm, in this algorithm time points are enumerated at which activities may be started (like in the parallel schedule generation scheme, cf. Section 3.4.2). The nodes of the enumeration tree correspond to partial schedules which are precedence- and resource-feasible. In each branching step a partial schedule is extended by enumerating different subsets of unscheduled activities which can be scheduled at the current decision point.

The tree has at most $n+1$ levels and the leafs correspond to complete schedules. With each node of the enumeration tree in stage $1 \leq \lambda \leq n+1$ a decision point t_λ and three disjoint activity sets are associated: the set of finished activities F_λ which do not complete later than time t_λ, the set A_λ of all active activities (i.e. scheduled activities which complete after time t_λ) and the set E_λ of all eligible activities (i.e. all unscheduled activities for which all predecessors are completed up to time t_λ).

It is easy to see that due to the constant resource availabilities only completion times of activities have to be considered as decision points for starting unscheduled activities. Thus, in each stage λ of the branch-and-bound procedure the current decision point t_λ is calculated as the minimal completion time of all active activities in $A_{\lambda-1}$. Then the set F_λ of completed activities for the new decision point t_λ contains all activities from $F_{\lambda-1}$ and all activities in $A_{\lambda-1}$ which complete at time t_λ. Furthermore, the set E_λ of eligible activities contains all unscheduled activities $j \notin F_\lambda \cup A_{\lambda-1}$ for which all predecessors i are completed up to time t_λ, i.e. which are contained in F_λ. Finally, the new set A_λ of active activities is given by $A_{\lambda-1} \setminus F_\lambda$.

In each branching step the current partial schedule is extended by scheduling a subset of eligible activities at the corresponding decision point t_λ without violating the resource constraints. More precisely, a so-called **extension alternative** $\gamma_\lambda \subseteq E_\lambda$ is chosen, which is a resource-feasible subset of the eligible activities in E_λ with $\sum_{i \in A_\lambda \cup \gamma_\lambda} r_{ik} \leq R_k$ for all resources k, i.e. all activities from the set γ_λ can be processed simultaneously with the currently active activities in A_λ. After choosing such an extension alternative γ_λ, all its activities are added to A_λ.

In order to ensure that the algorithm terminates, we must also consider the empty extension alternative $\gamma_\lambda = \emptyset$, but impose $\gamma_\lambda \neq \emptyset$ when no activities are active at the current decision point t_λ (i.e. when $A_\lambda = \emptyset$ holds).

An extension alternative is called **maximal** if it is not contained in another feasible extension alternative. Unfortunately, in order to find an optimal schedule it is not sufficient to consider only maximal extension alternatives. This can be seen from Example 3.15. If only maximal extension alternatives are considered, in the first iteration the extension alternative $\{1, 4\}$ has to be used. All extensions of the corresponding partial schedule provide a non-delay schedule (cf. Figure 3.34(d)), which is not optimal.

However, the following weaker form of dominance may be applied. An extension alternative γ_λ is called **dominated** if an unscheduled eligible activity $i \notin \gamma_\lambda$ exists which can be scheduled simultaneously with the activities in $A_\lambda \cup \gamma_\lambda$ and which does not finish later than the first completed activity in $A_\lambda \cup \gamma_\lambda$ (which defines the next decision point $t_{\lambda+1}$). In this situation, $\gamma_\lambda \cup \{i\}$ dominates γ_λ since scheduling i together with γ_λ does not worsen the situation. For this reason (which may also be seen as a special application of the left shift rule) dominated sets may be excluded from the search process.

```
Algorithm Branch-and-Bound based on Extension Alternatives
 1.  Calculate an initial upper bound UB for the instance;
 2.  S_0 := 0; F_0 := ∅; A_0 := {0}; λ := 1;
 3.  B&B (λ, F_0, A_0, S);

Procedure B&B (λ, F_{λ-1}, A_{λ-1}, S)
 1.  t_λ := min_{i∈A_{λ-1}} {S_i + p_i};
 2.  F_λ := F_{λ-1} ∪ {i ∈ A_{λ-1} | S_i + p_i = t_λ};
 3.  Calculate the set E_λ as the set of all activities
     j ∉ F_λ ∪ A_{λ-1} for which all predecessors are in F_λ;
 4.  A_λ := A_{λ-1} \ F_λ;
 5.  Determine the set Γ_λ of all non-dominated extension
     alternatives for the sets A_λ, E_λ;
 6.  WHILE Γ_λ ≠ ∅ DO
 7.     Choose an extension alternative γ_λ ∈ Γ_λ;
 8.     Γ_λ := Γ_λ \ {γ_λ};
 9.     Schedule all activities j ∈ γ_λ at time S_j := t_λ;
10.     A_λ := A_λ ∪ γ_λ;
11.     IF n + 1 ∈ γ_λ THEN
12.        UB := min {UB, S_{n+1}};
13.        RETURN
14.     ELSE
15.        Calculate a lower bound LB(S) for all extensions
           of the current partial schedule S;
16.        IF LB(S) < UB THEN
17.           B&B (λ + 1, F_λ, A_λ, S);
18.        Unschedule all j ∈ γ_λ and set A_λ := A_λ \ γ_λ;
19.     ENDIF
20.  ENDWHILE
```

Figure 3.50: A branch-and-bound algorithm based on extension alternatives

Using the current sets A_λ and E_λ the set Γ_λ of all non-dominated extension alternatives can easily be calculated for the current partial schedule. After choosing an extension alternative $\gamma_\lambda \in \Gamma_\lambda$ and scheduling all activities from this

set γ_λ, the next branching stage $\lambda + 1$ is considered.

If all activities are scheduled, the makespan of the corresponding schedule is compared to the best upper bound value found so far. Afterwards, backtracking to the previous stage occurs where the next unconsidered extension alternative is chosen. If all non-dominated extension alternatives have been considered as branching candidates in a stage, a further backtracking step to the previous stage is performed.

The preceding discussions are summarized in the algorithm shown in Figure 3.50. In this algorithm the recursive procedure `B&B` $(\lambda, F_{\lambda-1}, A_{\lambda-1}, S)$ is called, where λ denotes the current stage, $F_{\lambda-1}$ and $A_{\lambda-1}$ are the sets of finished and active activities from the previous stage $\lambda - 1$ and S is the starting time vector of the current partial schedule. In order to have a unique description of the recursive procedure initially again a dummy starting activity 0 is started at time $S_0 := 0$.

In Step 15 a lower bound $LB(S)$ for the makespan of all extensions of the current partial schedule S can be calculated similarly as in Step 11 of the precedence tree algorithm, for example as

$$LB(S) = \max_{j \in \gamma_\lambda} \{S_j + p_j + q_j\} \text{ or } LB(S) = \max_{i \notin (F_\lambda \cup A_\lambda)} \{\min_{j \in A_\lambda} (S_j + p_j) + p_i + q_i\}.$$

Furthermore, as in the precedence tree algorithm dominance rules like the local or global left shift rule can be integrated.

Example 3.20: Consider again the instance from Example 3.19 shown in Figure 3.46. The resulting enumeration tree for the algorithm based on extension alternatives can be found in Figure 3.51. For each node the current decision point t_λ, the set of active activities A_λ and the set Γ_λ of non-dominated extension alternatives are given.

In the first braching step for $t_1 = 0$ and $A_1 = \{0\}$ only the three extension alternatives $\{1,2\}, \{1,3\}$ and $\{2,3\}$ have to be considered since the extension alternative $\{1\}$ is dominated by the alternative $\{1,2\}$ and the sets $\{2\}, \{3\}$ are both dominated by $\{2,3\}$.

For example, in the third stage for the left branch $\{1,2\} - \{3\}$ the empty extension alternative has to be used since at the current decision point $t_3 = C_3 = 4$ we have $A_3 = \{1\} \neq \emptyset$ and no activity can be processed in parallel with the active activity 1 at time 4. On the other hand, in the second stage for the branch $\{2,3\}$ at the decision point $t_2 = C_2 = 2$ the empty extension alternative may not be used since the set of active activities A_2 is empty. □

Another approach which extends partial schedules by subsets of activities is the so-called **block extension procedure** using the concept of feasible subsets X introduced in Section 3.3.2. It is based on the observation that an optimal schedule defines time points $t_0 = 0 < t_1 < \ldots < t_\tau$ and corresponding feasible subsets $X_1, \ldots, X_\tau \subset \{1, \ldots, n\}$ such that

- each time point t_j with $j \in \{1, \ldots, \tau\}$ is the completion time of an activity,

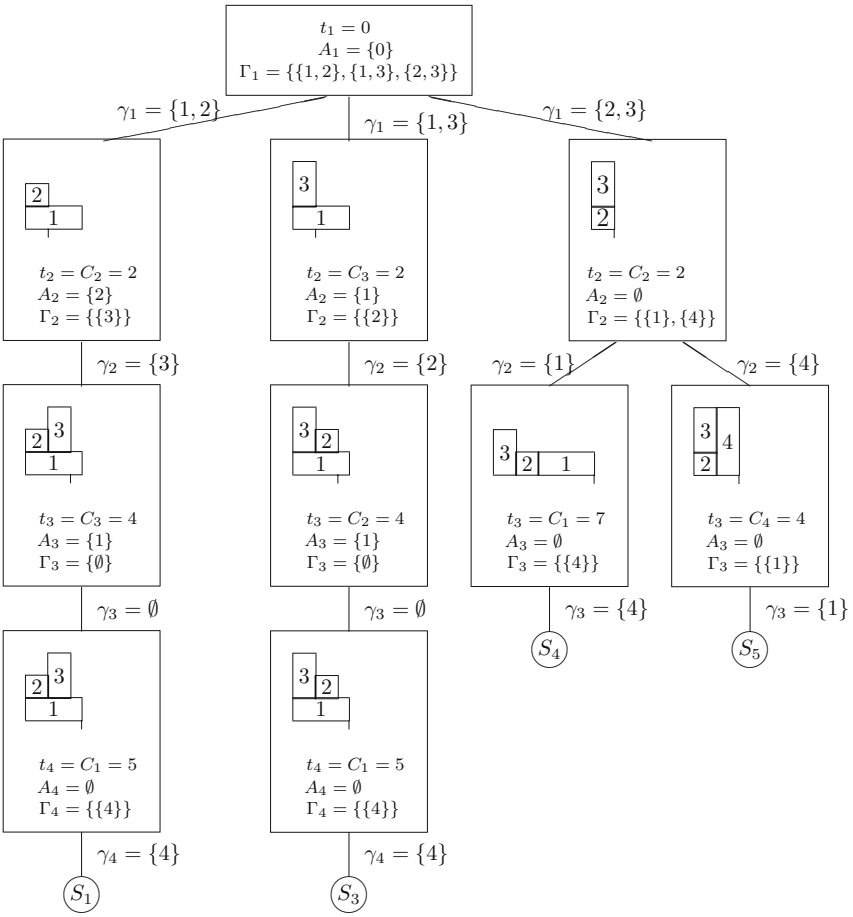


Figure 3.51: Search tree for the algorithm based on extension alternatives

- all activities in X_j are processed simultaneously in the whole interval $[t_{j-1}, t_j[$ for $j = 1, \ldots, \tau$,
- if an activity $i \in X_j$ is not finished in the interval $[t_{j-1}, t_j[$, it is continued in the interval $[t_j, t_{j+1}[$,
- all predecessors of an activity starting at time t_j are finished before time t_j.

An interval $[t_{j-1}, t_j[$ with a corresponding feasible subset X_j of activities satisfying the last two properties is called a feasible block. Obviously, a schedule can be built by successively adding a feasible block to the end of a partial schedule. Thus, in a branch-and-bound algorithm all possible block extensions of a given partial schedule may be enumerated.

3.5.3 An algorithm based on delaying alternatives

In this subsection we present a branch-and-bound algorithm which is based on scheduling activities until a resource conflict occurs. As in the algorithm based on extension alternatives time points are enumerated at which activities may be started. The nodes of the enumeration tree correspond to partial schedules which are precedence- and resource-feasible and are extended by starting all eligible activities. If this causes a resource conflict, in a branching step this conflict is resolved by delaying different subsets of activities.

As in the algorithm based on extension alternatives, the enumeration tree has at most $n+1$ levels and the leafs correspond to complete schedules. With each node of the enumeration tree in stage $1 \leq \lambda \leq n+1$ a decision point t_λ and three disjoint activity sets are associated: the set of finished activities F_λ which do not complete later than time t_λ, the set A_λ of all active activities (i.e. scheduled activities which complete after time t_λ) and the set E_λ of all eligible activities (i.e. all unscheduled activities for which all predecessors are completed up to time t_λ).

As in the algorithm based on extension alternatives, in each stage λ of the branch-and-bound procedure the current decision point t_λ is calculated as the minimal completion time of all active activities in $A_{\lambda-1}$ (again only completion times of activities have to be considered as decision points for starting unscheduled activities). Then the set F_λ of finished activities for the new decision point t_λ contains all activities from $F_{\lambda-1}$ and all activities in $A_{\lambda-1}$ which complete at time t_λ. Furthermore, the set E_λ of eligible activities contains all unscheduled activities $j \notin F_\lambda \cup A_{\lambda-1}$ for which all predecessors i are completed up to time t_λ, i.e. which are contained in F_λ.

Then, contrary to the algorithm based on extension alternatives, all eligible unscheduled activities are temporarily started, i.e. afterwards the new set of active activities is given by $A_\lambda = (A_{\lambda-1} \setminus F_\lambda) \cup E_\lambda$. If this set causes a resource conflict (i.e. $\sum_{i \in A_\lambda} r_{ik} > R_k$ for some resource k), some activities are delayed, i.e. they have to be started later.

In each branching step a resource conflict in the current partial schedule is resolved by delaying a subset of the active activities A_λ at the corresponding decision point t_λ such that the resources are sufficient for the non-delayed activities. More precisely, a so-called **delaying alternative** $\delta_\lambda \subset A_\lambda$ is chosen, such that the remaining set $A_\lambda \setminus \delta_\lambda$ satisfies $\sum_{i \in A_\lambda \setminus \delta_\lambda} r_{ik} \leq R_k$ for all resources k. Note that in this algorithm only activities from the set F_λ are permanently scheduled in the current partial schedule, activities from the set A_λ may be moved to the right in subsequent iterations.

A delaying alternative is called **minimal**, if it does not contain a proper subset which is also a delaying alternative. Contrary to the algorithm based on extension alternatives where also non-maximal extension alternatives have to be considered, in the algorithm based on delaying alternatives it is sufficient to con-

sider only minimal delaying alternatives. This is based on the fact that activities which are not delayed in the current stage of the branch-and-bound tree can still be delayed in later stages of the algorithm (in the algorithm based on extension alternatives previous decisions cannot be changed in later iterations). A proof of this property will be given after the description of the whole algorithm.

Using the current set A_λ the set Δ_λ of all minimal delaying alternatives can be calculated for the current partial schedule. After choosing a delaying alternative $\delta_\lambda \in \Delta_\lambda$ all activities from the set δ_λ are unscheduled, i.e. removed from the current partial schedule. Note that if no resource conflict occurs, the empty set is the only minimal delaying alternative. After storing the old starting time $S_j^\lambda := t_\lambda$ for all delayed activities $j \in \delta_\lambda$ (which are used in the backtracking step), the next branching stage $\lambda + 1$ is considered.

If all activities are scheduled, the makespan of the corresponding schedule is compared to the best upper bound value found so far. Afterwards, backtracking to the previous stage occurs where the next unconsidered minimal delaying alternative is chosen. If all minimal delaying alternatives have been considered as branching candidates in a stage, a further backtracking step to the previous stage is performed.

The preceding discussions are summarized in the algorithm shown in Figure 3.52. In this algorithm the recursive procedure `B&B` $(\lambda, F_{\lambda-1}, A_{\lambda-1}, \delta_{\lambda-1}, S)$ is called, where λ denotes the current stage, $F_{\lambda-1}$ and $A_{\lambda-1}$ are the sets of finished and active activities from the previous stage $\lambda - 1$, the set $\delta_{\lambda-1}$ is the chosen delaying alternative in stage $\lambda - 1$, and S is the starting time vector of the current partial schedule.

In Step 14 a lower bound $LB(\delta_\lambda)$ for the makespan of all extensions induced by the current delaying alternative δ_λ can be calculated using the tails q_j of the delayed activities $j \in \delta_\lambda$. For example, we may set

$$LB(\delta_\lambda) := \min_{i \in A_\lambda \setminus \delta_\lambda} \{S_i + p_i\} + \max_{j \in \delta_\lambda} \{p_j + q_j\},$$

since no delayed activity can be started before the first currently scheduled activity is completed.

Theorem 3.6 In the branch-and-bound algorithm based on delaying alternatives it is sufficient to consider only minimal delaying alternatives.

Proof: Assume to the contrary that an instance exists for which an optimal schedule can only be constructed by the algorithm if non-minimal delaying alternatives are used. Let $(\delta_1, \ldots, \delta_\lambda, \ldots, \delta_k)$ be a sequence of delaying alternatives where λ is the largest index such that $(\delta_1, \ldots, \delta_\lambda, \ldots, \delta_k)$ leads to an optimal schedule S^* and $\delta_1, \ldots, \delta_{\lambda-1}$ are minimal delaying alternatives. Since δ_λ is not minimal, an activity $x \in \delta_\lambda$ exists which can be eliminated such that $\delta'_\lambda := \delta_\lambda \setminus \{x\}$ is again a delaying alternative. Let S and S' be the partial schedules corresponding to the subsequences $(\delta_1, \ldots, \delta_\lambda)$ and $(\delta_1, \ldots, \delta_{\lambda-1}, \delta'_\lambda)$,

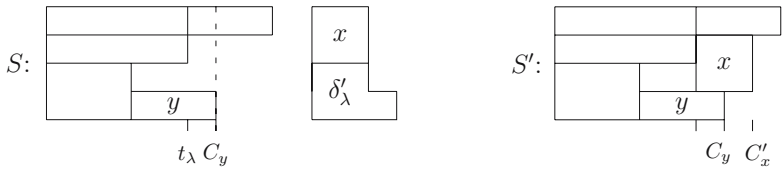
```
Algorithm Branch-and-Bound based on Delaying Alternatives
 1.  Calculate an initial upper bound UB for the instance;
 2.  S_0 := 0; F_0 := ∅; E_0 := ∅; A_0 := {0}; δ_0 := ∅; λ := 1;
 3.  B&B (λ, F_0, A_0, δ_0, S);

Procedure B&B (λ, F_{λ-1}, A_{λ-1}, δ_{λ-1}, S)
 1.  t_λ := min_{i∈A_{λ-1}} {S_i + p_i};
 2.  F_λ := F_{λ-1} ∪ {i ∈ A_{λ-1} | S_i + p_i = t_λ};
 3.  Calculate the set E_λ as the set of all activities
     j ∉ F_λ ∪ A_{λ-1} for which all predecessors are in F_λ;
 4.  Schedule all j ∈ E_λ temporarily at time S_j := t_λ;
 5.  A_λ := (A_{λ-1} \ F_λ) ∪ E_λ;
 6.  IF n + 1 ∈ E_λ THEN
 7.     UB := min {UB, S_{n+1}};
 8.     RETURN
 9.  ELSE
10.     Determine the set Δ_λ of all minimal delaying
        alternatives for the set A_λ;
11.     WHILE Δ_λ ≠ ∅ DO
12.        Choose a delaying alternative δ_λ ∈ Δ_λ;
13.        Δ_λ := Δ_λ \ {δ_λ};
14.        Calculate a lower bound LB(δ_λ) for all extensions
           of the delaying alternative δ_λ;
15.        IF LB(δ_λ) < UB THEN
16.           A_λ := A_λ \ δ_λ;
17.           FOR ALL j ∈ δ_λ DO S_j^λ := S_j;
18.           B&B (λ + 1, F_λ, A_λ, δ_λ, S);
19.           FOR ALL j ∈ δ_λ DO S_j := S_j^λ;
20.           A_λ := A_λ ∪ δ_λ;
21.        ENDIF
22.     ENDWHILE
23.  ENDIF
```

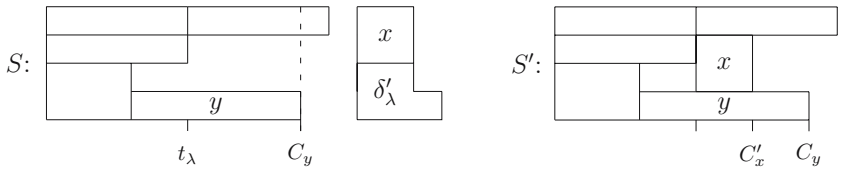
Figure 3.52: A branch-and-bound algorithm based on delaying alternatives

respectively. Denote by C_j and C'_j the completion times of the scheduled activities after delaying the activities in the sets δ_λ and δ'_λ. Furthermore, let y be an activity in $A_\lambda \setminus \delta_\lambda$ which completes first in S.

- Case 1: $C'_x \geq C_y$, i.e. when x is not delayed in stage λ, it does not complete earlier than y in S' (cf. Figure 3.53(a)). If we proceed the algorithm with the subsequence $(\delta_1, \ldots, \delta_{\lambda-1}, \delta'_\lambda)$, the next considered decision point is $t'_{\lambda+1} = C_y$ which is the same as for the subsequence $(\delta_1, \ldots, \delta_\lambda)$. Since the



(a) Case 1: $C'_x \geq C_y$



(b) Case 2: $C'_x < C_y$

Figure 3.53: The two cases in the proof of Theorem 3.6

schedule S^* can be generated by the second subsequence and in S' activity x may still be delayed at time $t'_{\lambda+1} = C_y$, it can also be generated by the first subsequence. But this contradicts our assumption that in order to construct an optimal schedule S^* in stage λ the non-minimal delaying alternative δ_λ has to be used.

- Case 2: $C'_x < C_y$ (cf. Figure 3.53(b)). In S and S' up to stage λ the same activities are scheduled, i.e. we have $C_j = C'_j$ for all activities $j \in F_\lambda$, and C'_x is smaller than the completion time of x in any extension of schedule S since in S activity x cannot be started before time $t_{\lambda+1} = C_y$. Consider the schedule $\tilde{S}$ with completion times

$$\tilde{C}_j := \begin{cases} C'_j & \text{for } j \in F_\lambda \cup \{x\} \\ C^*_j & \text{otherwise,} \end{cases}$$

which is an extension of the partial schedule S'. This schedule is feasible since in S' all activities $j \in F_\lambda \cup \{x\}$ are completed no later than time $t'_{\lambda+1} = C'_x < C_y = t_{\lambda+1}$ and all other activities do not overlap with them since they do not overlap in S^*. Furthermore, $\tilde{S}$ has the same makespan as the optimal schedule S^*.

Additionally, a schedule $\tilde{S}'$ with the same makespan as $\tilde{S}$ exists which is equal to $\tilde{S}$ or can be obtained from $\tilde{S}$ by some additional left shifts of activities due to the left shift of x. It can be shown that this optimal schedule $\tilde{S}'$ can be constructed by the subsequence $(\delta_1, \ldots, \delta_{\lambda-1}, \delta'_\lambda)$ since the next decision point $t'_{\lambda+1} = C'_x$ is smaller than the next decision point $t_{\lambda+1} = C_y$ for the subsequence $(\delta_1, \ldots, \delta_\lambda)$ and S^* can be constructed by this subsequence. This again contradicts the fact that in stage λ a non-minimal delaying alternative δ_λ has to be used in order to construct an optimal schedule. □

Example 3.21 : Consider again the instance from Example 3.19 shown in Figure 3.46. The resulting enumeration tree for the algorithm based on delaying alternatives can be found in Figure 3.54. For each node the current decision point t_λ, the set of eligible activities E_λ and the set Δ_λ of minimal delaying alternatives are given.

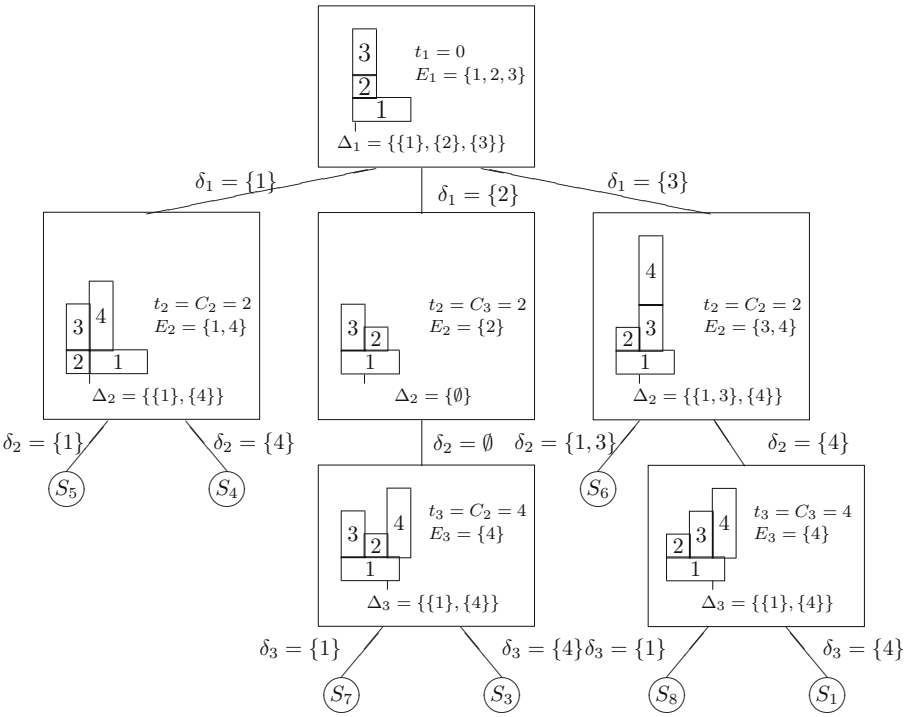


Figure 3.54: Search tree for the algorithm based on delaying alternatives

Compared with the enumeration trees of the previous two algorithms two new schedules are generated (cf. Figure 3.55).

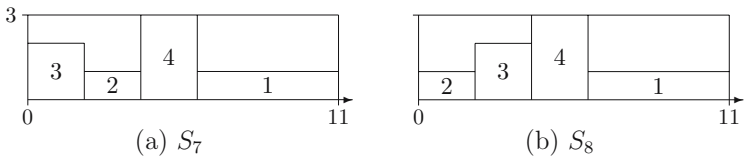


Figure 3.55: Additional schedules obtained in Example 3.21 □

A main difference between the the two previous algorithms (based on precedence trees and extension alternatives) and the algorithm based on delaying alternatives is the treatment of scheduling decisions from previous stages. While in the first two algorithms scheduling decisions from earlier stages are not withdrawn

in later stages, in the third algorithm activities which have been scheduled in previous stages may be delayed in later stages.

For this reason, also the left shift rules behave differently. Usually, in connection with delaying alternatives the local left shift rule is formulated as follows: If at the current decision point t_λ an activity i is scheduled which has been delayed in the previous stage (i.e. $i \in \delta_{\lambda-1} \setminus \delta_\lambda$) and delaying the activities from the current delaying alternative δ_λ allows that i can be locally shifted to the left, then the current partial schedule can be eliminated. However, delaying an activity in a later stage can make a local left shift applicable which is not feasible at the current stage. Thus, this version of the local left shift rule does not guarantee that only semi-active schedules are generated.

As an example, consider the schedule S_8 from the example above which is not semi-active. It is also generated if the local left shift rule is used. When in the third stage the delaying alternative $\delta_3 = \{1\}$ is chosen, only the activity $4 \in \delta_2 = \{4\}$ from the previous delaying alternative is tested for a local left shift. However, due to the delay of activity 1, now activity 3 can be shifted to the left, which is not detected.

This example shows that in order to guarantee that only semi-active schedules are generated, additionally some other activities have to be tested for possible left shifts. More precisely, all scheduled activities which do not start before the first currently delayed activity $j \in \delta_\lambda$ have to be tested:

Extended local left shift rule: Let $t_{\min} := \min_{j \in \delta_\lambda}\{S_j\}$ be the minimal starting time of all activities belonging to the current delaying alternative δ_λ in stage λ. If a scheduled activity $i \notin \delta_\lambda$ with $S_i \geq t_{\min}$ exists which can be locally shifted to the left after delaying δ_λ, the current partial schedule can be eliminated.

For the above example, with this modification the possible left shift of activity 3 in the schedule S_8 is detected since $t_{\min} = S_1 = 0 < S_3 = 2$.

Additionally to the left shift rule the following dominance rule was proposed:

Cutset rule: For each decision point t_λ let the cutset $C(t_\lambda)$ be defined as the set of all unscheduled activities for which all predecessors are already scheduled (i.e. belong to the set $F_{\lambda-1} \cup A_{\lambda-1}$). Furthermore, for each scheduled activity $j \in F_{\lambda-1} \cup A_{\lambda-1}$ denote by C_j^λ its completion time in the partial schedule belonging to t_λ. Assume that a cutset $C(t_\lambda)$ equals a cutset $C(t_\nu)$ which has previously been saved in a node of the search tree. If $t_\nu \leq t_\lambda$ and each activity j which is active at time t_ν in the partial schedule belonging to t_ν satisfies $C_j^\nu \leq \max\{t_\lambda, C_j^\lambda\}$, then the partial schedule belonging to t_λ is dominated.

In order to prove the correctness of this dominance rule, it can be shown that an optimal extension of the partial schedule C^λ cannot be better than an optimal extension of the partial schedule C^ν. Using this rule, the computation times can often be reduced since large parts of the enumeration tree can be pruned off. On the other hand, this rule needs a large amount of storage since the completion times in all considered partial schedules have to be stored.

3.5.4 An algorithm based on schedule schemes

In this subsection we present a branch-and-bound algorithm which is based on the basic relations introduced in Section 3.2.1. The nodes of the enumeration tree represent sets of feasible schedules which are defined by conjunctions, disjunctions, and parallelity relations. In each branching step the set of schedules belonging to a node is split into two subsets. This is accomplished by a binary branching scheme where for two activities i, j in one branch the parallelity relation $i \parallel j$ and in the other branch the disjunction $i - j$ is added.

First we will introduce schedule schemes, which are used to represent the nodes of the enumeration tree mathematically. Let $C, D, N \subseteq V \times V$ be disjoint relations where C is a set of conjunctions, D is a set of disjunctions, and N is a set of parallelity relations. The relations D and N are symmetric, i.e. with $i - j \in D$ $(i \parallel j \in N)$ also $j - i \in D$ $(j \parallel i \in N)$ holds. On the other hand, C is antisymmetric, i.e. if $i \rightarrow j \in C$, then $j \rightarrow i \notin C$. If $i - j \in D$, then it is not specified whether $i \rightarrow j$ or $j \rightarrow i$ holds. Based on C, D, N we define the set of remaining relations as

$$F = \{(i,j) \mid i,j \in V;\ i \neq j;\ i \rightarrow j \notin C,\ j \rightarrow i \notin C,\ i - j \notin D,\ i \parallel j \notin N\}.$$

The relations in F are called **flexibility relations** and are denoted by $i \sim j$. Note, that F is symmetric. The tuple (C, D, N, F) with the property that for all $i, j \in V$ with $i \neq j$ exactly one of the relations $i \rightarrow j \in C$, $j \rightarrow i \in C$, $i - j \in D$, $i \parallel j \in N$ or $i \sim j \in F$ holds, is called a **schedule scheme**.

A schedule scheme (C, D, N, F) defines a (possibly empty) set $\mathcal{S}(C, D, N, F)$ of schedules. More precisely, $\mathcal{S}(C, D, N, F)$ is the set of all schedules with the following properties:

- if $i \rightarrow j \in C$, then i is finished when j starts,
- if $i \parallel j \in N$, then i and j are processed in parallel for at least one time unit, and
- if $i - j \in D$, then i and j are not processed in parallel.

$\mathcal{S}_f(C, D, N, F)$ denotes the corresponding set of feasible schedules, i.e. all schedules in $\mathcal{S}(C, D, N, F)$ which additionally satisfy the resource constraints. If for an RCPSP instance the sets C_0, D_0, N_0 are defined as in Section 3.2.1 and F_0 are the corresponding remaining flexibility relations, then $\mathcal{S}_f(C_0, D_0, N_0, F_0)$ is the set of all feasible schedules for this instance.

Starting with the initial schedule scheme (C_0, D_0, N_0, F_0) as root of the enumeration tree, the branch-and-bound algorithm is based on a binary branching scheme where in each step for one flexibility relation $i \sim j \in F$ the two branches $i - j \in D$ and $i \parallel j \in N$ are created. This branching process is repeated until the set F becomes empty. We will show that for such a schedule scheme $(C, D, N, \emptyset)$

- we can either detect that $\mathcal{S}_f(C, D, N, \emptyset) = \emptyset$ (i.e. no feasible schedule corresponding to this scheme exists), or
- we can calculate a schedule which dominates all feasible schedules in the set $\mathcal{S}_f(C, D, N, \emptyset)$.

Thus, a schedule scheme $(C, D, N, \emptyset)$ with no flexibility relations can be treated as leaf in the enumeration tree.

Consider a schedule scheme $(C, D, N, \emptyset)$ with no flexibility relations. Then the schedule scheme $(C', \emptyset, N, \emptyset)$ is called a **transitive orientation** of $(C, D, N, \emptyset)$ if

- $C \subseteq C'$ and $(C', \emptyset, N, \emptyset)$ is derived from $(C, D, N, \emptyset)$ by changing each disjunction $i - j \in D$ into either $i \rightarrow j \in C'$ or $j \rightarrow i \in C'$,
- C' is transitive, i.e. $i \rightarrow j \in C'$ and $j \rightarrow k \in C'$ imply $i \rightarrow k \in C'$.

It is easy to see that each feasible schedule can be represented by a transitive orientation $(C', \emptyset, N, \emptyset)$ of some schedule scheme $(C, D, N, \emptyset)$ (if for a disjunction $i - j \in D$ activity i is started before j in the schedule, set $i \rightarrow j \in C'$). In the following we will show how a dominating schedule can be calculated for a transitive orientation $(C', \emptyset, N, \emptyset)$.

Note that if $(C', \emptyset, N, \emptyset)$ is a transitive orientation, then the directed graph (V, C') is acyclic because C' is antisymmetric. Let $(C', \emptyset, N, \emptyset)$ be a transitive orientation of a schedule scheme. Ignoring the parallelity relations we calculate a corresponding (not necessarily feasible) schedule as follows. Consider the acyclic directed graph (V, C'). For each activity $i \in V$ we calculate the length r_i of a longest path from 0 to i and start i at time r_i. We denote this **earliest start schedule** (which only depends on C'), by $S_{ES}(C')$. Note that $S_{ES}(C')$ does not have to belong to the set $\mathcal{S}_f(C', \emptyset, N, \emptyset)$ since the parallelity relations are not explicitly taken into account and resource constraints may be violated. However, since obviously $C_{\max}(S_{ES}(C')) \leq C_{\max}(S)$ for all schedules $S \in \mathcal{S}_f(C', \emptyset, N, \emptyset)$ holds and the following theorem is valid, this causes no problem.

Theorem 3.7 Let $(C', \emptyset, N, \emptyset)$ be an arbitrary transitive orientation of a schedule scheme $(C, D, N, \emptyset)$ and let $S_{ES}(C')$ be the corresponding earliest start schedule. If $S_{ES}(C')$ is feasible, then $S_{ES}(C')$ dominates all schedules in the set $\mathcal{S}_f(C, D, N, \emptyset)$. Otherwise, the set $\mathcal{S}_f(C, D, N, \emptyset)$ of feasible schedules is empty.

Proof: Assume that $S_{ES}(C')$ is infeasible, i.e. there is a time period t such that activities of a set H are processed in parallel during t and for some resource type k we have $\sum_{i \in H} r_{ik} > R_k$. If there exists a feasible schedule $S \in \mathcal{S}_f(C, D, N, \emptyset)$, then for at least two activities $i, j \in H$ one of the relations $i \rightarrow j \in C$ or $j \rightarrow i \in C$ or $i - j \in D$, i.e. $i \parallel j \notin N$ holds. Otherwise, if we have $i \parallel j \in N$ for all $i, j \in H$ with $i \neq j$, this implies that also in S all activities are processed

in parallel during some time period t. This can be seen as follows. Let $H^0 \subseteq H$ be a maximal set of activities which are processed in parallel in S during a time interval I and assume that $k \in H \setminus H^0$, i.e. for the processing interval $I_k := [S_k, C_k[$ we have $I \cap I_k = \emptyset$. Assume w.l.o.g. that I_k is to the right of I. Then an activity $l \in H^0$ exists which does not complete later than k starts in S. Thus, $I_k \cap I_l = \emptyset$, which is a contradiction to $k \parallel l \in N$. However, if all activities in H are processed in parallel, then S cannot be a feasible schedule. Hence, for at least two activities $i, j \in H$ we have $i \parallel j \notin N$ implying $i \rightarrow j \in C'$ or $j \rightarrow i \in C'$, which contradicts the definition of H. Thus, if $S_{ES}(C')$ is infeasible, the set $\mathcal{S}_f(C, D, N, \emptyset)$ of feasible schedules must be empty.

Now consider an arbitrary feasible schedule $S \in \mathcal{S}_f(C, D, N, \emptyset)$. This schedule S defines a transitive orientation $(C'', \emptyset, N, \emptyset)$ of $(C, D, N, \emptyset)$ and the corresponding earliest start schedule $S_{ES}(C'')$ does not have a larger makespan than S.

Due to graph theoretical results in connection with so-called comparability graphs an earliest start schedule $S_{ES}(C')$ associated with a transitive orientation $(C', \emptyset, N, \emptyset)$ of a schedule scheme $(C, D, N, \emptyset)$ has always the same $C_{\max}$-value, regardless which transitive orientation is chosen. Thus, $S_{ES}(C')$ and $S_{ES}(C'')$ have the same $C_{\max}$-value, i.e. $S_{ES}(C')$ dominates S as well. □

Thus, due to Theorem 3.7 we may proceed as follows. Whenever the set of flexibility relations becomes empty, we first check whether a transitive orientation of the corresponding schedule scheme $(C, D, N, \emptyset)$ exists (not every schedule scheme $(C, D, N, \emptyset)$ can be transitively oriented). This can be done by a polynomial-time algorithm (cf. the reference notes). If no transitive orientation exists, the schedule scheme does not represent a feasible schedule and backtracking may be performed. Otherwise, for the transitive orientation the corresponding earliest start schedule is calculated. If it is feasible, its makespan is compared to the best upper bound value found so far. Afterwards, backtracking occurs.

The preceding discussions are summarized in the algorithm shown in Figure 3.56. In this algorithm the recursive procedure `B&B` (C, D, N, F) is called. This procedure applies at first some constraint propagation techniques to the given schedule scheme in order to reduce the search space. It introduces additional conjunctions, disjunctions and parallelity relations as described in Section 3.2.4 and returns the new schedule scheme.

In Step 13 additionally a lower bound $LB(C, D, N, F)$ for all schedules in the set $\mathcal{S}_f(C, D, N, F)$ is calculated. Such a lower bound may for example be obtained by considering the distance matrix based on the schedule scheme (cf. Section 3.2.2) and taking the distance $d_{0,n+1}$ as lower bound value after constraint propagation has been applied. If this value is not smaller than the best makespan UB found so far, the corresponding branch in the tree does not have to be inspected and backtracking can be performed.

In Step 15 a flexibility relation $i \sim j \in F$ has to be chosen creating the two branches $i - j \in D$ and $i \parallel j \in N$. This choice may be guided by a priority rule. For example, weights w_{i-j} and $w_{i\parallel j}$ may be calculated which are approximations of lower bounds for the schedules in the two branches. Then a relation $i \sim j \in F$

```
Algorithm Branch-and-Bound based on Schedule Schemes
 1.  Calculate an initial upper bound UB for the instance;
 2.  B&B (C_0, D_0, N_0, F_0)

Procedure B&B (C, D, N, F)
 1.  ConstraintPropagation (C, D, N, F);
 2.  IF F = ∅ THEN
 3.     Check if a transitive orientation exists for
        (C, D, N, ∅);
 4.     IF no transitive orientation exists THEN
 5.        RETURN
 6.     ELSE
 7.        Calculate a transitive orientation (C', ∅, N, ∅) of
           (C, D, N, ∅), the earliest start schedule S_ES(C')
           and its makespan C_max(S_ES(C'));
 8.        IF S_ES(C') is feasible THEN
 9.           UB := min{UB, C_max(S_ES(C'))};
10.        ELSE RETURN
11.     ENDIF
12.  ELSE
13.     Calculate a lower bound LB(C, D, N, F);
14.     IF LB(C, D, N, F) < UB THEN
15.        Choose i ~ j ∈ F;
16.        B&B (C, D ∪ {i - j}, N, F \ {i ~ j});
17.        B&B (C, D, N ∪ {i ‖ j}, F \ {i ~ j});
18.     ENDIF
19.  ENDIF
```

Figure 3.56: A branch-and-bound algorithm based on schedule schemes

is chosen such that the sum $w_{i-j} + w_{i\|j}$ is maximal (in order to prune off the branch with a higher probability).

Example 3.22: Consider again the instance from Example 3.19 shown in Figure 3.46. Initially we have

$$C_0 = \{2 \to 4\},\ D_0 = \{1-4, 3-4\},\ N_0 = \emptyset,\ F_0 = \{1 \sim 2, 1 \sim 3, 2 \sim 3\}.$$

A possible enumeration tree for the algorithm based on schedule schemes can be found in Figure 3.57. In the first stage the relation $1 \sim 2$ is chosen, in the second $1 \sim 3$ and in the third $2 \sim 3$. For all resulting schedule schemes $(C, D, N, \emptyset)$ in the 8 leafs transitive orientations exist. In Figure 3.58 possible orientations for the schedule schemes in the 8 leafs are shown together with the corresponding earliest-start schedules.

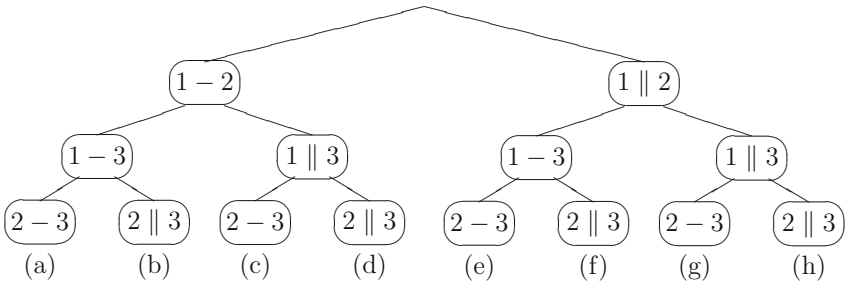


Figure 3.57: Search tree for the algorithm based on schedule schemes

Note that in (d) and (f) the earliest start schedules do not belong to the set of feasible schedules represented by the corresponding schedule schemes $(C, D, N, \emptyset)$ since in (d) the parallelity relation $2 \parallel 3$ and in (f) the parallelity relation $1 \parallel 2$ is violated. Furthermore, for the orientation in (h) the earliest start schedule is not feasible since processing the activities $1, 2, 3$ in parallel violates the resource constraints. Thus, in this case the set of feasible schedules belonging to the corresponding schedule scheme is empty. □

3.5.5 Algorithms for the multi-mode case

In order to solve the multi-mode RCPSP by a branch-and-bound algorithm additionally modes have to be enumerated. The algorithms from Subsections 3.5.1 to 3.5.3 can be extended to the multi-mode case as follows.

In the precedence tree algorithm in each stage λ besides an eligible activity $j \in E_\lambda$ also a mode $m \in \mathcal{M}_j$ for the activity has to be chosen. Each combination of an eligible activity and a mode for it corresponds to a successor of the current node in the enumeration tree.

In the algorithm based on extension alternatives additionally modes have to be determined before the activities in an extension alternative can be started. Thus, in this step the modes are enumerated by so-called mode alternatives which consist of all feasible mode combinations for the eligible activities. Enumerating all possible mode alternatives for these activities leads to a refined branching step where each combination of a mode alternative and a corresponding feasible extension alternative leads to a successor of the current search tree node.

The same idea can also be used in the algorithm based on delaying alternatives. Here, in stage λ modes have to be assigned to all eligible activities in $E_\lambda \setminus E_{\lambda-1}$ before they are temporarily started. Enumerating all possible mode alternatives for these activities leads to a refined branching step where each combination of a mode alternative and a corresponding feasible delaying alternative corresponds to a successor of the current search tree node.

For all these algorithms also dominance rules have been generalized. For example, in all algorithms which enumerate partial schedules the following rule

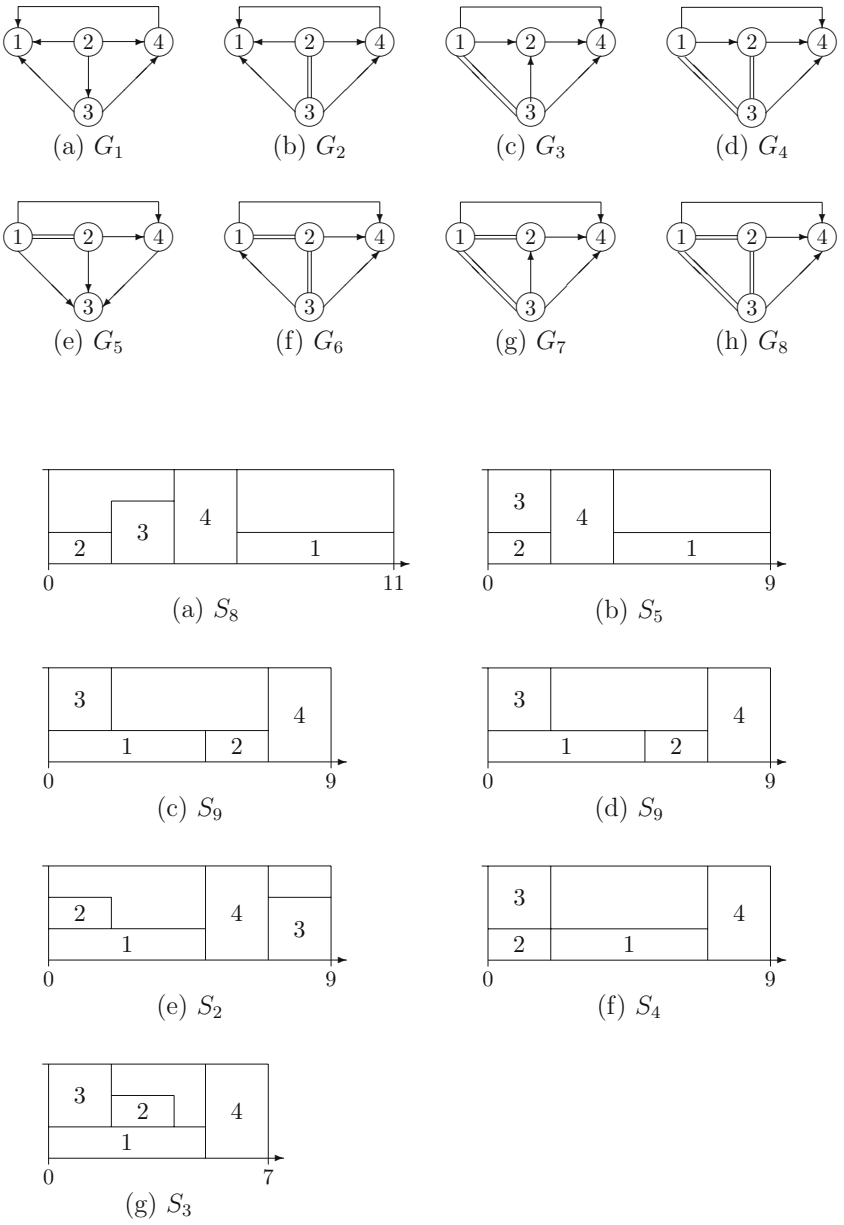


Figure 3.58: Enumerated graphs and schedules for Example 3.22

may be applied: If an eligible unscheduled activity cannot be feasibly scheduled in any mode in the current partial schedule, then the current schedule can be eliminated.

Furthermore, in addition to local left shifts where the mode of the shifted activity remains the same, so-called **multi-mode left shifts** may be applied. For a given schedule a multi-mode left shift of an activity i reduces the completion time of i without changing the modes or completion times of the other activities (but allowing that the mode of activity i may be changed).

3.5.6 Reference notes

The precedence tree algorithm was proposed by Patterson et al. [119], [120] and improved by additional dominance rules in Sprecher [132]. Furthermore, in Sprecher and Drexl [133] it was generalized to the multi-mode case.

The branch-and-bound algorithm based on extension alternatives is due to Stinson et al. [137]. It was generalized to the multi-mode case by Hartmann and Drexl [68]. A branch-and-bound algorithm based on block extensions was proposed by Mingozzi et al. [105].

The concept of delaying alternatives was first used by Christofides et al. [33] and enhanced by Demeulemeester and Herroelen [40], [41]. In Sprecher and Drexl [134] it was shown that the local left shift rule in connection with the branch-and-bound algorithm based on delaying alternatives does not guarantee that only semi-active schedules are enumerated. The branch-and-bound algorithm based on delaying alternatives was generalized to the multi-mode case by Sprecher et al. [135], to the RCPSP with generalized precedence relations by De Reyck and Herroelen [43] and to the multi-mode RCPSP with generalized precedence relations by De Reyck and Herroelen [44]. Other time-oriented branch-and-bound algorithms for the RCPSP with generalized precedence constraints can be found in Dorndorf et al. [50] and Fest et al. [53].

The branch-and-bound algorithm based on schedule schemes was proposed by Brucker et al. [22] using concepts from Bartusch et al. [9] and Krämer [94]. The mentioned graph theoretical results in connection with comparability graphs can be found in Golumbic [62] and Möhring [106]. Checking whether a transitive orientation exists, can be done by a polynomial-time algorithm of Korte and Möhring [92].

A comparison of exact algorithms for the multi-mode RCPSP was performed by Hartmann and Drexl [68]. Besides computational results for different algorithms a theoretical comparison of the enumerated search spaces (with and without different dominance criteria) is given.

Almost all solution algorithms (exact or heuristic procedures) for the RCPSP, multi-mode RCPSP or the RCPSP with generalized precedence relations were tested on benchmark instances which can be found in the so-called PSPLIB (project scheduling problem library) on the website [126]. These instances and their generation process are described in Kolisch et al. [89], [90] and Kolisch and Sprecher [91].

3.6 General Objective Functions for Problems without Resource Constraints

In this section we consider project scheduling problems with more general objective functions than the makespan, but assume that no resource constraints are given. More precisely, we study the following optimization problem (P)

$$\begin{aligned} \min \quad & f(S_0, \ldots, S_{n+1}) && (3.79)\\ \text{s.t.} \quad & S_j - S_i \geq d_{ij} \qquad ((i,j) \in A) && (3.80)\\ & S_0 = 0. && (3.81) \end{aligned}$$

As before, $V = \{0, \ldots, n+1\}$ is a set of activities (operations), where 0 and $n+1$ are a dummy starting and terminating activity, respectively. The set A is a set of directed arcs of a graph $G = (V, A)$ with vertex set V. Associated with each arc $(i,j) \in A$ is a value $d_{ij} \in \mathbb{Z}$ representing a minimal start-start distance between activities i and j. The vector $S = (S_i)_{i=0}^{n+1}$ represents the starting times of all activities $i = 0, \ldots, n+1$ in a nonpreemptive schedule (inducing completion times $C_i = S_i + p_i$), and $f(S) = f(S_0, \ldots, S_{n+1})$ may be interpreted as the costs of schedule S.

Additionally, we assume that a time interval $[0, T]$ is given, in which all activities have to be processed. As described in Section 3.2.1, the conditions $0 = S_0 \leq S_i \leq S_i + p_i \leq S_{n+1} \leq T$ may be covered by relations of the form (3.80) by defining minimal distances $d_{0i} = 0$, $d_{i,n+1} = p_i$ for all $i = 1, \ldots, n$ and $d_{n+1,0} = -T$.

For the remainder of this section we assume that a feasible solution S for problem (P) exists, which can be easily checked by longest path calculations. If the Floyd-Warshall algorithm detects a positive cycle in the network (V, A) according to the distances d, no feasible solution exists, i.e. we do not have to search for an optimal solution of (P).

Problem (P) with $d_{ij} = p_i$ for all $(i,j) \in A$ may be interpreted as an RCPSP in which the resource constraints are relaxed and the makespan objective function $\max_{i=0}^{n}\{S_i + p_i\}$ is replaced by an arbitrary objective function $f(S_0, \ldots, S_{n+1})$. Thus, the solution of (P) provides a lower bound for such an RCPSP.

In the following we will discuss solution methods for problem (P) under the assumption that the objective function $f(S_0, \ldots, S_{n+1})$ has additional properties. In Subsection 3.6.1 regular and antiregular objective functions are considered. Afterwards we assume that $f(S_0, \ldots, S_{n+1})$ is separable, i.e. f can be written as $f(S) = \sum_{i=0}^{n+1} f_i(S_i)$. While in Subsection 3.6.2 linear functions $f(S) = \sum_{i=0}^{n+1} b_i S_i$ are considered, in Subsection 3.6.3 all functions $f_i(S_i)$ are assumed to be convex piecewise linear. In both cases the problem can be reduced to a maximum cost flow problem. Finally, in Subsection 3.6.4 a pseudo-polyomial algorithm for a general sum objective function is derived.

3.6.1 Regular functions

On page 10 several examples for regular objective functions of the form $f(S) = \sum f_i(S_i)$ or $f(S) = \max f_i(S_i)$ have been introduced (like the weighted flow time $\sum w_i C_i$ with non-negative weights $w_i \geq 0$, the maximum lateness $L_{\max}$ or the total weighted tardiness $\sum w_i T_i$ with $w_i \geq 0$ for all i).

If f is regular, i.e. if $f(S) = f(S_0, \ldots, S_{n+1}) \leq f(S') = f(S'_0, \ldots, S'_{n+1})$ holds for all schedules S, S' with $S_i \leq S'_i$ for $i = 0, \ldots, n+1$, then an earliest-start schedule $S^* = (S^*_i)$ is optimal. It can be obtained by calculating the lengths ℓ_i of longest paths from 0 to i in the network (V, A) with arc distances d_{ij} and setting $S^*_i := \ell_i$ for $i = 0, \ldots, n+1$.

Symmetrically, for an antiregular objective function (i.e. $f(S) \geq f(S')$ for all schedules S, S' with $S_i \leq S'_i$ for $i = 0, \ldots, n+1$) a latest-start schedule is optimal. It can be obtained by reversing all arcs in A and calculating longest path lengths from the dummy terminating activity $n+1$ to each node i in the resulting network (V, A') with $A' := \{(j,i) \mid (i,j) \in A\}$ and arc distances $d'_{ji} := d_{ij}$ for all $(i,j) \in A$. If ℓ'_i denotes the calculated longest path lengths, activity i is started at time $T - \ell'_i$.

If the arc set A is acyclic, the longest path lengths may be calculated in a topological order in $O(|A|)$ time. For the general case (i.e. a cyclic graph with arbitrary distances $d_{ij} \in \mathbb{Z}$) a label-correcting algorithm with complexity $O(|V||A|)$ may be used (cf. Section 2.2.2).

3.6.2 Linear functions

A linear objective function has the form $f(S) = \sum_{i=0}^{n+1} f_i(S_i) = \sum_{i=0}^{n+1} b_i S_i$, where the coefficients b_i are given integers. If some of the b_i are negative, while others are positive, then the corresponding linear objective function is neither regular nor antiregular. An example for such a function is the weighted flow time $\sum_{i=0}^{n} w_i C_i = \sum_{i=0}^{n} w_i (S_i + p_i)$ with arbitrary weights $w_i \in \mathbb{Z}$ which (up to the additive constant $\sum_{i=0}^{n} w_i p_i$) is a linear function.

The corresponding problem (P) is equivalent to

$$\begin{aligned}
\min \quad & \sum_{i=0}^{n+1} b_i S_i && && (3.82)\\
\text{s.t.} \quad & S_j - S_i \geq d_{ij} && ((i,j) \in A) && (3.83)\\
& S_0 = 0 && && (3.84)
\end{aligned}$$

where b_0 can be chosen arbitrarily. The dual has the form

$$\max \quad \sum_{(i,j)\in A} d_{ij}x_{ij} \tag{3.85}$$

$$\text{s.t.} \quad \sum_{\{j|(j,i)\in A\}} x_{ji} - \sum_{\{j|(i,j)\in A\}} x_{ij} = b_i \qquad (i = 1,\ldots,n+1) \tag{3.86}$$

$$\lambda - \sum_{j=1}^{n+1} x_{0j} = b_0 \tag{3.87}$$

$$x_{ij} \geq 0 \qquad ((i,j) \in A) \tag{3.88}$$

$$\lambda \in \mathbb{R}. \tag{3.89}$$

If we choose b_0 such that $\sum_{i=0}^{n+1} b_i = 0$ holds, then by adding all equations (3.86) for $i = 1,\ldots,n+1$ and equation (3.87) we conclude $\lambda = 0$. Thus, with this choice of b_0 the dual has the form

$$\max \quad \sum_{(i,j)\in A} d_{ij}x_{ij} \tag{3.90}$$

$$\text{s.t.} \quad \sum_{\{j|(j,i)\in A\}} x_{ji} - \sum_{\{j|(i,j)\in A\}} x_{ij} = b_i \qquad (i = 1,\ldots,n+1) \tag{3.91}$$

$$\sum_{j=1}^{n+1} x_{0j} = -b_0 \tag{3.92}$$

$$x_{ij} \geq 0 \qquad ((i,j) \in A). \tag{3.93}$$

This problem is a maximum cost flow problem, which is equivalent to a minimum cost flow problem and can be polynomially solved by methods discussed in Section 2.4.6.

A special linear objective function, which is of practical interest in connection with the job-shop problem, is the function $\sum w_i(C_i - S_i)$, where $C_i - S_i$ denotes the difference between the completion time $C_{n_i,i}$ of the last operation of job i and the starting time S_{1i} of the first operation of job i. This situation can be handled as follows. For each job $i = i_1 \to i_2 \to \ldots \to i_{n_i}$ we introduce an additional starting job i^{min} and an additional finishing job i^{max} with distances $d_{i^{min},i_1} = 0$ and $d_{i_n,i^{max}} = p_{i_n}$. Then the corresponding variables $S_{i^{min}}$ and $S_{i^{max}}$ satisfy $S_{i^{min}} \leq S_i = S_{i_1}$ and $S_{i^{max}} \geq C_i = S_{i_{n_i}} + p_{n_i}$. By setting the coefficients in the objective function to $b_{i^{min}} := -w_i$, $b_{i^{max}} := w_i$ and $b_{i_j} := 0$ for all operations i_j of job i, we get the objective $\min \sum_{i=1}^{n} w_i(S_{i^{max}} - S_{i^{min}})$, which is equivalent to minimize $\sum w_i(C_i - S_i)$.

Also terms involving the lateness of jobs may be added to the objective function (3.82). If the maximum lateness of jobs in a subset I (e.g. corresponding to jobs in a partial project) is important, we introduce an additional job I with $d_{iI} = p_i - d_i$ for all jobs $i \in I$. Then $S_I \geq L_i = S_i + p_i - d_i$. By setting the

coefficient $b_I := 1$ we get the problem to minimize S_I subject to the conditions $S_I \geq L_i$ for all $i \in I$, which is equivalent to minimize $L_{\max}$ in the subset I.

3.6.3 Convex piecewise linear functions

In this section we generalize the preceding concepts derived for linear objective functions and study problem (P) with a sum objective function $f(S) = \sum_{i=0}^{n+1} f_i(S_i)$, where all functions f_i are convex piecewise linear and bounded from below. An example for such a function is the weighted earliness-tardiness $\sum_{i=1}^{n} w_i^E E_i + \sum_{i=1}^{n} w_i^T T_i$ with earliness weights $w_i^E \geq 0$ and tardiness weights $w_i^T \geq 0$.

A function $f : \mathbb{R} \to \mathbb{R}$ is called **convex** if for all $\lambda \in [0,1]$ and all $y_1, y_2 \in \mathbb{R}$ the inequality

$$f(\lambda y_1 + (1-\lambda)y_2) \leq \lambda f(y_1) + (1-\lambda) f(y_2)$$

holds. Furthermore, f is **piecewise linear** if breaking points $b_0 := -\infty < b_1 < b_2 < \ldots < b_m < b_{m+1} := \infty$ exist such that f is a linear function in each interval $]b_{k-1}, b_k[$ for $k = 1, \ldots, m+1$. We denote by Θ_k the slope of f in the interval $]b_{k-1}, b_k[$ for $k = 1, \ldots, m+1$, and assume that $f(y)$ achieves its minimum in b_l. Then we have $\Theta_1 < \Theta_2 < \ldots < \Theta_l \leq 0 \leq \Theta_{l+1} < \ldots < \Theta_{m+1}$ and $f(y)$ may be written as

$$f(y) = \beta + \sum_{k=1}^{l} \gamma_k (b_k - y)^+ + \sum_{k=l+1}^{m+1} \gamma_k (y - b_{k-1})^+ \tag{3.94}$$

where $y^+ := \max\{y, 0\}$, $\beta := f(b_l)$ and

$$\gamma_k := \Theta_{k+1} - \Theta_k \text{ for } k = 1, \ldots, l-1; \qquad \gamma_l := -\Theta_l; \qquad \gamma_{l+1} := \Theta_{l+1};$$
$$\gamma_k := \Theta_k - \Theta_{k-1} \text{ for } k = l+2, \ldots, m+1.$$

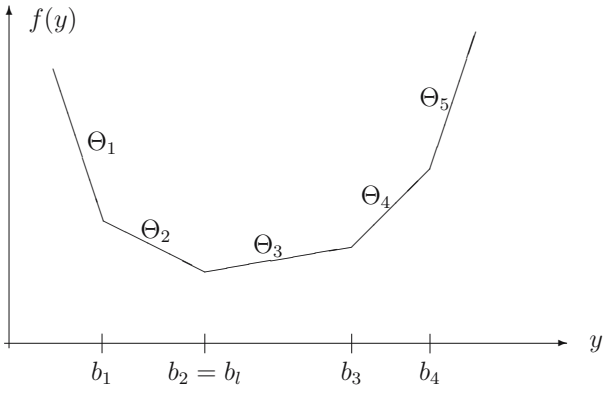


Figure 3.59: A convex piecewise linear function

Example 3.23: Consider the convex piecewise linear function shown in Figure 3.59. It has the representation

$$\begin{aligned} f(y) &= f(b_2) + (\Theta_2 - \Theta_1)(b_1 - y)^+ - \Theta_2(b_2 - y)^+ + \Theta_3(y - b_2)^+ \\ &\quad + (\Theta_4 - \Theta_3)(y - b_3)^+ + (\Theta_5 - \Theta_4)(y - b_4)^+. \end{aligned}$$

For example, for $y \in]-\infty, b_1[$ we have

$$\begin{aligned} f(y) &= f(b_2) + (\Theta_2 - \Theta_1)(b_1 - y)^+ - \Theta_2(b_2 - y)^+ \\ &= f(b_2) + (\Theta_2 - \Theta_1)(b_1 - y) - \Theta_2(b_2 - b_1) - \Theta_2(b_1 - y) \\ &= f(b_2) - \Theta_2(b_2 - b_1) - \Theta_1(b_1 - y) \end{aligned}$$

and for $y \in]b_4, \infty[$ we have

$$\begin{aligned} f(y) &= f(b_2) + \Theta_3(y - b_2) + (\Theta_4 - \Theta_3)(y - b_3) + (\Theta_5 - \Theta_4)(y - b_4) \\ &= f(b_2) + \Theta_3(b_3 - b_2) + \Theta_4(b_4 - b_3) + \Theta_5(y - b_4). \end{aligned}$$

Expressions for other y-values are derived similarly. □

Since f is convex, all values γ_k are non-negative. Furthermore, $(b_k - y)^+ = \min\{\alpha_k \mid \alpha_k \geq 0, y + \alpha_k \geq b_k\}$ and $(y - b_{k-1})^+ = \min\{\alpha_k \mid \alpha_k \geq 0, -y + \alpha_k \geq -b_{k-1}\}$ for $k = 1, \ldots, m+1$. Thus, the value $f(y)$ in (3.94) may also be calulated as the optimal objective value of the following linear program

$$f(y) = \min \quad \sum_{k=1}^{m+1} \gamma_k \alpha_k + \beta \tag{3.95}$$

$$\text{s.t.} \quad y + \alpha_k \geq b_k \qquad (k = 1, \ldots, l) \tag{3.96}$$

$$-y + \alpha_k \geq -b_{k-1} \qquad (k = l+1, \ldots, m+1) \tag{3.97}$$

$$\alpha_k \geq 0 \qquad (k = 1, \ldots, m). \tag{3.98}$$

We now consider problem (P) with a sum objective function

$$f(S_0, \ldots, S_{n+1}) = \sum_{\substack{i,j=0 \\ i \neq j}}^{n+1} f_{ij}(S_j - S_i), \tag{3.99}$$

where all functions f_{ij} are convex piecewise linear and bounded from below. Let $b_1^{ij}, b_2^{ij}, \ldots, b_{m_{ij}}^{ij}$ be the breaking points of f_{ij} and assume that l_{ij} is the breaking point where f_{ij} achieves its minimum. Then problem (P) with objective function (3.99) can be written as

$$\min \quad \sum_{\substack{i,j=0 \\ i \neq j}}^{n+1} \sum_{k=1}^{m_{ij}+1} \gamma_k^{ij} \alpha_k^{ij} + \beta \tag{3.100}$$

$$\begin{aligned}
\text{s.t.} \quad S_j - S_i + \alpha_k^{ij} &\geq b_k^{ij} && (i,j \in V,\ i \neq j; k = 1,\dots,l_{ij}) && (3.101)\\
S_i - S_j + \alpha_k^{ij} &\geq -b_{k-1}^{ij} && (i,j \in V,\ i \neq j; k = l_{ij}+1,\dots,m_{ij}+1) && (3.102)\\
S_j - S_i &\geq d_{ij} && ((i,j) \in A_0) && (3.103)\\
\alpha_k^{ij} &\geq 0 && (i,j \in V,\ i \neq j; k = 1,\dots,m_{ij}+1) && (3.104)\\
S_i &\in \mathbb{R} && (i \in V) && (3.105)
\end{aligned}$$

In connection with this problem it is convenient to introduce a multigraph, i.e. a directed graph with parallel arcs. Associated with each pair (i,j), $i,j = 0,\dots,n+1,\ i \neq j$, is a set of parallel arcs consisting of arcs

- $(i,j)_k$ for $k = 1,\dots,l_{ij}$, corresponding to the constraints (3.101),
- $(i,j)_k$ for $k = l_{ij}+1,\dots,m_{ij}+1$, corresponding to the constraints (3.102), and
- $(i,j)_0$ if $(i,j) \in A_0$, corresponding to the constraint (3.103).

Let A be the set of all these (possibly parallel) arcs. Then the dual (we ignore the constant β) has the form

$$\max \quad \sum_{\substack{i,j=0\\ i\neq j}}^{n+1} \sum_{k=1}^{l_{ij}} b_k^{ij} x_{ij}^k - \sum_{\substack{i,j=0\\ i\neq j}}^{n+1} \sum_{k=l_{ij}+1}^{m_{ij}+1} b_{k-1}^{ij} x_{ij}^k + \sum_{(i,j)\in A_0} d_{ij} x_{ij}^0 \tag{3.106}$$

$$\begin{aligned}
\text{s.t.} \quad \sum_{k=0}^{m} \sum_{\{j|(j,i)_k \in A\}} x_{ji}^k - \sum_{k=0}^{m} \sum_{\{j|(i,j)_k \in A\}} x_{ij}^k &= 0 && (i \in V) && (3.107)\\
x_{ij}^k &\leq \gamma_k^{ij} && ((i,j)_k \in A \setminus A_0) && (3.108)\\
x_{ij}^k &\geq 0 && ((i,j)_k \in A) && (3.109)
\end{aligned}$$

This problem is a maximum cost circulation problem, which can be solved polynomially by network flow methods (cf. Section 2.4.6).

In problem (P) with objective function (3.99) one may again assume $S_0 = 0$ because the problem does not change if each variable S_i is replaced by $S_i - S_0$.

3.6.4 General sum functions

In the following we consider problem (P) with an arbitrary sum objective

$$\begin{aligned}
\min \quad & \sum_{i=0}^{n+1} f_i(S_i) && && (3.110)\\
\text{s.t.} \quad & S_j - S_i \geq d_{ij} && ((i,j) \in A) && (3.111)\\
& S_0 = 0. && && (3.112)
\end{aligned}$$

From the distances d_{ij} we may calculate for each activity $i = 0, \ldots, n+1$ a time window $[ES_i, LS_i]$ consisting of an earliest starting time ES_i with respect to $ES_0 = 0$ and a latest starting time LS_i with respect to the given time horizon $LS_{n+1} = T$ (cf. Section 3.2.1). Recall that we assume that a feasible solution S for problem (P) exists, i.e. the network (V, A) with arc distances d contains no positive cycle. In this case ES_i is the length of a longest path from 0 to i and LS_i is equal to T minus the longest path length from i to $n+1$.

W.l.o.g. all function values $f_i(t)$ with $t \in \{ES_i, \ldots, LS_i\}$ can be assumed to be positive (otherwise we may add to each $f_i(t)$ the constant

$$C := \max_{i=0}^{n+1} \max_{t=ES_i}^{LS_i} \{|f_i(t)| + 1\}$$

resulting in an additive constant $(n+2)C$ for the objective function).

We will show that problem (P) can be reduced to a minimum cut problem in an arc-capacitated network $\tilde{G} = (\tilde{V}, \tilde{A})$ with arc capacities u, which is defined as follows:

- The set of nodes $\tilde{V}$ contains for each activity $i = 1, \ldots, n$ and each integer $t \in [ES_i, LS_i + 1]$ a node v_{it}. Furthermore, it contains a dummy source a and a dummy sink b. Thus,

$$\tilde{V} = \{a, b\} \cup \bigcup_{i=1}^{n} \{v_{it} \mid t \in \{ES_i, \ldots, LS_i + 1\}\}.$$

- The set of arcs $\tilde{A}$ can be divided into three disjoint subsets $\tilde{A}_F, \tilde{A}_T, \tilde{A}_D$.

  The set $\tilde{A}_F$ of **function-value arcs** is defined by

$$\tilde{A}_F := \bigcup_{i=1}^{n} \{(v_{it}, v_{i,t+1}) \mid t \in \{ES_i, \ldots, LS_i\}\}$$

  taking care of the objective function.

  The set $\tilde{A}_T$ of **temporal arcs** guarantees that no temporal constraint is violated. This set is defined by

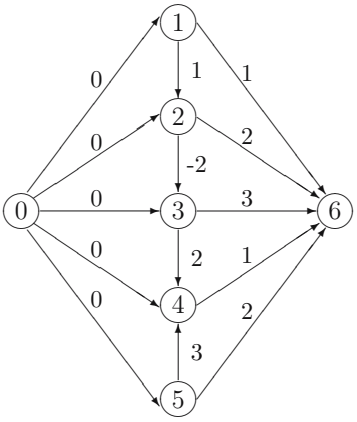
$$\tilde{A}_T := \bigcup_{i=1}^{n} \{ (v_{it}, v_{j,t+d_{ij}}) \mid (i,j) \in A; \; t \in \{\max(ES_i + 1, ES_j + 1 - d_{ij}), ..., \min(LS_i, LS_j - d_{ij})\}.$$

  The set $\tilde{A}_D$ of **dummy arcs** is defined by

$$\tilde{A}_D := \bigcup_{i=1}^{n} \{(a, v_{i,ES_i})\} \cup \bigcup_{i=1}^{n} \{(v_{i,LS_i+1}, b)\}.$$

- The arc capacities u of the function-value arcs $(v_{it}, v_{i,t+1})$ are equal to the function values $f_i(t)$. The capacities of all temporal and dummy arcs are equal to infinity.

Example 3.24: Consider a problem with $n = 5$ activities, processing times $p_1 = 1, p_2 = 2, p_3 = 3, p_4 = 1, p_5 = 2$, time horizon $T = 6$ and distances $d_{12} = 1,\ d_{23} = -2,\ d_{34} = 2,\ d_{54} = 3$ (cf. Figure 3.60). We introduce an additional dummy starting activity 0 and a dummy terminating activity $n+1 = 6$ and set $d_{0i} = 0$ and $d_{i,n+1} = d_{i6} = p_i$ for $i = 1, \ldots, 5$. With the initial values $ES_0 := 0$ and $LS_{n+1} = LS_6 := T = 6$, the remaining earliest and latest starting times are obtained by calculating corresponding longest path lengths l_{0i} and $l_{i,n+1}$ for $i = 1, \ldots, 5$ and setting $ES_i = l_{0i},\ LS_i = T - l_{i,n+1}$.



i	1	2	3	4	5
p_i	1	2	3	1	2
ES_i	0	1	0	3	0
$l_{i,n+1}$	3	2	3	1	4
LS_i	3	4	3	5	2

Figure 3.60: Earliest and latest starting times for Example 3.24

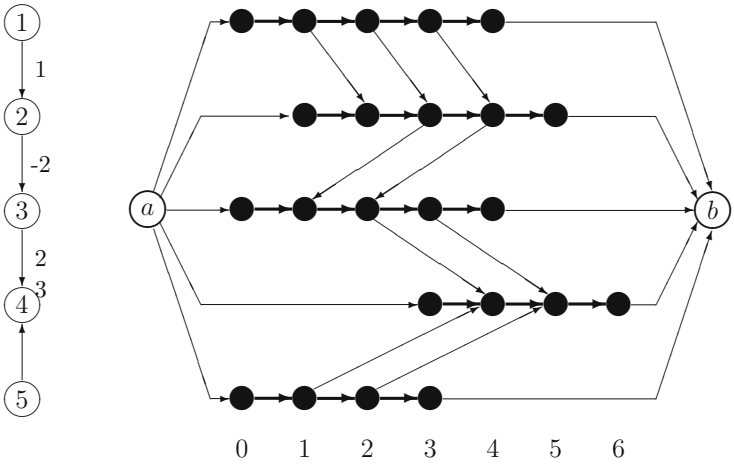


Figure 3.61: Network $\tilde{G} = (\tilde{V}, \tilde{A})$ for Example 3.24

The corresponding network $\tilde{G} = (\tilde{V}, \tilde{A})$ is shown in Figure 3.61. The chains in the rows consisting of nodes v_{it} correspond to the activities $i = 1, \ldots, 5$, the

columns to the time periods $t = 0, \ldots, 6$. The function-value arcs are drawn in a thick mode, all other arcs have infinite capacity. □

In the following we will show that our problem can be reduced to a minimum cut problem in $\tilde{G}$. Recall that an a, b-cut in $\tilde{G}$ is a pair $[X, \overline{X}]$ of disjoint sets $X, \overline{X} \subseteq \tilde{V}$ with $X \cup \overline{X} = \tilde{V}$ and $a \in X$, $b \in \overline{X}$ (cf. Section 2.4.4). The capacity $u[X, \overline{X}]$ of such a cut $[X, \overline{X}]$ is the sum

$$u[X, \overline{X}] := \sum_{\substack{v \in X \\ \overline{v} \in \overline{X}}} u_{v,\overline{v}}$$

of the capacities of forward arcs, i.e. all arcs $(v, \overline{v})$ with $v \in X$, $\overline{v} \in \overline{X}$.

At first we will show that for each feasible solution S of (P) an a, b-cut $[X, \overline{X}]$ exists such that $f(S) = u[X, \overline{X}]$. This implies that if S^* is an optimal solution for problem (P) and $[X^*, \overline{X}^*]$ is a minimum a, b-cut in $\tilde{G}$, then $f(S^*) \geq u[X^*, \overline{X}^*]$ holds. In a second step we will show that a minimum a, b-cut $[X^*, \overline{X}^*]$ provides a fesible solution S with $f(S) = u[X^*, \overline{X}^*]$. This solution must also be optimal for (P).

Constructing an a, b-cut from a feasible schedule

Given a feasible solution $S = (S_i)$, we define a cut $[X, \overline{X}]$ in $\tilde{G}$ by

$$X := \bigcup_{i=1}^{n} \{v_{it} \mid t \leq S_i\} \cup \{a\} \text{ and } \overline{X} := \tilde{V} \setminus X. \tag{3.113}$$

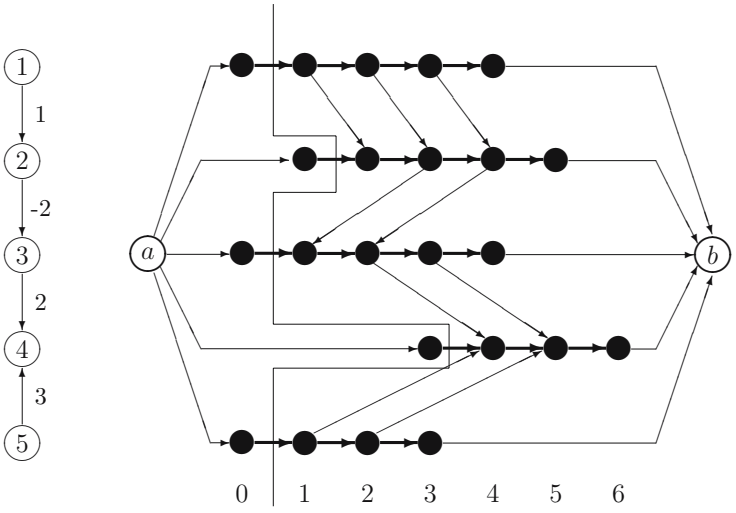


Figure 3.62: Cut in $\tilde{G} = (\tilde{V}, \tilde{A})$ corresponding to the earliest start schedule

For example, for the earliest start schedule $S = (ES_i)$ from Example 3.24 we get the cut shown in Figure 3.62.

We now show that the capacity $u[X, \overline{X}]$ of this cut equals $f(S)$. All arcs (v_{iS_i}, v_{i,S_i+1}) with $i = 1, \ldots, n$ are forward arcs of $[X, \overline{X}]$, and the sum of the capacities of these arcs is equal to $f(S)$. Furthermore, no temporal arc is contained in $[X, \overline{X}]$ because if $v_{is} \in X$ and $v_{jt} \in \overline{X}$ then $s \leq S_i$ and $t > S_j$ due to the definition (3.113) of the cut. Since S is feasible, we have $d_{ij} \leq S_j - S_i < t - s$, i.e. $s + d_{ij} < t$. But then (v_{is}, v_{jt}) cannot be a temporal arc since for such an arc $t = s + d_{ij}$ holds. Thus, the cut contains only function-value arcs and $u[X, \overline{X}] = f(S)$.

Constructing a feasible schedule from a minimum a, b-cut

At first we will show that a minimum a, b-cut of $\tilde{G}$ contains for each activity $i = 1, \ldots, n$ exactly one function-value arc $(v_{it}, v_{i,t+1})$ as forward arc.

From the preceding arguments we know that a feasible solution S of (P) induces an a, b-cut with finite capacity (defined by (3.113). Thus, also a minimum a, b-cut $[X, \overline{X}]$ has a finite capacity, i.e. it does not contain any of the dummy or temporal arcs as forward arcs. It follows that for each $i = 1, \ldots, n$ at least one function-value arc $(v_{it}, v_{i,t+1})$ is contained as forward arc in the cut $[X, \overline{X}]$. Assume that $[X, \overline{X}]$ for some i contains more than one of such function value arcs. Then we derive a contradiction by constructing a cut $[X', \overline{X'}]$ which has a smaller capacity than $[X, \overline{X}]$.

For each i let t_i be the smallest index such that $(v_{it_i}, v_{i,t_i+1}) \in (X, \overline{X})$. Define $X' := \bigcup_{i=1}^{n} \{v_{it} \mid t \leq t_i\} \cup \{a\}$ and $\overline{X'} := \tilde{V} \setminus X'$. Clearly, $X' \subset X$ and the set of function-value arcs (v_{it_i}, v_{i,t_i+1}) is a proper subset of the corresponding set of function-value arcs of $(X, \overline{X})$. Recall that the weights $f_i(t)$ of all arcs $(v_{it}, v_{i,t+1})$ are positive. Thus, it is sufficient to show that $[X', \overline{X'}]$ does not contain any of the temporal arcs as forward arc. So assume that there exists a temporal arc $(v_{is}, v_{jt}) \in (X', \overline{X'})$ with $t = s + d_{ij}$ and $s \leq t_i$, $t > t_j$. Since $(v_{is}, v_{jt}) \in \tilde{A}$, it follows by the construction of $\tilde{G}$ that all arcs $(v_{i,s-z}, v_{j,t-z}) \in \tilde{A}$ for all integers z such that $ES_i + 1 \leq s - z \leq LS_i$ and $ES_j + 1 \leq t - z \leq LS_j$. Now let $z := t - t_j - 1$. Then

$$ES_j + 1 \leq t_j + 1 = t - z \leq t \leq LS_j$$

and

$$ES_i + 1 \leq t_i + 1 \leq t_j - d_{ij} + 1 = t_j + (s - t) + 1 = s - z \leq s \leq LS_i,$$

i.e. $(t - z) \in [ES_j + 1, LS_j]$ and $(s - z) \in [ES_i + 1, LS_i]$.

Since $s \leq t_i$ we have $v_{i,s-z} \in X' \subset X$. Moreover, by definition of t_j we have $v_{j,t-z} = v_{j,t_j+1} \in \overline{X}$. Thus, we have identified a temporal arc $(v_{i,s-z}, v_{j,t-z}) \in (X, \overline{X})$, which contradicts the fact that $[X, \overline{X}]$ has a finite capacity.

Now, in a second step we will construct a feasible schedule from a minimum cut. Given a minimum cut $[X^*, \overline{X^*}]$ in $\tilde{G}$ (which has finite capacity), for each $i = 1, \ldots, n$ consider the unique function-value arc $(v_{it_i}, v_{i,t_i+1}) \in (X^*, \overline{X^*})$. If we define $S_i := t_i$ for $i = 1, \ldots, n$, then the corresponding vector $S = (S_i)$ satisfies $f(S) = u[X^*, \overline{X^*}]$. It remains to show that S is feasible, i.e. that the constraints (3.111) are satisfied. Assume that for $s := S_i$ and $t := S_j$ we have a violated constraint (3.111), i.e. $s + d_{ij} > t$. Then for the temporal arc $(v_{is}, v_{i,s+d_{ij}})$ in $\tilde{G}$ we have $v_{is} \in X^*$ and $v_{j,s+d_{ij}} \in \overline{X^*}$, which contradicts the fact that $u[X^*, \overline{X^*}]$ is finite. Thus, S is feasible.

We have proven

Theorem 3.8 If problem (P) with an arbitrary objective function $f(S) = \sum f_i(S_i)$ has a feasible solution, then an a, b-cut $[X^*, \overline{X^*}]$ of $\tilde{G} = (\tilde{V}, \tilde{A})$ with minimum capacity corresponds to an optimal solution S^* of problem (P) and $u[X^*, \overline{X^*}] = f(S^*)$. Moreover, $S^* = (S_i^*)$ is given by $S_i^* = t_i$ where (v_{i,t_i}, v_{i,t_i+1}) is the unique forward arc of activity i in $(X^*, \overline{X^*})$.

A minimum cut in $\tilde{G}$ can be calculated by solving a maximum flow problem for $\tilde{G}$ (cf. Section 2.4.4). Let $m := |A|$ be the number of relations (3.111). Since $\max_{i=0}^{n+1}(LS_i - ES_i) = O(T)$, the graph $\tilde{G} = (\tilde{V}, \tilde{A})$ has $|\tilde{V}| = O(nT)$ nodes and $|\tilde{A}| = O((n+m)T) = O(mT)$ arcs. Thus, by applying the FIFO preflow-push algorithm for maximum flows (cf. Section 2.4.5) the minimum cut can be computed in $O(|\tilde{V}||\tilde{A}| \log(|\tilde{V}|^2/|\tilde{A}|) = O(nmT^2 \log \frac{n^2 T}{m})$ time.

If each function f_i is explicitly given by its values $f_i(t)$ for $t \in \{ES_i, \ldots, LS_i\}$, then the input length of (P) is bounded by $O(|A| + nT) = O(m + nT)$. Thus, in this case the presented algorithm is polynomial. If, on the other hand, the functions $f_i(t)$ are given in a more compact form (e.g. by formulas which can be evaluated in polynomial time), then the input length of (P) is bounded by $O(m + \log T)$ since only the value T has to be encoded. Thus, in this case the presented algorithm is only pseudo-polynomial.

3.6.5 Reference notes

Problem (P) with linear or convex piecewise linear functions was studied by Wennink [142]. He showed that in both cases the problem can be reduced to a maximum cost flow problem and developed local search algorithms for solving it. Möhring et al. [107], [108] studied problem (P) with a general sum objective function in connection with lower bound calculations for the RCPSP based on Lagrangian Relaxation. They showed that the problem can be reduced to a minimum cut problem and used it to provide lower and upper bounds for the RCPSP.

Chapter 4

Complex Job-Shop Scheduling

This chapter is devoted to another important basic scheduling problem: the job-shop problem. This problem is a special case of the RCPSP where only disjunctive resources (machines) and special precedence constraints in form of chains are given. After giving a precise formulation of the job-shop problem in Section 4.1, solution algorithms are discussed. While Section 4.2 deals with heuristic methods, foundations for exact branch-and-bound algorithms are presented in Section 4.3. Afterwards, generalizations and applications of the job-shop problem are considered. In Section 4.4 the concepts of time-lags and blocking are introduced. Then, we consider job-shop problems with flexible machines in Section 4.5, job-shop problems with transport robots in Section 4.6 and job-shop problems with limited buffers in Section 4.7.

4.1 The Job-Shop Problem

In this section we will introduce the job-shop problem. After giving a formulation of the problem in Subsection 4.1.1, in Subsection 4.1.2 the disjunctive graph model is presented. This model is very important in order to construct feasible schedules from sequences.

4.1.1 Problem formulation

The classical **job-shop problem** may be formulated as follows. We are given m machines $M_1, \ldots, M_m$ and N jobs $J_1, \ldots, J_N$. Job J_j consists of n_j operations O_{ij} $(i = 1, \ldots, n_j)$ which have to be processed in the order $O_{1j} \rightarrow O_{2j} \rightarrow \ldots \rightarrow O_{n_j,j}$. Operation O_{ij} must be processed for $p_{ij} > 0$ time units without preemption on a dedicated machine $\mu_{ij} \in \{M_1, \ldots, M_m\}$. Each machine can process only one job at a time. Furthermore, w.l.o.g. we may assume $\mu_{ij} \neq \mu_{i+1,j}$ for all $j = 1, \ldots, N$ and $i = 1, \ldots, n_j - 1$, i.e. no two subsequent operations of a job are processed on the same machine (otherwise the two operations may be replaced by one operation). Such a job-shop is also called a job-shop without machine repetition.

If not stated differently, we assume that there is sufficient buffer space between the machines to store a job if it finishes on one machine and the next machine is still occupied by another job.

A schedule $S = (S_{ij})$ is defined by the starting times of all operations O_{ij}. A schedule S is feasible if

- $S_{ij} + p_{ij} \leq S_{i+1,j}$ for all jobs $j = 1, \ldots, N$ and $i = 1, \ldots, n_j - 1$, i.e. the precedences $O_{ij} \rightarrow O_{i+1,j}$ are respected, and
- $S_{ij} + p_{ij} \leq S_{uv}$ or $S_{uv} + p_{uv} \leq S_{ij}$ for all pairs O_{ij}, O_{uv} of operations with $\mu_{ij} = \mu_{uv}$, i.e. each machine processes only one job at a time.

The objective is to determine a feasible schedule S with minimal makespan $C_{\max} = \max\limits_{j=1}^{N} \{C_j\}$, where $C_j = S_{n_j,j} + p_{n_j,j}$ is the completion time of job J_j.

Sometimes it is convenient to simplify the notations for a job-shop problem by identifying the operations O_{ij} by numbers $1, \ldots, n$, where $n := \sum\limits_{j=1}^{N} n_j$. If no confusions arise, we again use the index i for the operations $i = 1, \ldots, n$. Then the processing time of operation i is simply denoted by p_i, the job to which i belongs is denoted by $J(i)$ and $\mu(i)$ denotes the machine on which i has to be processed. Often it is also useful to introduce a dummy starting operation 0 and a dummy terminating operation $n + 1$ with $p_0 = p_{n+1} = 0$. For an operation $i = O_{uv}$ let $\sigma(i) = O_{u+1,v}$ be its successor where $\sigma(i) = n + 1$ if i is the last operation of a job. Furthermore, the first operation of each job is a successor of the starting operation 0.

A feasible schedule S defines for each machine M_k an order (machine sequence) $\pi^k = (\pi^k(1), \ldots, \pi^k(m_k))$ in which all m_k operations i with $\mu(i) = M_k$ are processed on M_k.

A **flow-shop scheduling problem** is a special case of the job-shop problem in which each job has exactly m operations and the i-th operation of each job must be processed on machine M_i for $i = 1, \ldots, m$. A flow-shop problem is called a **permutation flow-shop problem** if the jobs must be processed in the same order on all machines (i.e. we must have $\pi^1 = \ldots = \pi^m$).

4.1.2 The disjunctive graph model

The so-called disjunctive graph model is used to represent schedules for the job-shop problem. A **disjunctive graph** $G = (V, C, D)$ is a mixed graph with vertex set V, a set C of directed arcs (conjunctions) and a set D of undirected arcs (disjunctions). In connection with the job-shop problem G is defined as follows:

- The set V of vertices represents the set of all operations (including the dummy operations 0 and $n + 1$). Each vertex i is weighted with the corresponding processing time p_i.

- The set C of **conjunctions** represents the precedence constraints between consecutive operations of the same job. Thus, we have a conjunction $i \to \sigma(i)$ for each operation i. Additionally, we have conjunctions $0 \to i$ for each first operation i of a job. The meaning of a conjunction $i \to j$ is that a feasible schedule S must satisfy the condition $S_i + p_i \leq S_j$.

- The set D of **disjunctions** represents the different orders in which operations on the same machine may be scheduled. It consists of undirected arcs between all pairs of operations which have to be processed on the same machine, i.e. for any two operations i, j with $\mu(i) = \mu(j)$ (and $J(i) \neq J(j)$) the set D contains an undirected arc denoted by $i - j$. The meaning of a disjunction $i-j$ is that a feasible schedule S must either satisfy $S_i+p_i \leq S_j$ or $S_j + p_j \leq S_i$.

The problem of finding a feasible schedule for the job-shop problem is equivalent to the problem of fixing a direction for each disjunction such that the corresponding graph does not contain a cycle. A set $\mathcal{S}$ of fixed disjunctions is called a **selection**. If for each disjunction a direction has been fixed, then the corresponding selection is called **complete**. If additionally the graph $G(\mathcal{S}) = (V, C \cup \mathcal{S})$ is acyclic, then $\mathcal{S}$ is called **consistent**.

Given a complete consistent selection $\mathcal{S}$, we may construct a corresponding feasible earliest start schedule as follows. For each operation i let r_i be the length of a longest path from 0 to i in $G(\mathcal{S})$, where the length of a path is defined as the sum of the weights of all vertices on the path, vertex i excluded. We consider the schedule where each operation i is started at time $S_i := r_i$. Obviously, this schedule S is feasible and no operation can be shifted to the left without changing the orders defined by $\mathcal{S}$. Thus, the set of schedules represented by selections is equal to the set of semi-active schedules (cf. the classification of schedules in Section 3.4.1). Furthermore, the makespan $C_{\max}(S)$ of schedule S is equal to the length r_{n+1} of a longest path from 0 to $n+1$.

On the other hand, any feasible schedule S defines an order for all operations to be processed on the same machine. This induces a complete consistent selection $\mathcal{S}$. If the objective function is regular, the corresponding earliest start schedule does not have a larger objective function value than S. Thus, for regular objective functions always a complete consistent selection exists which represents an optimal schedule. Hence, it is sufficient to consider the set of all schedules defined by complete consistent selections.

On the other hand, if a non-regular objective function is given, in general the earliest start schedule is not optimal for a given selection. In order to get an optimal solution for the (sub)problem defined by a selection (a so-called selection-optimal schedule) we have to solve problem (P) from Section 3.6. Due to the results in Section 3.6 for special objective functions this problem is easy to solve. Note that in this situation the set A in (3.80) is given by the combined set of conjunctions C and selected arcs $\mathcal{S}$.

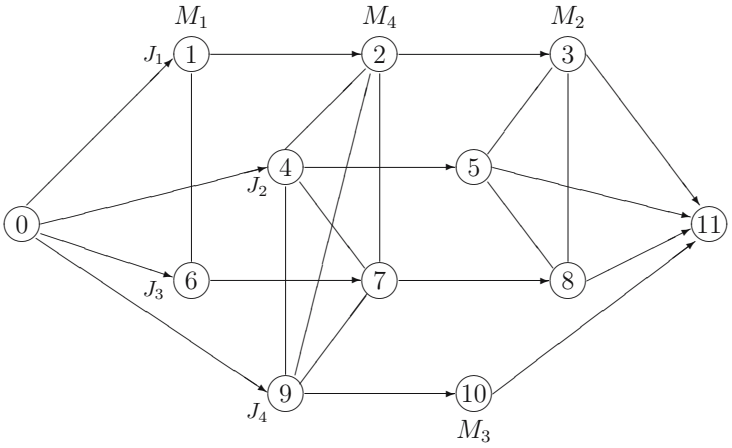


Figure 4.1: Disjunctive graph for a job-shop

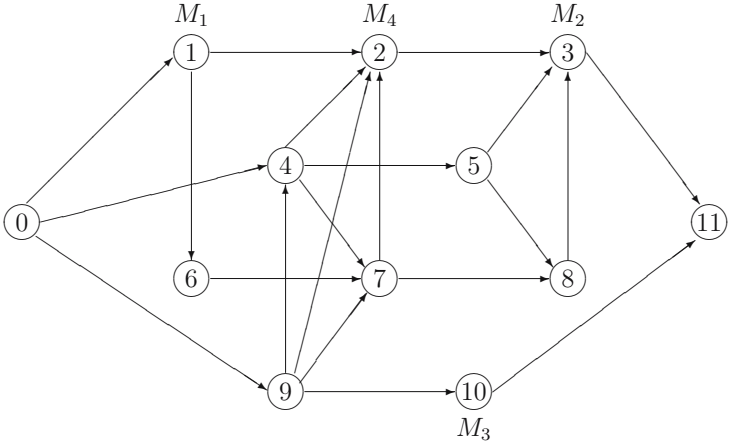


Figure 4.2: A complete selection

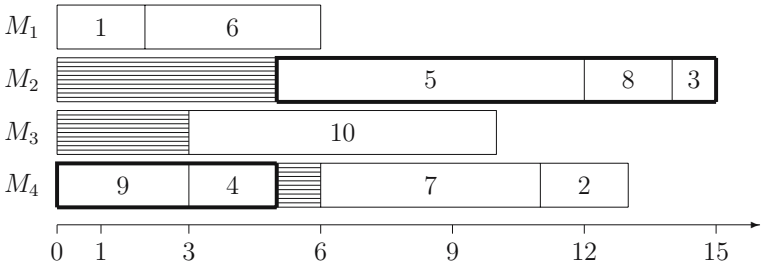


Figure 4.3: A corresponding schedule

Example 4.1: Consider an example with $N = 4$ jobs and $m = 4$ machines, where jobs $j = 1, 3$ consist of $n_j = 3$ operations and jobs $j = 2, 4$ consist of $n_j = 2$ operations. Furthermore, for the $n = 10$ operations $i = 1, \ldots, 10$ the following data is given:

i	1	2	3	4	5	6	7	8	9	10
p_i	2	2	1	2	7	4	5	2	3	7
$J(i)$	1	1	1	2	2	3	3	3	4	4
$\mu(i)$	M_1	M_4	M_2	M_4	M_2	M_1	M_4	M_2	M_4	M_3

The corresponding disjunctive graph can be found in Figure 4.1. We have machine disjunctions $1-6$ on M_1, $2-4$, $2-7$, $2-9$, $4-7$, $4-9$, $7-9$ on M_4, and $3-5$, $3-8$, $5-8$ on M_2.

A complete consistent selection is shown in Figure 4.2. In this selection on machine M_1 we have the sequence $\pi^1 = (1, 6)$, on M_2 we have the sequence $\pi^2 = (5, 8, 3)$, on M_3 the sequence $\pi^3 = (10)$ and on M_4 the sequence $\pi^4 = (9, 4, 7, 2)$. The earliest start schedule S associated with this selection is depicted in Figure 4.3. The makespan of this schedule is $C_{\max}(S) = 15$, a critical path is $0 \to 9 \to 4 \to 5 \to 8 \to 3 \to 11$ with length $p_9 + p_4 + p_5 + p_8 + p_3 = 15$. □

The job-shop problem to minimize the makespan can also be formulated as a linear program with additional disjunctive constraints (a so-called **disjunctive linear program**):

$$\begin{aligned}
\min \quad & S_{n+1} && && (4.1)\\
\text{s.t.} \quad & S_i + p_i \le S_j && (i \to j \in C) && (4.2)\\
& S_i + p_i \le S_j \vee S_j + p_j \le S_i && (i - j \in D) && (4.3)\\
& S_i \ge 0 && (i = 0, \ldots, n+1) && (4.4)
\end{aligned}$$

Usually S_0 is set to zero. In this case conditions (4.4) are implied by conditions (4.2). This disjunctive linear program can be transformed into a a mixed integer linear program by introducing a binary variable y_{ij} for each disjunction in (4.3) where

$$y_{ij} = \begin{cases} 1, & \text{if } i \text{ is scheduled before } j \\ 0, & \text{otherwise.} \end{cases}$$

Then we get the following mixed integer program, where M is a large integer:

$$\begin{aligned}
\min \quad & S_{n+1} && && (4.5)\\
\text{s.t.} \quad & S_i + p_i \le S_j && (i \to j \in C) && (4.6)\\
& S_i + p_i - M(1 - y_{ij}) \le S_j && (i - j \in D) && (4.7)\\
& S_j + p_j - M y_{ij} \le S_i && (i - j \in D) && (4.8)\\
& S_i \ge 0 && (i = 0, \ldots, n+1) && (4.9)\\
& y_{ij} \in \{0, 1\} && (i - j \in D) && (4.10)
\end{aligned}$$

4.2 Heuristic Methods

In this section we describe some foundations concerning heuristics for the job-shop problem. Especially, we concentrate on the definition of problem-specific neighborhoods for local search algorithms which are based on the disjunctive graph model. A natural way to go from one complete selection $\mathcal{S}$ to another selection $\mathcal{S}'$ is to change the orientation of certain disjunctive arcs. In connection with such an approach two questions arise:

1. Can it easily be decided whether a neighbor selection $\mathcal{S}'$ is feasible, i.e. the corresponding disjunctive graph $G(\mathcal{S}')$ is acyclic?

2. Does the neighbor selection $\mathcal{S}'$ lead to a better objective function value, i.e. does $C_{\max}(S') < C_{\max}(S)$ for the corresponding schedules S, S' hold?

Besides these two questions we also deal with the question whether a certain neighborhood is connected or at least opt-connected (cf. the definitions in Section 2.7).

Consider again the schedule in Figure 4.3 for Example 4.1. It is easy to see that by changing the orders of operations which do not belong to the critical path the makespan cannot be reduced. Thus, if we want to improve the current solution, we have to "destroy" all critical paths. In the following we assume that we are given a complete consistent selection $\mathcal{S}$ with the corresponding schedule S. We want to determine a neighbor selection $\mathcal{S}'$ which is feasible and for which $C_{\max}(S') < C_{\max}(S)$ holds.

A first neighborhood $\mathcal{N}_{ca}$ may be defined by reversing critical arcs. More specifically, $\mathcal{N}_{ca}(\mathcal{S})$ contains all selections $\mathcal{S}'$ which can be obtained from $\mathcal{S}$ by reversing one oriented disjunctive arc $(i,j) \in \mathcal{S}$ on some critical path in $G(\mathcal{S})$ into the arc (j,i). This neighborhood has the property that all selections $\mathcal{S}' \in \mathcal{N}_{ca}(\mathcal{S})$ are feasible, i.e. define complete consistent selections.

This can be proved as follows. Assume to the contrary that a selection $\mathcal{S}' \in \mathcal{N}_{ca}(\mathcal{S})$ is not feasible, i.e. the graph $G(\mathcal{S}')$ contains a cycle. Because $\mathcal{S}$ is assumed to be feasible, $G(\mathcal{S})$ is acyclic. Thus, the reversed arc $(j,i) \in \mathcal{S}'$ must be part of the cycle in $G(\mathcal{S}')$ implying that a path from i to j in $G(\mathcal{S}')$ exists which consists of at least three operations i, h, j. This path must also be contained in $G(\mathcal{S})$ and must be longer than the path consisting of the single arc $(i,j) \in \mathcal{S}$ because all processing times are assumed to be positive. Since this is a contradiction to the fact that the arc (i,j) belongs to a longest path in $G(\mathcal{S})$, the graph $G(\mathcal{S}')$ cannot be cyclic.

In the case that the neighborhood $\mathcal{N}_{ca}(\mathcal{S})$ of a selection $\mathcal{S}$ is empty, we may conclude that $\mathcal{S}$ is optimal. If $\mathcal{N}_{ca}(\mathcal{S}) = \emptyset$ holds, then the chosen critical path contains no disjunctive arc. But then the critical path contains only conjunctive arcs which implies that $C_{\max}(S)$ is equal to the total processing time of a job. Since this value is a lower bound for the optimal makespan, S must be optimal.

It is easy to construct examples which show that the neighborhood $\mathcal{N}_{ca}$ is not connected, i.e. it is not possible to transform two arbitrary selections $\mathcal{S}, \mathcal{S}'$ into each other by only reversing critical arcs. But, we have

Theorem 4.1 The neighborhood $\mathcal{N}_{ca}$ is opt-connected.

Proof: In order to show that $\mathcal{N}_{ca}$ is opt-connected, consider an arbitrary non-optimal selection $\mathcal{S}$ and an optimal selection $\mathcal{S}^*$. We have to show that we can transform $\mathcal{S}$ into $\mathcal{S}^*$ or into another optimal selection by only reversing critical arcs. If $\mathcal{S}$ is not optimal, a critical arc $(i,j) \in \mathcal{S}$ exists which does not belong to $\mathcal{S}^*$ (since otherwise we would have $C_{\max}(S) \leq C_{\max}(S^*)$ for the corresponding schedules S and S^*). By reversing the arc $(i,j) \in \mathcal{S}$ we transform $\mathcal{S}$ into a new feasible selection $\mathcal{S}'$. If $\mathcal{S}'$ is not optimal, we can argue as above. Since the number of critical arcs not belonging to $\mathcal{S}^*$ is finite, repeating such transformations leads to $\mathcal{S}^*$ or another optimal selection in a finite number of steps. □

A disadvantage of $\mathcal{N}_{ca}$ is that several neighbors $\mathcal{S}'$ exist which do not improve the makespan of $\mathcal{S}$. If P^S is a critical path in $G(\mathcal{S})$, a sequence $i_1, \ldots, i_k$ of at least two successive operations (i.e. $k > 1$) on P^S is called a (machine) **block** if the operations of the sequence are processed consecutively on the same machine, and enlarging the subsequence by one operation leads to a subsequence which does not fulfill this property.

For example, in the schedule shown in Figure 4.3 the critical path $0 \to 9 \to 4 \to 5 \to 8 \to 3 \to 11$ contains the two blocks $(9,4)$ and $(5,8,3)$ on M_4 and M_2, respectively.

Using this definition the following property can be proved:

Theorem 4.2 Let $\mathcal{S}$ be a complete consistent selection with makespan $C_{\max}(S)$ and let P^S be a critical path in $G(\mathcal{S})$. If another complete consistent selection $\mathcal{S}'$ with $C_{\max}(S') < C_{\max}(S)$ exists, then in $\mathcal{S}'$ at least one operation of a block B on P^S has to be processed before the first or after the last operation of B.

The proof of this property is based on the fact that if the first and the last operation of each block remain the same, all permutations of the operations in the inner parts of the blocks do not shorten the length of the critical path.

Based on this theorem the following block shift neighborhood may be defined: $\mathcal{N}^1_{bs}(\mathcal{S})$ contains all feasible selections $\mathcal{S}'$ which can be obtained from $\mathcal{S}$ by shifting an operation of some block on a critical path to the beginning or the end of the block. Note that in this neighborhood the orientation of several disjunctive arcs may be changed. For this reason, feasibility of neighbor selections must explicitly be stated because it is not automatically fulfilled as in $\mathcal{N}_{ca}$. Since infeasible moves are not allowed, also the question of opt-connectivity is more involved. It is still open whether neighborhood $\mathcal{N}^1_{bs}$ is opt-connected or not. However, $\mathcal{N}^1_{bs}$ can be enlarged to an opt-connected neighborhood $\mathcal{N}^2_{bs}$ by allowing some additional moves. If a move of an operation to the beginning or the end

of a block is infeasible, then we move the operation to the first (last) position in the block such that the resulting selection is feasible. By similar arguments as above it can be shown that neighborhood $\mathcal{N}_{bs}^2$ is opt-connected.

In the case that the neighborhood $\mathcal{N}_{bs}^2(\mathcal{S})$ of a selection $\mathcal{S}$ is empty, we may again conclude that $\mathcal{S}$ is optimal. If $\mathcal{N}_{bs}^2(\mathcal{S}) = \emptyset$ holds, then the chosen critical path contains no blocks. But then as for the neighborhood $\mathcal{N}_{ca}$, the critical path contains only conjunctive arcs which implies that $C_{\max}(S)$ is equal to a lower bound value.

A disadvantage of neighborhoods $\mathcal{N}_{bs}^1$ and $\mathcal{N}_{bs}^2$ is that they may be quite large (each critical operation is moved to two other positions). For this reason a smaller neighborhood $\mathcal{N}_{ca}^2 \subseteq \mathcal{N}_{ca} \cap \mathcal{N}_{bs}^1$ was proposed. $\mathcal{N}_{ca}^2(\mathcal{S})$ contains all selections $\mathcal{S}'$ which can be obtained from $\mathcal{S}$ by interchanging the first two or the last two operations of a block. Since $\mathcal{N}_{ca}^2(\mathcal{S})$ is a subset of $\mathcal{N}_{ca}(\mathcal{S})$, all selections in $\mathcal{N}_{ca}^2(\mathcal{S})$ are feasible. Furthermore, due to Theorem 4.2 all selections $\mathcal{S}' \in \mathcal{N}_{ca}(\mathcal{S}) \setminus \mathcal{N}_{ca}^2(\mathcal{S})$ satisfy $C_{\max}(S') \geq C_{\max}(S)$ since only operations in the inner part of blocks are changed. Thus, only non-improving solutions are excluded in the reduced neighborhood $\mathcal{N}_{ca}^2$.

All these neighborhoods may be used in connection with local search methods or genetic algorithms. Computational experiments have shown that among such algorithms tabu search provides the best results. In connection with tabu search or other local search methods the selection process of a neighbor for a given feasible solution is the most time consuming part. To determine the best neighbor for a given solution, the makespan has to be computed for each solution in the neighborhood. Due to the fact that each operation has at most two predecessors (a job and a machine predecessor), the makespan can be calculated in $O(n)$ time by longest path calculations where n is the number of operations. This effort can be reduced considerably by the following considerations.

As discussed in Section 4.1.1, a schedule can be specified by a vector $\pi = (\pi^1, \ldots, \pi^m)$ where for $k = 1, \ldots, m$ the sequence $\pi^k = (\pi^k(1), \ldots, \pi^k(m_k))$ defines the order in which all m_k operations i with $\mu(i) = M_k$ are processed on M_k. Let $G(\pi)$ be the corresponding directed graph achieved by fixing disjunctive arcs according to π. A solution π is feasible if and only if $G(\pi)$ contains no cycles.

Let π be a feasible solution and let π' be a neighbor of π with respect to the neighborhood $\mathcal{N}_{ca}$. The solution π' is derived from π by reversing a critical arc (i, j) (satisfying $\mu(i) = \mu(j)$) in $G(\pi)$. This means that i and j are consecutive operations in some machine sequence π^k which are swapped when moving from π to π'. We will denote such a solution π' by $\pi^{(i,j)}$. For a feasible schedule π and operations u, v denote by $P_\pi(u \vee v)$ and $P_\pi(\overline{u})$ the set of all paths in $G(\pi)$ "containing nodes u or v" and "not containing u", respectively. Furthermore, $P_\pi(\overline{u} \wedge \overline{v})$ and $P_\pi(\overline{u} \wedge v)$ denote the set of all paths in $G(\pi)$ "not containing nodes u and v" and "not containing u but containing v", respectively. For a given set of paths P the length of a longest path in P is denoted by $l(P)$.

Property 4.1 The makespan of a solution $\pi' := \pi^{(i,j)} \in \mathcal{N}_{ca}(\pi)$ is given by

$$C_{\max}(\pi') = \max\{\, l(P_{\pi'}(i \vee j)),\, l(P_\pi(\bar{i}))\}. \tag{4.11}$$

Proof: Since in π' only the arc (i,j) is reversed, a path not containing i and j is contained in $G(\pi')$ if and only if it is contained in $G(\pi)$, i.e. $P_{\pi'}(\bar{i} \wedge \bar{j}) = P_\pi(\bar{i} \wedge \bar{j})$. Any path in $G(\pi)$ containing j but not i is also contained in $P_{\pi'}(i \vee j)$, i.e. $l(P_\pi(\bar{i} \wedge j)) \leq l(P_{\pi'}(i \vee j))$.

Therefore,

$$\begin{aligned} C_{\max}(\pi') &= \max\{l(P_{\pi'}(i \vee j), l(P_{\pi'}(\bar{i} \wedge \bar{j}))\} \\ &= \max\{l(P_{\pi'}(i \vee j), l(P_\pi(\bar{i} \wedge \bar{j}))\} \\ &= \max\{l(P_{\pi'}(i \vee j), l(P_\pi(\bar{i} \wedge \bar{j})), l(P_\pi(\bar{i} \wedge j))\} \\ &= \max\{l(P_{\pi'}(i \vee j)), l(P_\pi(\bar{i}))\}, \end{aligned}$$

which provides (4.11). □

Next we will show that the expression (4.11) can be evaluated in an efficient way. For this purpose we introduce some additional notations. For a given solution π and each operation u let $MP_\pi(u)$ and $MS_\pi(u)$ denote the machine predecessor and machine successor of u, respectively. Similarly, $JP(u)$ and $JS(u)$ denote the job predecessor and job successor of u. Additionally, denote by $\hat{r}_u := r_u + p_u$ the sum of the head of u and its processing time in $G(\pi)$ and let $\hat{q}_u := q_u + p_u$. Note that the old machine predecessor $MP_\pi(i)$ of i in π becomes the new machine predecessor $MP_{\pi'}(j)$ of j in π' and symmetrically $MS_{\pi'}(i) = MS_\pi(j)$.

With

$$\Delta := \max\{\hat{r}_{JP(j)}, \hat{r}_{MP_\pi(i)}\} + p_j$$

the first term in the max-expression of (4.11) can be written as

$$\begin{aligned} l(P_{\pi'}(i \vee j)) = \max\{\quad & \Delta + \hat{q}_{JS(j)}, \\ & \max\{\Delta, \hat{r}_{JP(i)}\} + p_i + \max\{\hat{q}_{JS(i)}, \hat{q}_{MS_\pi(j)}\}\}. \end{aligned} \tag{4.12}$$

It is not difficult to see that (4.12) covers all paths containing i or j (see Figure 4.4), which implies that (4.12) is valid.

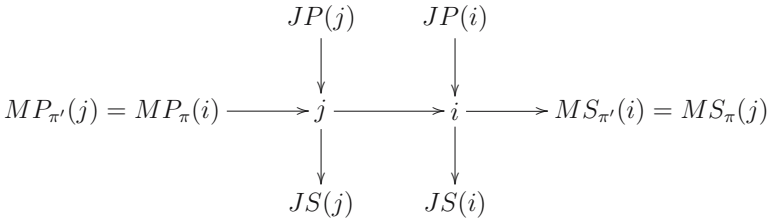


Figure 4.4: Subgraph of $G(\pi')$ containing i and j and their neighbors

Obviously, expression (4.12) can be evaluated in $O(1)$ time if the values $\hat{r}_u$ and $\hat{q}_u$ are known.

A calculation of the second term $l(P_\pi(\bar{i}))$ in (4.11) which is derived from the old graph $G(\pi)$ is slightly more complex but seldom needs to be performed. Indeed, if $l(P_{\pi'}(i \vee j)) \geq C_{\max}(\pi)$, then $C_{\max}(\pi') = l(P_{\pi'}(i \vee j))$ because $C_{\max}(\pi) \geq l(P_\pi(\bar{i}))$. Thus, $l(P_\pi(\bar{i}))$ only has to be calculated if $l(P_{\pi'}(i \vee j)) < C_{\max}(\pi)$.

Since the graph $G(\pi)$ is acyclic for a feasible solution π, all operations $1, \ldots, n$ can be ordered topologically. Let $F_\pi = (F_\pi(1), \ldots, F_\pi(n))$ be a topological ordering for π (i.e. $f_\pi(u) < f_\pi(v)$ for all arcs (u, v) in $G(\pi)$, where $f_\pi(u)$ denotes the position of operation u in F_π). With this list F_π the value $l(P_\pi(\bar{i}))$ can be calculated as follows.

Property 4.2 Let $FJ = \{u \mid JP(u) = 0\}$ be the first operations of the jobs and $LJ = \{u \mid JS(u) = n+1\}$ be the last operations of the jobs. Then we have

$$l(P_\pi(\bar{i})) = \max\{RQ, R', R'', Q', Q''\}, \tag{4.13}$$

where

$$RQ = \max\{\hat{r}_u + \hat{q}_v \mid (u, v) \text{ belongs to } G(\pi) \text{ and } f_\pi(u) < f_\pi(i) < f_\pi(v)\},$$
$$R' = \max\{\hat{r}_u \mid f_\pi(u) < f_\pi(i), u \in LJ\},\ R'' = \max\{\hat{r}_u \mid u \in \{MP_\pi(i), JP(i)\}\},$$
$$Q' = \max\{\hat{q}_u \mid f_\pi(i) < f_\pi(u), u \in FJ\},\ Q'' = \max\{\hat{q}_u \mid u \in \{MS_\pi(i), JS(i)\}\}.$$

Additionally, for any $\pi' = \pi^{(i,j)} \in \mathcal{N}_{ca}(\pi)$ the inequality

$$l(P_{\pi'}(i \vee j)) \geq \max\{R'', Q''\} \tag{4.14}$$

holds.

Proof: Let $w = (0 \to w_1 \to w_2 \to \ldots \to w_z \to n+1)$ be a longest path in $P_\pi(\bar{i})$. Then one of the following three cases holds:

$$\begin{aligned} &f_\pi(w_h) < f_\pi(i) < f_\pi(w_{h+1}) \text{ for some } h \in \{1, \ldots, z-1\} \\ &\text{or } f_\pi(w_z) < f_\pi(i) \text{ or } f_\pi(i) < f_\pi(w_1). \end{aligned} \tag{4.15}$$

In the first case we have $l(w) = RQ$. Furthermore, if $f_\pi(w_z) < f_\pi(i)$, then $w_z \in LJ$ or $w_z \in \{MP_\pi(i), JP(i)\}$. This follows from the fact that if $w_z \notin LJ$ and w_z has a successor different from i and $n+1$, then w cannot be a longest path in $P_\pi(\bar{i})$. Symmetrically, if $f_\pi(i) < f_\pi(w_1)$, then $w_1 \in FJ$ or $w_1 \in \{MS_\pi(i), JS(i)\}$. Thus, in the second case $l(w) = \max\{R', R''\}$ and in the third case $l(w) = \max\{Q', Q''\}$. Therefore, (4.13) is correct.

The inequality (4.14) holds due to the following two inequalities:

- $l(P_{\pi'}(i \vee j)) \geq R'' = \max\{\hat{r}_u \mid u \in \{MP_\pi(i), JP(i)\}\}$
  because $\hat{r}_{JP(i)}$ and $\hat{r}_{MP_\pi(i)}$ are contained in expression (4.12),
- $l(P_{\pi'}(i \vee j)) \geq Q'' = \max\{\hat{q}_u \mid u \in \{MS_\pi(i), JS(i)\}\}$
  because $p_i + \max\{\hat{q}_{JS(i)}, \hat{q}_{MS_\pi(j)}\}$ is contained in (4.12) and j is no job successor of i. □

The preceding discussions imply

Theorem 4.3 The makespan $C_{\max}(\pi')$ for $\pi' = \pi^{(i,j)} \in \mathcal{N}_{ca}(\pi)$ is given by

$$C_{\max}(\pi') = \begin{cases} l(P_{\pi'}(i \vee j)), & \text{if } l(P_{\pi'}(i \vee j)) \geq C_{\max}(\pi) \\ \max\{l(P_{\pi'}(i \vee j)), RQ, R', Q'\}, & \text{otherwise.} \end{cases}$$

If the list F_π and the values $\hat{r}_u$, $\hat{q}_u$ for all operations u are known, the makespan $C_{\max}(\pi')$ can be calculated in $O(\max\{\sum_{\nu=1}^{N} \log n_\nu, \sum_{k=1}^{m} \log m_k\})$ time.

Proof: The formula for $C_{\max}(\pi')$ follows from Properties 4.1 and 4.2 and the discussion on page 198. It remains to prove the complexity. The values $l(P_{\pi'}(i \vee j))$, R' and Q' can be found in time $O(1)$, $O(N)$, and $O(N)$, respectively. To estimate the time for calculating RQ let us assume that job J_ν consists of the operations $O_{1\nu} \to O_{2\nu} \to \ldots \to O_{n_\nu,\nu}$. In the set of conjunctive arcs $O_{\lambda\nu} \to O_{\lambda+1,\nu}$ associated with job J_ν at most one arc (u, v) exists such that $f_\pi(u) < f_\pi(i) < f_\pi(v)$. One can find this arc (or an answer that it does not exist) in time $O(\log n_\nu)$ by means of binary search in the increasing sequence $f_\pi(O_{1\nu}), f_\pi(O_{2\nu}), \ldots, f_\pi(O_{n_\nu,\nu})$. Similarly, one can find in $O(\log m_k)$ time in the sequence $\pi^k = (\pi^k(1), \ldots, \pi^k(m_k))$ associated with machine M_k an index h such that $f_\pi(\pi^k(h)) < f_\pi(i) < f_\pi(\pi^k(h+1))$ (if it exists).

To calculate RQ, one has to evaluate $\hat{r}_u + \hat{q}_v$ for one arc (u, v) in each job chain and each machine sequence and to choose the maximum of all these $(\hat{r}_u + \hat{q}_v)$-values. All these arcs (u, v) can be identified in time $O(\sum_{\nu=1}^{N} \log n_\nu + \sum_{k=1}^{m} \log m_k) = O(\max\{\sum_{\nu=1}^{N} \log n_\nu, \sum_{k=1}^{m} \log m_k\})$, which is also the complexity for calculating the maximum of all $(\hat{r}_u + \hat{q}_v)$-values. □

When moving from a solution π to a neighbor $\pi' = \pi^{(i,j)} \in \mathcal{N}_{ca}(\pi)$ we have to update the topological ordering as well as heads and tails if we want to apply the efficient methods discussed above. Next we describe how these updates can be done in an efficient way:

- **Updating the topological ordering**:

  Let $\nu = f_\pi(i)$ and $\mu = f_\pi(j)$ with $\nu < \mu$ be the positions in F_π of the operations i, j which are to be swapped. To calculate $F_{\pi'}$ for $\pi' = \pi^{(i,j)}$ we split the subsequence $Y : (i = F_\pi(\nu), F_\pi(\nu+1), \ldots, F_\pi(\mu) = j)$ of F_π into the two subsequences

  $$W : (w_1, \ldots, w_r = j) \text{ and } Z : (z_1 = i, z_2, \ldots, z_s) \text{ with } r + s = \mu - \nu + 1$$

  such that W contains all nodes from Y (excluding i) from which a path to node j in $G(\pi)$ exists. Then the list

  $$F_{\pi'} : (F_\pi(1), \ldots, F_\pi(\nu-1), W, Z, F_\pi(\mu+1), \ldots, F_\pi(n))$$

  is a topological ordering of $G(\pi')$ because

- all predecessors of i in $G(\pi')$ are placed before i in $F_{\pi'}$ (analogously all predecessors of j are placed before j in $F_{\pi'}$),
- no vertex z in W is a successor of i in $F_{\pi'}$ because otherwise in $G(\pi)$ there would exist a path from i to j containing z, contradicting the fact that (i,j) is a critical arc in $G(\pi)$.

$F_{\pi'}$ can be computed in time $O(\mu-\nu+1)$ because when we compute backwards from j all predecessors of j which are contained in Y we need to consider only operations contained in Y (the existence of a path from an operation $z \in W$ to j containing an operation from the list $F_\pi(1),\ldots,F_\pi(\nu-1)$ would contradict the fact that F_π is a topological ordering of $G(\pi)$).

The worst case computational time for this procedure is $O(n)$, but on average this procedure is much faster.

- **Updating heads and tails**:

  Updating the values $\hat{r}_u = r_u + p_u$ and $\hat{q}_u = q_u + p_u$ can be done as follows. If the topological ordering $F_{\pi'}$ is given, the new values $\hat{r}'_u$ and $\hat{q}'_u$ in $G(\pi')$ can be calculated according to this ordering in $O(n)$ time using the recursions

$$\begin{aligned} \hat{r}'_u &= \max\{\hat{r}'_v \mid v \in \{MP_{\pi'}(u), JP(u)\}\} + p_u \qquad (4.16)\\ &\quad \text{for } u = F_{\pi'}(1),\ldots,F_{\pi'}(n),\\ \hat{q}'_u &= \max\{\hat{q}'_v \mid v \in \{MS_{\pi'}(u), JS(u)\}\} + p_u \qquad (4.17)\\ &\quad \text{for } u = F_{\pi'}(n),\ldots,F_{\pi'}(1) \end{aligned}$$

  where initially $\hat{r}'_0 = \hat{q}'_{n+1} := 0$.

  This effort can be reduced by the following observations. The $\hat{r}_u$-values for the operations in the initial subsequence $F_\pi(1),\ldots,F_\pi(\nu-1), W\backslash\{j\}$ are identical with the $\hat{r}_u$-values because for $z \in Z \cup \{i\}$ no path to $w \in W$ in $G(\pi')$ exists due to the fact that

  - for $w \in W$ there is a path from w to j in $G(\pi)$ (and also in $G(\pi')$), and
  - $F_{\pi'}$ is a topological ordering.

  Thus, it is sufficient to apply formula (4.16) successively for operations $u = F_{\pi'}(\nu-1+r), F_{\pi'}(\nu+r),\ldots,F_{\pi'}(n)$ and (4.17) for operations $u = F_{\pi'}(\nu+r), F_{\pi'}(\nu+r-1),\ldots,F_{\pi'}(1)$.

Finally, in Figure 4.5 an enhanced tabu search algorithm `ETS` is presented, which uses a tabu search procedure `TS` as a subroutine. It is based on the assumption that tabu search applied to different starting solutions leads to solutions concentrated in some area in which the optimum can be found. We write $(\pi, C) :=$`TS`(ψ) with the meaning that the tabu search procedure `TS` provides a solution π with makespan $C = C_{\max}(\pi)$ when it is applied to the initial solution ψ. The algorithm starts with an initial solution π_0 and creates a set

Start $= \{\pi_1, \ldots, \pi_{max_1}\}$ of starting solutions with a given parameter max_1. These solutions are used to improve the best solution π^* found so far in subsequent steps.

```
Algorithm ETS
 1.  (π_1, C^1) := TS(π_0); C* := C^1;
 2.  FOR λ := 2 TO max_1 DO
 3.      ψ := NIS(π_{λ-1}, π_0, C*);
 4.      (π_λ, C^λ) := TS(ψ);
 5.      IF C^λ < C* THEN
 6.          (π*, C*) := (π_λ, C^λ);
 7.  ENDFOR
 8.  REPEAT
 9.      Find an index μ* ∈ {1, ..., max_1} such that
         D(π*, π_{μ*}) = max_{μ=1}^{max_1} D(π*, π_μ);
10.      ψ := NIS(π*, π_{μ*}, C*); (π, C) := TS(ψ);
11.      IF C < C* THEN
             (π*, C*) := (π, C);
12.  UNTIL max_{μ=1}^{max_1} D(π*, π_μ) < max_2;

Procedure NIS(π_1, π_2, C*)
 1.  π := π_1; iter := 0;
 2.  REPEAT
 3.      iter := iter + 1;
 4.      Let CA be the set of all critical arcs (i, j) in
         G(π) such that j is scheduled before i in π_2;
 5.      IF CA = ∅ THEN
 6.          Let CA be the set of all critical arcs in G(π);
 7.      Determine the best solution π' ∈ N_ca(π) with
         respect to the critical arcs in CA;
 8.      π := π';
 9.      IF C_max(π') < C* THEN
10.          C* := C_max(π'); RETURN π';
11.      ENDIF
12.  UNTIL iter ≥ max_3 · D(π_1, π_2);
13.  RETURN π;
```

Figure 4.5: Enhanced tabu search algorithm

For two feasible solutions π_1, π_2 a "distance measure" $D(\pi_1, \pi_2)$ is introduced, where $D(\pi_1, \pi_2)$ is defined as the number of pairs (u, v) of operations u, v with $\mu(u) = \mu(v)$ such that u is scheduled before v in π_1 and v is scheduled before u in π_2.

In order to improve π^* we consider a solution $\pi_{\mu^*} \in Start$ with maximal distance $D(\pi^*, \pi_{\mu^*})$ and calculate a solution ψ "between" π^* and π_{μ^*}. After applying `TS` to ψ we iterate with the (possibly new) best solution π^*. The whole process is stopped if the maximal distance between the current best solution π^* and solutions in the set $Start$ is smaller than a given threshold max_2.

The procedure `NIS` (π_1, π_2, C^*) called in Step 10 is a new initial solution generator. It is applied to two feasible solutions π_1, π_2, where C^* denotes the best makespan found so far, and returns a new feasible solution π "between" π_1 and π_2.

4.3 Branch-and-Bound Algorithms

In this subsection we briefly describe some foundations concerning branch-and-bound algorithms for the job-shop problem. The proposed approaches can be classified according to the following branching strategies:

- **Generating all active schedules**: An active schedule may be generated by iteratively scheduling a single operation at a certain time point. For a partial schedule the next decision point t is given by the minimal completion time of all scheduled operations. On a machine on which an operation completes at time t all eligible operations (i.e. operations for which all predecessors are finished) are determined. Then in a branching step the partial schedule is extended by scheduling one of these eligible operations at its earliest possible starting time.

- **Branching on disjunctive arcs**: In each branching step a disjunctive arc $i - j \in D$ is oriented either into $i \rightarrow j$ or into $j \rightarrow i$. If all disjunctions are oriented and the corresponding graph is acyclic, feasible schedules are obtained in the leafs of the enumeration tree. This approach may be improved by using efficient constraint propagation techniques which orient disjunctions without branching ("immediate selection"). Furthermore, in more complicated branching schemes several disjunctions are oriented simultaneously.

- **Branching according to blocks**: With each node of the enumeration tree a heuristic solution is associated (which may be obtained by a priority-based heuristic). Then all candidates for an improvement according to the block theorem (Theorem 4.2) are enumerated. After determining for each block the sets of "before-candidates" and "after-candidates" in a branching step a before-candidat is moved to the first position or an after-candidate is moved to the last position of the corresponding block. Additionally, in each branching step some disjunctions are fixed in order to guarantee that the solution sets for the created branches are disjoint. A node of the enumeration tree can be treated as a leaf if the corresponding disjunctive graph is cyclic (then the corresponding solution set is empty) or if the

critical path does not contain any block (then the heuristic solution is optimal for the corresponding solution set).

- **Time-oriented branching on time windows**: With each operation i a time window $[r_i, d_i - p_i]$ for its starting time is associated. Then in a binary branching step the activity i with smallest head r_i is either scheduled at time r_i or it is delayed (meaning that another operation has to be scheduled on the corresponding machine first). In another branching strategy in each iteration the time window $[r_i, d_i - p_i]$ of some operation is split into two disjoint intervals $[r_i, u_i - 1]$ and $[u_i, d_i - p_i]$.

In all branch-and-bound algorithms different lower bounds may be used for the nodes of the enumeration tree. Most of them are based on the calculation of heads and tails for the operations. Longest paths are calculated in the corresponding disjunctive graph taking into account fixed disjunctions. Also machine-based bounds are calculated in which all operations to be processed on the same machine are considered.

4.4 Generalizations

The classical job-shop problem can be generalized in different directions. In this section we will introduce the concepts of time-lags and blocking, in Sections 4.5 to 4.7 more complex applications are studied.

Time-lags

Time-lags are general timing constraints between the starting times of operations (cf. the concept of generalized precedence relations in Section 1.1.1). They can be introduced by the start-start relations

$$S_i + d_{ij} \leq S_j \tag{4.18}$$

where d_{ij} is an arbitrary integer number. As discussed in Section 1.1.1, if $d_{ij} \geq 0$, then (4.18) implies that operation j cannot start earlier than d_{ij} time units after the starting time of operation i. On the other hand, if $d_{ij} < 0$, then operation i cannot start later than $-d_{ij}$ time units after the starting time of operation j. Note, that relations (4.2) are special relations with positive time-lags $d_{ij} := p_i$. Thus, the following disjunctive program is a generalization of (4.1) to (4.4):

$$\begin{aligned}
\min \quad & S_{n+1} && && (4.19)\\
\text{s.t.} \quad & S_i + d_{ij} \leq S_j && (i \to j \in C) && (4.20)\\
& S_i + d_{ij} \leq S_j \vee S_j + d_{ji} \leq S_i && (i - j \in D) && (4.21)\\
& S_i \geq 0 && (i = 0, \ldots, n+1) && (4.22)
\end{aligned}$$

We assume that the restrictions (4.2) are included in the restrictions (4.20) and that (4.21) is at least restrictive as (4.3).

Time-lags d_{ij} may be incorporated into the disjunctive graph $G = (V, C, D)$ by weighting the conjunctive arcs $i \to j \in C$ with the distances d_{ij} and the disjunctive arcs $i - j \in D$ with the pairs (d_{ij}, d_{ji}). If a disjunction $i - j$ is oriented into the direction (i, j), it gets the weight d_{ij}; if it is oriented into the direction (j, i), it gets the weight d_{ji}. In this situation a complete selection $\mathcal{S}$ provides a feasible schedule if and only if the corresponding directed graph $G(\mathcal{S})$ contains no positive cycles.

Time-lags may for example be used to model the following situations:

- **Release times and deadlines**: If r_i is a release time and d_i is a deadline for operation i, then the conditions $S_0 + r_i \leq S_i$ and $S_i + p_i - d_i \leq S_0$ with $S_0 = 0$ must hold. Thus, release times and deadlines can be modeled by the time-lags $d_{0i} := r_i$ and $d_{i0} := p_i - d_i$.

- **Exact relative timing**: If an operation j must start exactly l_{ij} time units after the completion of operation i, we may introduce the relations $S_i + p_i + l_{ij} \leq S_j$ and $S_j - (p_i + l_{ij}) \leq S_i$. A no-wait constraint between operations i and j can be modeled by setting $l_{ij} := 0$.

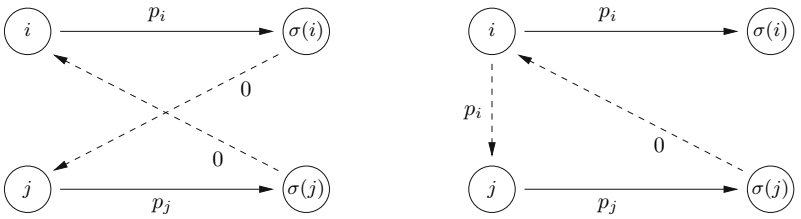
- **Setup times**: If a setup time is needed between the completion of an operation and the beginning of another operation on the same machine, we may introduce the constraints $S_i + p_i + s_{ij} \leq S_j \vee S_j + p_j + s_{ji} \leq S_i$, where s_{ij} denotes the setup time between operations i and j.

- **Machine unavailabilities**: If a machine M_k is unavailable within time intervals $]a_\lambda, b_\lambda[$ for $\lambda = 1, \ldots, v$, we may introduce v artificial operations $\lambda = 1, \ldots, v$ with $\mu(\lambda) = M_k$, processing times $p_\lambda := b_\lambda - a_\lambda$, release times $r_\lambda := a_\lambda$ and deadlines $d_\lambda := b_\lambda$ which ensure that the intervals $]a_\lambda, b_\lambda[$ are blocked.

- **Maximum lateness objective**: As indicated on page 11, problems with maximum lateness objective $L_{\max} = \max_{j=1}^{N}\{C_j - d_j\}$ can be reduced to $C_{\max}$-problems by setting $d_{l(j),n+1} = p_{l(j)} - d_j$ for each job j where $l(j)$ denotes the last operation of job j.

Blocking

In the following we will generalize the job-shop problem by additionally considering **blocking operations**. This situation occurs if a job has finished on a machine M_k and the next machine is still occupied by another job. If a finished job cannot be stored outside the machine (in a so-called buffer), then it remains on M_k and blocks it until the next machine becomes available. The corresponding operation of this job is called a **blocking operation**. A **blocking**



(a) blocking operations i and j (b) blocking operation j

Figure 4.6: A pair of alternative arcs

job-shop is a job-shop problem in which all operations are blocking operations. More formally, an operation i is blocking if no buffer exists to store $J(i)$ when $\mu(\sigma(i))$ is occupied by another job. Obviously, blocking operations may delay the start of succeeding operations on the same machine.

Consider two blocking operations i and j which have to be processed on the same machine $\mu(i) = \mu(j)$. If operation i precedes operation j, the successor operation $\sigma(i)$ of operation i must start before operation j in order to unblock the machine, i.e. we must have $S_{\sigma(i)} \leq S_j$. Conversely, if operation j precedes operation i, then operation $\sigma(j)$ must start before operation i, i.e. we must have $S_{\sigma(j)} \leq S_i$. Thus, there are two mutually exclusive (alternative) relations

$$S_{\sigma(i)} \leq S_j \qquad \text{or} \qquad S_{\sigma(j)} \leq S_i$$

in connection with i and j. These two mutually exclusive relations can be modeled by a **pair of alternative arcs** $(\sigma(i), j)$ and $(\sigma(j), i)$ with arc weights $d_{\sigma(i),j} = d_{\sigma(j),i} = 0$ as shown in Figure 4.6(a). The pair of alternative arcs is depicted by dashed lines, whereas the solid lines represent the job precedences $(i, \sigma(i))$ and $(j, \sigma(j))$ with arc weights $d_{i,\sigma(i)} = p_i$ and $d_{j,\sigma(j)} = p_j$.

In order to get a feasible schedule, exactly one of the two alternatives has to be chosen (selected). Choosing the arc $(\sigma(i), j)$ implies that operation i has to leave the machine before j can start and choosing $(\sigma(j), i)$ implies that j has to leave the machine before i can start.

Note that choosing $(\sigma(i), j)$ implies that i must precede j by transitivity, and choosing $(\sigma(j), i)$ implies that j must precede i. Thus, if blocking arcs $(\sigma(i), j)$, $(\sigma(j), i)$ are introduced, we do not have to add disjunctive arcs $(i, j), (j, i)$ between operations which are processed on the same machine $\mu(i) = \mu(j)$.

Next, we consider the case where operation i is non-blocking, operation j is blocking and both have to be scheduled on the same machine $\mu(i) = \mu(j)$. If operation i precedes operation j, machine $\mu(i)$ is not blocked after the processing of i. Thus, operation j can start as soon as operation i is finished, i.e. we have $S_i + p_i \leq S_j$. On the other hand, if operation j precedes operation i, then operation $\sigma(j)$ must start before operation i, i.e. we have $S_{\sigma(j)} \leq S_i$.

Thus, we have the alternative relations

$$S_i + p_i \leq S_j \qquad \text{or} \qquad S_{\sigma(j)} \leq S_i$$

in connection with i and j. Figure 4.6(b) shows the corresponding pair of alternative arcs (i,j) and $(\sigma(j),i)$ weighted with $d_{ij} = p_i$ and $d_{\sigma(j),i} = 0$, respectively.

Finally, we consider two non-blocking operations i and j which have to be processed on the same machine. This leads to the alternative relations

$$S_i + p_i \leq S_j \qquad \text{or} \qquad S_j + p_j \leq S_i.$$

These relations can be represented by the alternative arcs (i,j) and (j,i) weighted with $d_{ij} = p_i$ and $d_{ji} = p_j$, respectively. This pair of alternative arcs corresponds to a disjunction between operation i and operation j in the classical disjunctive graph model. Choosing one of the two alternative arcs (i,j) or (j,i) is equivalent to fixing a direction for the disjunction between i and j.

In the special case when operation i is the last operation of job $J(i)$, machine $\mu(i)$ is not blocked after the processing of i. Thus, in this case, operation i is always assumed to be non-blocking.

More generally, a set of mutually exclusive pairs of relations

$$(S_i + d_{ij} \leq S_j) \text{ or } (S_f + d_{fg} \leq S_g) \quad \{(i,j),(f,g)\} \in A \tag{4.23}$$

with pairs of alternative arcs $\{(i,j),(f,g)\} \in A$ may be introduced. Then the graph $G = (V, C, A)$ which corresponds to problem (4.19), (4.20), (4.22), and (4.23) is called **alternative graph**.

Similarly as in the disjunctive graph model we have to determine a selection $\mathcal{S}$ which contains exactly one arc (i,j) or (f,g) for each pair of alternative arcs in A such that the resulting graph $G(\mathcal{S}) = (V, C \cup \mathcal{S})$ with arc weights d does not contain a positive cycle. Note that the graph $G(\mathcal{S})$ may contain cycles of length 0. In this case, all operations included in such a cycle start processing at the same time.

Given a complete selection $\mathcal{S}$, we may calculate a corresponding earliest start schedule S by longest path calculations as in the disjunctive graph model.

Example 4.2: Consider a job-shop problem with $m = 3$ machines and $N = 3$ jobs where jobs $j = 1, 2$ consist of $n_j = 3$ operations and job 3 contains $n_3 = 2$ operations. The first two operations of jobs 1 and 2 are assumed to be blocking, all other operations are non-blocking. Jobs 1 and 2 have to be processed first on M_1, then on M_2 and last on M_3, whereas job 3 has to be processed first on M_2 and next on M_1. Furthermore, we have the processing times $p_{11} = 3$, $p_{21} = 2$, $p_{31} = 1$, $p_{12} = 1$, $p_{22} = 2$, $p_{32} = 2$, $p_{13} = 6$, $p_{23} = 2$.

In Figure 4.7 the alternative graph $G = (V, C, A)$ for this instance is shown (without the dummy nodes $0, n+1$). The job chains for each job are shown horizontally. Black circles represent blocking operations whereas white circles

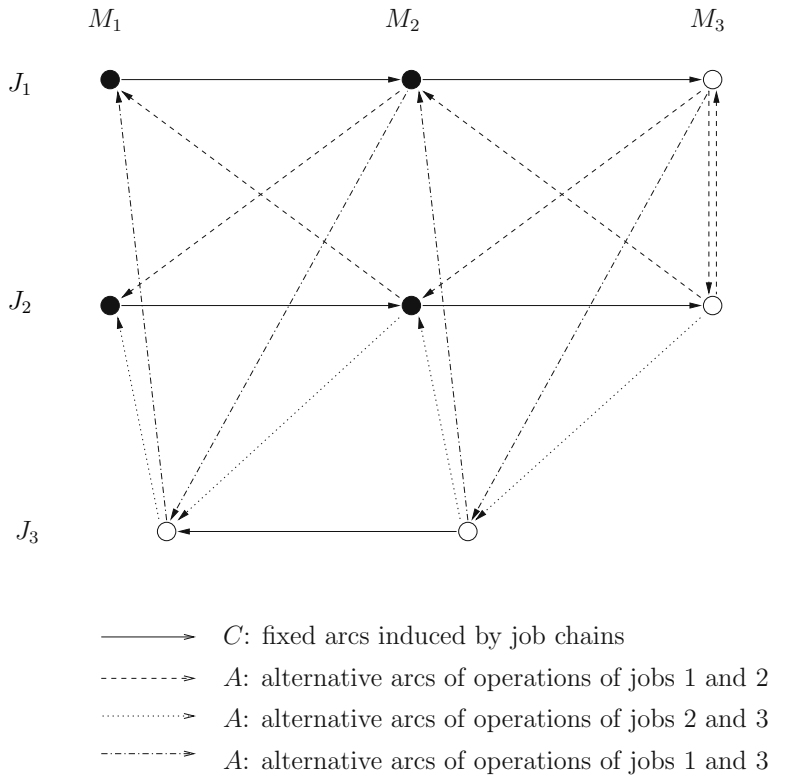


Figure 4.7: Alternative graph $G = (V, C, A)$

represent non-blocking operations. In order to distinguish different pairs of alternative arcs, alternative arcs induced by operations of the same two jobs are depicted in the same line pattern.

Assume that the operations are scheduled in the orders $\pi^1 = (O_{11}, O_{12}, O_{23})$ on M_1, $\pi^2 = (O_{13}, O_{21}, O_{22})$ on M_2 and $\pi^3 = (O_{31}, O_{32})$ on M_3. These machine sequences induce a complete selection $\mathcal{S}$, for which the corresponding graph $G(\mathcal{S})$ is shown in Figure 4.8. By longest path calculations in $G(\mathcal{S})$ the corresponding earliest start schedule S of Figure 4.9 with $C_{\max}(S) = 12$ is calculated. Note that job 2 cannot start on M_1 before time 6 because job 1 blocks machine M_1 from time 3 to time 6. Similarly, job 2 blocks M_1 from time 7 to time 8. □

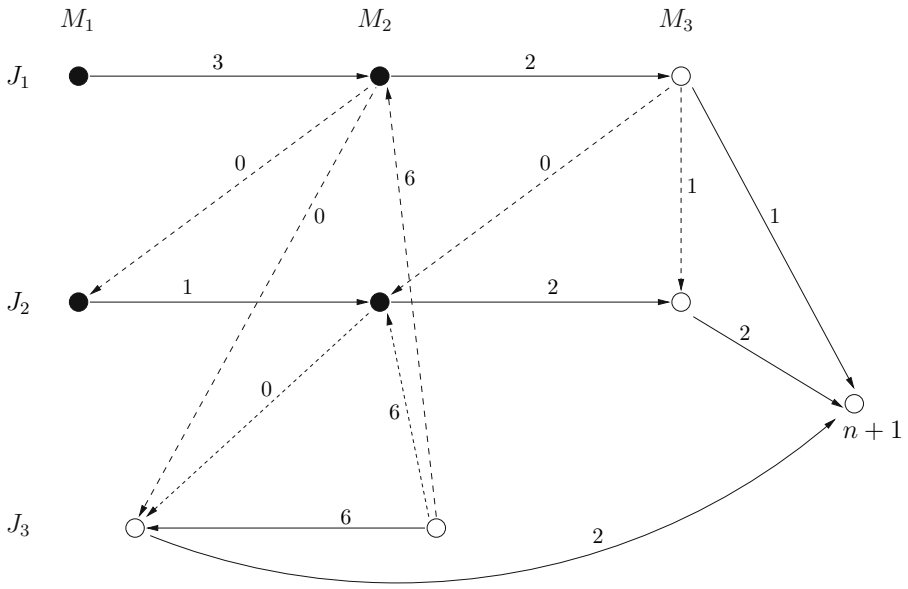


Figure 4.8: The graph $G(\mathcal{S}) = (V, C \cup \mathcal{S})$

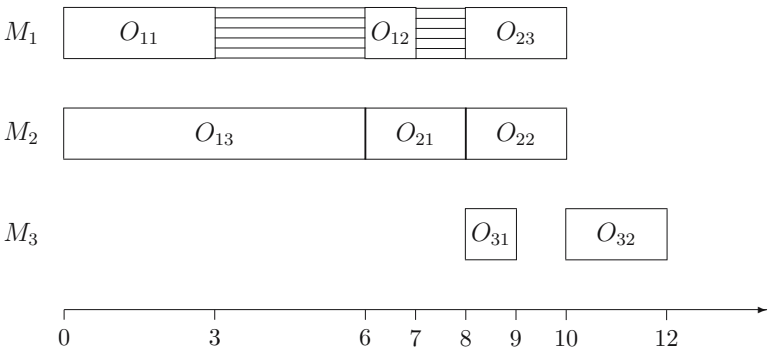


Figure 4.9: A corresponding schedule S

4.5 Job-Shop Problems with Flexible Machines

In this section we study job-shop problems with **flexible machines** (multi-purpose machines), also called **flexible job-shops**. After giving a precise formulation of the problem in Subsection 4.5.1, in Subsection 4.5.2 we describe foundations for heuristic methods which are based on the disjunctive graph model.

4.5.1 Problem formulation

In the following we will generalize the job-shop problem by **flexible machines** (multi-purpose machines). Assume that with each operation i a set of machine indices $\mathcal{M}(i) \subseteq \{1, \ldots, m\}$ is associated indicating that i can be executed by any machine of this set. If operation i is executed on machine $k \in \mathcal{M}(i)$, then its processing time is equal to p_{ik} (or simply to p_i if the processing time does not depend on the assigned machine). One has to assign to each operation i a machine from the machine set $\mathcal{M}(i)$ in such a way that a corresponding optimal makespan is minimal over all possible assignments.

A model with flexible machines may for example be used when the operations need certain tools for processing and the machines are only equipped with a few of them. Then $\mathcal{M}(i)$ denotes all machines which are equipped with the tools needed by operation i.

To give a mathematical description of this problem, we introduce assignment variables $x_{ik} \in \{0, 1\}$, where

$$x_{ik} = \begin{cases} 1, & \text{if operation } i \text{ is assigned to machine } k \in \mathcal{M}(i) \\ 0, & \text{otherwise.} \end{cases}$$

For each assignment $x = (x_{ik})$, we define disjunctions

$$D(x) = \{i - j \mid x_{ik} = x_{jk} = 1 \text{ for some } k \in \mathcal{M}(i) \cap \mathcal{M}(j)\}$$

between operations i, j assigned to the same machine. Then the problem can be formulated as follows:

$$\min \quad S_{n+1} \tag{4.24}$$

$$\text{s.t.} \quad S_i + \sum_{k \in \mathcal{M}(i)} p_{ik} x_{ik} \leq S_j \qquad (i \to j \in C) \tag{4.25}$$

$$S_i + \sum_{k \in \mathcal{M}(i)} p_{ik} x_{ik} \leq S_j \vee S_j + \sum_{k \in \mathcal{M}(j)} p_{jk} x_{jk} \leq S_i \qquad (i - j \in D(x)) \tag{4.26}$$

$$\sum_{k \in \mathcal{M}(i)} x_{ik} = 1 \qquad (i = 1, \ldots, n) \tag{4.27}$$

$$S_i \geq 0 \qquad (i = 0, \ldots, n+1) \tag{4.28}$$

$$x_{ik} \in \{0, 1\} \quad (i = 1, \ldots, n; k \in \mathcal{M}(i)) \tag{4.29}$$

Given a machine assignment $x = (x_{ik})$ satisfying (4.27) and (4.29), the problem reduces to problem (4.1) to (4.4) with operation processing times $p_i = \sum_{k \in \mathcal{M}(i)} p_{ik} x_{ik}$ and disjunctions $D = D(x)$. Thus, the problem may be solved with a two-stage approach where at first machines are assigned to the operations and afterwards the resulting classical job-shop problem is solved.

4.5.2 Heuristic methods

In this subsection we discuss heuristic methods for the flexible job-shop problem. Especially, we describe foundations for neighborhoods which are based on the disjunctive graph model.

In the following we assume that we are given a machine assignment μ (where $\mu(i) \in \mathcal{M}(i)$ denotes the machine assigned to operation i) and a corresponding complete consistent selection $\mathcal{S}$ with the associated earliest start schedule S. We want to determine a neighbor solution $(\mu', \mathcal{S}')$ of $(\mu, \mathcal{S})$ which is feasible and satisfies $C_{\max}(S') < C_{\max}(S)$.

The following theorem is a straightforward generalization of Theorem 4.2:

Theorem 4.4 Let $(\mu, \mathcal{S})$ be a feasible solution with makespan $C_{\max}(S)$ and let P^S be a critical path in $G(\mathcal{S})$. If another feasible solution $(\mu', \mathcal{S}')$ with $C_{\max}(S') < C_{\max}(S)$ exists, then

- in μ' at least one critical operation on P^S has to be assigned to another machine, or
- in $\mathcal{S}'$ at least one operation of some block B on P^S has to be processed before the first or after the last operation of B.

Based on this theorem the block shift neighborhoods $\mathcal{N}^1_{bs}$ and $\mathcal{N}^2_{bs}$ for the classical job-shop problem as defined in Section 4.2 may be enlarged by additionally allowing that a critical operation is processed at some position on another machine. Again it can be shown that the enlarged neighborhood $\mathcal{N}^2_{bs}$ is opt-connected.

A disadvantage of these neighborhoods is that they may be quite large since each critical operation may be moved to a large number of positions on other machines. In the following we will reduce the number of moves by allowing only feasible moves (i.e. the resulting disjunctive graph is acyclic) and trying to calculate the best insertion position of an operation which is assigned to another machine (i.e. all other insertions of this operation on the same machine do not lead to schedules with a better makespan).

In the following let $\pi^1, \ldots, \pi^m$ be the machine sequences corresponding to the solution $(\mu, \mathcal{S})$. We consider a fixed operation v and a fixed machine index $k \in \{1, \ldots, m\}$. We want to remove v from its machine sequence $\pi^{\mu(v)}$ and insert it into the machine sequence π^k on M_k (where also $k = \mu(v)$ is allowed).

Such a k-insertion is called feasible if the resulting disjunctive graph is acyclic. A feasible k-insertion is called an **optimal k-insertion** if the makespan of the resulting schedule is not larger than the makespan of any other schedule which is obtained by a feasible k-insertion of v. In the remainder of this section we will define a reduced neighborhood which always contains a solution corresponding to an optimal k-insertion

Denote by G^- the disjunctive graph which is obtained from $G(\mathcal{S})$ by deleting operation v from its current machine sequence $\pi^{\mu(v)}$ (i.e. by removing all fixed disjunctive machine arcs containing v and setting the weight of vertex v to 0). For each operation i denote by r_i^- its head and by q_i^- its tail obtained by longest path calculations in G^-. Since in G^- operation v is removed, we have $r_i^- \leq r_i$ and $q_i^- \leq q_i$ for all operations i.

Denote by $JS(v)$ again the job successor of v in $J(v)$ and by $MS(v)$ its machine successor in $G(\mathcal{S})$. Since in G^- the arc $(v, MS(v))$ is removed and in $G(\mathcal{S})$ no path from $JS(v)$ to $n+1$ containing the arc $(v, MS(v))$ exists (otherwise the arc $(v, JS(v))$ would create a cycle in G), the tail of $JS(v)$ is not changed. Thus, $q_v^- = q_{JS(v)}^- + p_{JS(v)} = q_{JS(v)} + p_{JS(v)}$. Symmetrically, $r_v^- = r_{JP(v)}^- + p_{JP(v)} = r_{JP(v)} + p_{JP(v)}$, where $JP(v)$ denotes the job predecessor of v in $J(v)$.

Let R^k, L^k be subsequences from π^k with

$$R^k := \{i \in \pi^k \mid r_i + p_i > r_v^-\}, \quad L^k := \{i \in \pi^k \mid p_i + q_i > q_v^-\}.$$

For each $i \in \pi^k \setminus R^k$ and each $j \in R^k$ we have $r_i + p_i \leq r_v^- < r_j + p_j$, i.e. operation i is completed before operation j. Symmetrically, for each $j \in L^k$ and each $h \in \pi^k \setminus L^k$ we have $p_j + q_j > q_v^- \geq p_h + q_h$, i.e. operation j is started (and thus completed) before operation h. Hence, we may conclude that each operation $i \in L^k \setminus R^k$ is completed before each $j \in L^k \cap R^k$ and each $j \in L^k \cap R^k$ is completed before each $h \in R^k \setminus L^k$ (cf. Figure 4.10).

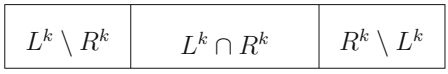


Figure 4.10: Position of the sets $L^k \setminus R^k$, $L^k \cap R^k$ and $R^k \setminus L^k$

Therefore, for $k = 1, \ldots, m$ we may define N^{vk} as the set of all solutions which are obtained from $(\mu, \mathcal{S})$ by inserting operation v into the machine sequence π^k after all operations from $L^k \setminus R^k$ and before all operations from $R^k \setminus L^k$.

Theorem 4.5 All solutions in the set N^{vk} are feasible and N^{vk} contains an optimal k-insertion of v. Furthermore, if $L^k \cap R^k = \emptyset$ holds, all solutions in N^{vk} are optimal k-insertions.

Proof: At first we will show that all solutions in N^{vk} are feasible, i.e. by the corresponding insertions of v into the machine sequence π^k no cycle is created.

Firstly, we show that for all $i \in R^k$ no path from i to v in G^- exists. If for $i \in R^k$ the condition $r_i^- = r_i$ holds, then $r_i^- + p_i = r_i + p_i > r_v^-$ is valid, which

implies that in G^- no path from i to v exists. If for $i \in R^k$ the condition $r_i^- < r_i$ holds, then v must be a predecessor of i in G, i.e. in G (and hence also in G^-) no path from i to v exists since G is acyclic.

For all $i \in \pi^k \setminus R^k$ we have $r_i + p_i \leq r_v^- \leq r_v$ implying $r_i < r_v + p_v$ due to $p_v > 0$. Thus, in G (and hence also in G^-) no path exists from operation v to an operation $i \in \pi^k \setminus R^k$.

Symmetrically it can be shown that in G^- neither a path from v to an operation in L^k nor a path from an operation in $\pi^k \setminus L^k$ to v exists. Not existing paths between v and other vertices in G^- are summarized in Figure 4.11.

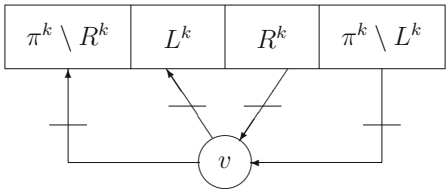


Figure 4.11: Not existing paths between v and other vertices in G^-

We conclude that for $i \in L^k \setminus R^k$ and $j \in R^k \setminus L^k$ neither a path from v to i nor a path from j to v in G^- exists. For all other operations

$$u \in \pi^k \setminus ((L^k \setminus R^k) \cup (R^k \setminus L^k)) = ((\pi^k \setminus L^k) \cap (\pi^k \setminus R^k)) \cup (L^k \cap R^k)$$

neither a path from u to v nor a path from v to u exists (cf. Figure 4.11). Thus, every k-insertion of v after all operations from $L^k \setminus R^k$ and before all operations from $R^k \setminus L^k$ does not create a cycle.

Now we will show that N^{vk} contains an optimal k-insertion of v. For successive operations $u, w \in \pi^k$ we consider the insertion of operation v between u and w. Let $G^{(u,w)}$ be the corresponding disjunctive graph which is obtained from G^- by inserting the machine arcs (u, v), (v, w) and setting the weight of vertex v to p_{vk}. Since all other arcs and weights in G^- and $G^{(u,w)}$ are the same, the makespan of the new solution is given by the maximum of the length of a longest path in G^- and the length of a longest path in $G^{(u,w)}$ containing operation v.

If we denote for each operation i its head in $G^{(u,w)}$ by $r_i^{(u,w)}$ and its tail by $q_i^{(u,w)}$, then we have

$$C_{\max}^{(u,w)} = \max\{C_{\max}^-, r_v^{(u,w)} + p_{vk} + q_v^{(u,w)}\} \tag{4.30}$$

In order to determine an optimal k-insertion of v, we have to find successive operations $u, w \in \pi^k$ such that ${C_{\max}}^{(u,w)}$ is minimal. Since $C_{\max}^-$ is constant for each insertion of v, we have to determine u, w such that the second term in (4.30) is minimal. Due to $r_v^{(u,w)} = \max\{r_u^- + p_u, r_v^-\}$ and $q_v^{(u,w)} = \max\{p_w + q_w^-, q_v^-\}$

we have

$$\begin{aligned} r_v^{(u,w)} + p_{vk} + q_v^{(u,w)} &= \max\{r_u^- + p_u, r_v^-\} + p_{vk} + \max\{p_w + q_w^-, q_v^-\} \\ &= \max\{r_u^- + p_u - r_v^-, 0\} + r_v^- + p_{vk} + q_v^- \\ &\qquad + \max\{p_w + q_w^- - q_v^-, 0\} \\ &= r_v^- + p_{vk} + q_v^- + \Delta(u,w) \end{aligned}$$

with

$$\Delta(u,w) := \max\{r_u^- + p_u - r_v^-, 0\} + \max\{p_w + q_w^- - q_v^-, 0\}.$$

Thus, an optimal k-insertion of v corresponds to an insertion (u,w) with minimal $\Delta(u,w)$-value. In order to show that N^{vk} contains such an insertion, recall that N^{vk} contains all solutions in which v is inserted after all operations from $L^k \setminus R^k$ and before all operations from $R^k \setminus L^k$. We will show that

(i) all insertions of v into the sequence π^k before the last operation from $L^k \setminus R^k$ are not better than the insertion of v as a direct successor of the last operation from $L^k \setminus R^k$,

(ii) all insertions of v into the sequence π^k after the first operation from $R^k \setminus L^k$ are not better than the insertion of v as a direct predecessor of the first operation from $R^k \setminus L^k$.

(i) For all $u \in L^k \setminus R^k$ we have $r_u + p_u \leq r_v^-$ implying $\max\{r_u^- + p_u - r_v^-, 0\} = 0$ due to $r_u^- \leq r_u$. Thus, for each $u \in L^k \setminus R^k$ and its successor w in π^k we get $\Delta(u,w) = \max\{p_w + q_w^- - q_v^-, 0\}$.

Assume that $L^k \setminus R^k = (u_1, \ldots, u_{z-1})$ with $z \geq 2$. Let u_z be the direct successor of u_{z-1} in π^k (or $u_z := n+1$ if no such element exists) and let $u_0 := 0$. Because operation u_λ is sequenced before operation $u_{\lambda+1}$ in π^k, we have $q_{u_\lambda}^- \geq p_{u_{\lambda+1}} + q_{u_{\lambda+1}}^-$ for $\lambda = 0, \ldots, z-1$. This implies $\Delta(u_{\lambda-1}, u_\lambda) = \max\{p_{u_\lambda} + q_{u_\lambda}^- - q_v^-, 0\} \geq \max\{p_{u_{\lambda+1}} + q_{u_{\lambda+1}}^- - q_v^-, 0\} = \Delta(u_\lambda, u_{\lambda+1})$ for $\lambda = 1, \ldots, z-1$.

Thus, among the insertion positions $\lambda = 1, \ldots, z$ the value $\Delta(u_{\lambda-1}, u_\lambda)$ is minimal for $\lambda = z$. Hence, all insertions of v into π^k before the last operation $u_{z-1} \in L^k \setminus R^k$ do not lead to a smaller makespan than the insertion of v as a direct successor of the last operation from $L^k \setminus R^k$.

(ii) can be proved symmetrically.

Finally, we consider the situation $L^k \cap R^k = \emptyset$, i.e. $L^k \setminus R^k = L^k$ and $R^k \setminus L^k = R^k$. For each operation $i \in \pi^k \setminus (L^k \cup R^k)$ we have $r_i + p_i \leq r_v^-$ and $p_i + q_i \leq q_v^-$ implying $\max\{r_i^- + p_i - r_v^-, 0\} = \max\{p_i + q_i^- - q_v^-, 0\} = 0$ due to $r_i^- \leq r_i$ and $q_i^- \leq q_i$.

For each operation $u \in L^k = L^k \setminus R^k$ we get $\max\{r_u^- + p_u - r_v^-, 0\} = 0$ and for each operation $w \in R^k = R^k \setminus L^k$ we get $\max\{p_w + q_w^- - q_v^-, 0\} = 0$. Thus, $\Delta(u,w) = 0$ holds for all successive operations $u, w \in \pi^k$. Since all Δ-values are non-negative,

all insertions of v after all operations from L^k and before all operations from R^k (which have been shown to be feasible) are optimal k-insertions. □

Based on the previous theorem the reduced neighborhood $\mathcal{N}^{red}$ may be defined by considering all possible k-insertions of critical operations $v \in P^S$ from the sets N^{vk}. For each critical operation v and each machine sequence π^k the subsequences L^k and R^k can be determined in $O(\log|\pi^k|)$ time by binary search (for R^k it is sufficient to find the first operation $i \in \pi^k$ with $r_i + p_i > r_v^-$ and for L^k it is sufficient to find the last operation $i \in \pi^k$ with $p_i + q_i > q_v^-$).

Thus, the neighbors in $\mathcal{N}^{red}(\mu, \mathcal{S}) = \bigcup_{v \in P^S} \bigcup_{k=1}^{m} N^{vk}$ can be computed in $O(n \log n)$ time. To determine the best neighbor for a given solution, the makespan has to be computed for each solution in the sets N^{vk} (which can be done in $O(n)$ time for each neighbor by longest path calculations). Since this may be very time consuming, an attempt to avoid these calculations has been introduced. Instead of calculating the exact makespan after inserting v, an upper bound value is determined for the length of a longest path containing operation v.

If $L^k \cap R^k = \emptyset$, then we have $\Delta(u, w) = 0$ for all successive operations $u, w \in \pi^k$, i.e. the exact length of a longest path containing v is given by $r_v^- + p_{vk} + q_v^-$.

If $L^k \cap R^k \neq \emptyset$, assume that $L^k \cap R^k = (u_1, \ldots, u_{z-1})$ with $z \geq 2$. Let u_z be the first element in $R^k \setminus L^k$ (or $u_z := n+1$ if no such element exists) and let u_0 be the last element in $L^k \setminus R^k$ (or $u_0 := 0$ if no such element exists).

When v is inserted between $u_{\lambda-1}$ and u_λ for a position $\lambda \in \{1, \ldots, z\}$, then $\Delta(u_{\lambda-1}, u_\lambda)$ equals

$$\begin{cases} \max\{p_{u_1} + q_{u_1}^- - q_v^-, 0\}, & \lambda = 1 \\ \max\{r_{u_{\lambda-1}}^- + p_{u_{\lambda-1}} - r_v^-, 0\} + \max\{p_{u_\lambda} + q_{u_\lambda}^- - q_v^-, 0\}, & 1 < \lambda < z \\ \max\{r_{u_{z-1}}^- + p_{u_{z-1}} - r_v^-, 0\}, & \lambda = z \end{cases}$$

since $u_0 \in L^k$ and $u_z \in R^k$.

Thus, in order to calculate these values, the heads $r_{u_\lambda}^-$ and the tails $q_{u_\lambda}^-$ in G^- have to be determined. If we replace the values $r_{u_\lambda}^-, q_{u_\lambda}^-$ by the possibly larger values $r_{u_\lambda}, q_{u_\lambda}$, then due to $u_1, \ldots, u_{z-1} \in L^k \cap R^k$ we obtain

$$\tilde{\Delta}(u_{\lambda-1}, u_\lambda) = \begin{cases} p_{u_1} + q_{u_1} - q_v^-, & \lambda = 1 \\ r_{u_{\lambda-1}} + p_{u_{\lambda-1}} - r_v^- + p_{u_\lambda} + q_{u_\lambda} - q_v^-, & 1 < \lambda < z \\ r_{u_{z-1}} + p_{u_{z-1}} - r_v^-, & \lambda = z. \end{cases}$$

Since $r_{u_\lambda}^- \leq r_{u_\lambda}$ and $q_{u_\lambda}^- \leq q_{u_\lambda}$, the $\tilde{\Delta}(u_{\lambda-1}, u_\lambda)$-values are upper bound values for the exact $\Delta(u_{\lambda-1}, u_\lambda)$-values. Furthermore, these values can be computed in constant time, i.e. using them each neighbor can be evaluated in $O(1)$ instead of $O(n)$ time.

In the case that the neighborhood $\mathcal{N}^{red}(\mu, \mathcal{S})$ of a solution $(\mu, \mathcal{S})$ is empty, again it can be shown that the solution is optimal. If $\mathcal{N}^{red}(\mu, \mathcal{S}) = \emptyset$ holds, then we must have the situation that each critical operation $v \in P^S$ can only

be processed by one machine (i.e. $|\mathcal{M}(v)| = 1$) since otherwise a move of v to another machine would be possible. Furthermore, for all critical operations v and $k = \mu(v)$ we must have $L^k \cap R^k = \emptyset$ since otherwise at least two insertion positions for v would be available.

We will show that for each critical operation v also its job predecessor $JP(v)$ and its job successor $JS(v)$ are critical in G. Assume to the contrary that $JS(v)$ is not critical. Since v is critical, then its machine successor $MS(v)$ must be critical, i.e.

$$q_v = p_{MS(v)} + q_{MS(v)} > p_{JS(v)} + q_{JS(v)} = q_v^-.$$

For $MS(v)$ we have $r_{MS(v)} \geq r_v + p_v$ implying $r_{MS(v)} + p_{MS(v)} > r_v \geq r_v^-$, i.e. $MS(v) \in R^k$. Together with $L^k \cap R^k = \emptyset$ this implies $MS(v) \notin L^k$, i.e. $p_{MS(v)} + q_{MS(v)} \leq q_v^-$, which is a contradiction. Thus, also $JS(v)$ must be critical. In the same way it can be proved that the job predecessor $JP(v)$ must be critical.

Thus, a critical path exists containing only conjunctive arcs. But then as for the neighborhoods $\mathcal{N}_{ca}$ and $\mathcal{N}_{bs}^2$, the makespan equals the total processing time of a job, which is a lower bound value for the optimal makespan.

Theorem 4.6 The neighborhood $\mathcal{N}^{red}$ is opt-connected.

Proof: In order to show that $\mathcal{N}^{red}$ is opt-connected, consider an arbitrary non-optimal solution $(\mu, \mathcal{S})$ and an optimal solution $(\mu^*, \mathcal{S}^*)$. We have to show that we can transform $(\mu, \mathcal{S})$ into $(\mu^*, \mathcal{S}^*)$ or into another optimal solution by a finite number of steps in the neighborhood $\mathcal{N}^{red}$. Let $\alpha(\mu, \mu^*)$ be the number of operations which are assigned to different machines in μ and μ^* and let $\beta(\mathcal{S}, \mathcal{S}^*)$ be the number of disjunctive arcs which belong to both $\mathcal{S}$ and $\mathcal{S}^*$, but are oriented into different directions in these two selections (we do not count arcs which are missing in one of the two selections due to the fact that the corresponding operations are assigned to different machines).

In the following we will show that it is possible to construct a sequence $(\mu, \mathcal{S}) = (\mu_0, \mathcal{S}_0), (\mu_1, \mathcal{S}_1), \ldots, (\mu_z, \mathcal{S}_z)$ such that

- the solution $(\mu_z, \mathcal{S}_z)$ is optimal,
- $(\mu_\lambda, \mathcal{S}_\lambda) \in \mathcal{N}^{red}(\mu_{\lambda-1}, \mathcal{S}_{\lambda-1})$ for $\lambda = 1, \ldots, z$,
- operations which are assigned to the same machine in assignment μ_λ and the optimal assignment μ^* are not assigned to other machines in subsequent assignments $\mu_{\lambda+1}, \ldots, \mu_z$, which implies $\alpha(\mu_{\lambda+1}, \mu^*) \leq \alpha(\mu_\lambda, \mu^*)$ for $\lambda = 1, \ldots, z$,
- if $\alpha(\mu_{\lambda+1}, \mu^*) = \alpha(\mu_\lambda, \mu^*)$ for an index λ holds, then we have $\beta(\mathcal{S}_{\lambda+1}, \mathcal{S}^*) < \beta(\mathcal{S}_\lambda, \mathcal{S}^*)$, i.e. if the number of operations which are assigned to different machines in μ_λ and μ^* is not decreased after iteration λ, then the number of disjunctive arcs oriented into different directions is decreased.

The last two properties imply that the pair $(\alpha(\mu_\lambda, \mu^*), \beta(\mathcal{S}_\lambda, \mathcal{S}^*))$ is lexicographically decreasing in each iteration, i.e. the procedure will stop after a finite number of iterations.

Assume that solutions $(\mu_1, \mathcal{S}_1), \ldots, (\mu_\lambda, \mathcal{S}_\lambda)$ satisfying the above properties have been constructed up to some index λ. If $(\mu_\lambda, \mathcal{S}_\lambda)$ is optimal, we are done. Otherwise, in order to improve the current solution according to Theorem 4.4 a critical operation has to be assigned to another machine or an operation belonging to a block has to be moved to the beginning or the end of the block. Let $P^{\mathcal{S}_\lambda}$ be a critical path in $G(\mathcal{S}_\lambda)$ which is constructed as follows: We start with a first operation i on an arbitrary critical path. If the job successor $JS(i)$ is critical, this operation is added to $P^{\mathcal{S}_\lambda}$, otherwise the machine successor $MS(i)$ is added. This procedure is repeated until the dummy operation $n+1$ is reached.

For the resulting critical path $P^{\mathcal{S}_\lambda}$ we distinguish two cases: If a critical operation i on $P^{\mathcal{S}_\lambda}$ with $\mu_\lambda(i) \neq \mu^*(i)$ exists (i.e. i is assigned to the "wrong" machine compared with μ^*), then let $(\mu_{\lambda+1}, \mathcal{S}_{\lambda+1})$ be an arbitrary solution in the set $N^{i\mu^*(i)} \subset \mathcal{N}^{red}(\mu_\lambda, \mathcal{S}_\lambda)$ (in which i is removed from machine $\mu_\lambda(i)$ and feasibly inserted into the machine sequence on $\mu^*(i)$). Obviously, this solution satisfies $\alpha(\mu_{\lambda+1}, \mu^*) < \alpha(\mu_\lambda, \mu^*)$.

If no such operation exists, at least one operation v belonging to a block B on $P^{\mathcal{S}_\lambda}$ is processed before or after the block in $\mathcal{S}^*$. W.l.o.g. assume that v is processed after B, the other case can be treated symmetrically. Since v is not the last operation in B, its machine successor $w := MS(v)$ in $\mathcal{S}$ also belongs to B. This implies that w must also be critical, i.e. $q_v = q_w + p_w$.

In the following we will show that the solution which is obtained by inserting v directly after w belongs to the set N^{vk}, where $k = \mu^*(v)$ is the index of the assigned machine of v in μ^*. Recall that N^{vk} contains all solutions which are obtained by inserting v into π^k after all operations from $L^k \setminus R^k$ and before all operations from $R^k \setminus L^k$. It is sufficient to show that w belongs to $L^k \cap R^k$ and its machine successor $MS(w)$ belongs to R^k, since then v may be moved between w and $MS(w)$ (if $MS(w) \in L^k \cap R^k$, operation v is moved between w and $MS(w)$ in $L^k \cap R^k$, if $MS(w) \in R^k \setminus L^k$, operation v is moved directly before the first operation $MS(w) \in R^k \setminus L^k$, cf. Figure 4.10).

For v we have $q_v^- \leq q_v = q_w + p_w$. If $q_v^- = q_v$ holds, then $q_v = q_{JS(v)} + p_{JS(v)}$ is determined by its job successor $JS(v)$. But then also $JS(v)$ is critical, which contradicts the fact that the machine successor $w = MS(v)$ was chosen in the construction of $P^{\mathcal{S}_\lambda}$. Thus, $q_v^- < q_v = q_w + p_w$ must hold, i.e. w belongs to L^k. Since additionally $r_w + p_w > r_v \geq r_v^-$ holds, w also belongs to R^k. For its machine successor $MS(w)$ we get $r_{MS(w)} + p_{MS(w)} > r_w = r_v + p_v > r_v^-$, i.e. $MS(w) \in R^k$.

For the resulting solution $(\mu_{\lambda+1}, \mathcal{S}_{\lambda+1})$ we have $\alpha(\mu_{\lambda+1}, \mu^*) = \alpha(\mu_\lambda, \mu^*)$ and $\beta(\mathcal{S}_{\lambda+1}, \mathcal{S}^*) < \beta(\mathcal{S}_\lambda, \mathcal{S}^*)$ since one disjunction is oriented into the other direction.

□

4.6 Job-Shop Problems with Transport Robots

In this section we study job-shop problems with transport robots. After giving a precise formulation of the problem in Subsection 4.6.1, we consider two subcases with an unlimited number of robots in Subsection 4.6.2 and a limited number of robots in Subsection 4.6.3. Afterwards, in Sections 4.6.4 to 4.6.6 constraint propagation techniques, lower bounds and heuristic methods are discussed.

4.6.1 Problem formulation

In the following we will generalize the job-shop problem by additionally considering **transportation times**. They occur if a job changes from one machine to another, i.e. if job J_j is processed on machine M_k and afterwards on machine M_l, a transportation time t_{jkl} arises. These transportation times may be job-dependent or job-independent ($t_{jkl} = t_{kl}$). We assume that the transportation is done by **transport robots** (automated guided vehicles) which can handle at most one job at a time.

We again assume that sufficient buffer space exists between the machines. This means that each machine M_k has an unlimited output buffer where jobs processed on M_k and waiting for a robot may be stored. The jobs are automatically transferred into this buffer and no further times for this transfer are considered. Additionally, each machine M_l has an unlimited input buffer where jobs which have been transported and await processing on M_l may be stored.

We suppose $t_{jkk} = 0$ for all jobs J_j and all machines M_k. This means that if two consecutive operations of a job J_j are processed on the same machine, no transport is necessary between them. Furthermore, the transportation times are supposed to satisfy the following triangle inequality for all jobs J_j and all machines M_k, M_l, M_h:

$$t_{jkh} + t_{jhl} \geq t_{jkl}. \tag{4.31}$$

If this inequality does not hold, i.e. $t_{jkh} + t_{jhl} < t_{jkl}$ for some indices j, k, h, l, we could save time for the transport from M_k to M_l by first transporting job J_j from M_k to M_h and then to M_l. In practice, this situation is unlikely to occur, hence this assumption is not a real restriction.

We assume that there are r identical transport robots $R_1, \ldots, R_r$ and each job can be transported by any of these robots. Then the following two cases can be distinguished:

- an **unlimited number of robots** $r \geq N$,
- a **limited number of robots** $r < N$.

If we have an unlimited number of robots, for each of the N jobs J_j a robot exists which can do all transportations concerning job J_j. This means that each job can be transported immediately after finishing on a machine and no conflicts

arise between different transport operations. Thus, we do not have to decide which robot transports which job and the transportation times can be modeled by minimal time-lags between operations of the same job. In the second case not only the machines, but also the robots are a resource with limited capacity. Since a robot can transport only one job at a time, situations may occur in which more jobs require transportation than free robots are available. Then it has to be decided which robot transports which job and which jobs have to wait for a free robot before transportation.

In a further generalization of this model the robots $R_1, \ldots, R_r$ may be non-identical and may have different characteristics (so-called multi-purpose robots). Then it can be possible that some jobs cannot be transported by any of the robots. This more general situation is modeled by associating with each job J_j a set $\mu_j^R \subseteq \{R_1, \ldots, R_r\}$ containing all robots by which J_j may be transported (like in a situation with flexible machines).

In the case of a limited number of robots we consider **empty moving times** t'_{kl} in addition to the transportation times. While the transportation times arise when a job is transported from one machine to another, the empty moving times arise when an empty robot moves from machine M_k to M_l without carrying a job. For these times we assume

$$t'_{kk} = 0, \quad t'_{kh} + t'_{hl} \geq t'_{kl} \text{ and } t'_{kl} \leq \min_{j=1}^{N} \{t_{jkl}\} \tag{4.32}$$

for all machine indices $k, h, l \in \{1, \ldots, m\}$. The first assumption means that no empty moving times have to be considered if the robot waits at a machine for the next transport. The second condition is again a triangle inequality and the third states that moving empty does not take longer than moving a job. In practical situations these assumptions are satisfied in most cases. Due to the empty moving times the scheduling decisions strongly depend on the routing of the robots. In order to avoid waiting times for the jobs the robots should be scheduled in such a way that they do not waste much time by empty moves.

4.6.2 Problems without transportation conflicts

In the following we study the job-shop problem with transportation times t_{jkl} where no conflicts between different transportations arise. This situation occurs if we have an unlimited number $r \geq N$ of identical robots $R_1, \ldots, R_r$, i.e. for each job a transport robot is always available.

In order to simplify the presentation we again number all operations O_{ij} $(j = 1, \ldots, N;\ i = 1, \ldots, n_j)$ consecutively from 1 to $n := \sum_{j=1}^{N} n_j$. For each operation $u \in \{1, \ldots, n\}$ let $J(u)$ be the index j of the associated job J_j and let $\mu(u)$ be the machine on which u has to be processed.

For a given instance of the job-shop problem with transportation times again a **disjunctive graph** $G = (V, C, D)$ may be defined. The only difference between

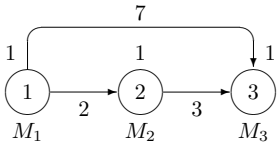
the disjunctive graph for the classical case (without transportation) and the situation with transportation times is that the conjunctions are weighted with the corresponding transportation times. More precisely, we have a conjunction $u \to \sigma(u)$ for each operation u. These arcs are weighted with $w_{u\sigma(u)} = t_{jkl}$ where $j = J(u)$ denotes the job of operation u, and $M_k := \mu(u)$, $M_l := \mu(\sigma(u))$ are the machines between which the transport takes place. If u is the last operation of a job (i.e. $\sigma(u) = n + 1$), the corresponding weight is set to zero.

To solve the scheduling problem we have to find an optimal ordering for all operations processed on the same machine respecting the given conjunctions. This can again be done by turning all undirected arcs in D into directed arcs, i.e. we can again use complete consistent selections. In connection with selections only the calculation of an earliest start schedule has to be modified by taking into account the weights on the arcs.

Given a complete consistent selection $\mathcal{S}$, we may construct a corresponding feasible earliest start schedule as follows. For each operation u let r_u be the length of a longest path from 0 to u in $G(\mathcal{S})$, where the length of a path is defined as the sum of the weights of all vertices and arcs on the path, vertex u excluded. We consider the schedule where each operation u is started at time $S_u := r_u$. Obviously, this schedule is feasible.

We also claim that for this selection no operation u can be started before time r_u, i.e. no operation can be shifted to the left without changing the orders defined by $\mathcal{S}$. Since the transportation times t_{jkl} only occur between operations of the same job which are processed consecutively, all transitive job arcs (these are arcs $u \xrightarrow{t_{jkl}} v$ where an operation w exists with $u \xrightarrow{t_{jkh}} w \xrightarrow{t_{jhl}} v$ and u, v, w belong to the same job) are superfluous and their weights must not be taken into account. It follows from the triangle inequality $t_{jkh} + t_{jhl} \geq t_{jkl}$ in (4.31) that these transitive arcs do not change the longest path lengths r_u. If, on the other hand, the triangle inequality is not satisfied, all transitive arcs have to be removed from the disjunctive graph before the longest paths are calculated, since otherwise they may increase the longest path lengths.

Consider, for example, one job j consisting of 3 unit time operations $i = 1, 2, 3$ with $\mu(i) = M_i$. Let $t_{j12} = 2, t_{j23} = 3, t_{j13} = 7$, i.e. the triangle inequality is violated.



With the orientation $1 \to 2 \to 3$ operation 3 can be started at time $p_1 + t_{j12} + p_2 + t_{j23} = 7$, but the path $0 \to 1 \to 3$ has the length $p_1 + t_{j13} = 8$. Thus, a feasible schedule exists where operation 3 can be started before time $r_3 = 8$.

But, if inequality (4.31) is fulfilled, this situation cannot occur and each complete consistent selection defines a feasible schedule, in which no operation can

be shifted to the left. Again, if the objective function is regular, in the set of schedules represented by complete consistent selections an optimal solution always exists. The completion time $C_{\max}(\mathcal{S})$ of the schedule corresponding to a selection $\mathcal{S}$ is again given by the length of a longest path from 0 to $n+1$ in the acyclic graph $G(\mathcal{S})$.

Example 4.3: Consider again Example 4.1 with $N = 4$ jobs, $m = 4$ machines, and $n = 10$ operations. Furthermore, assume that the transportation times $t_{114} = 5$, $t_{142} = 1$, $t_{242} = 2$, $t_{314} = 2$, $t_{342} = 2$ and $t_{443} = 1$ are given.

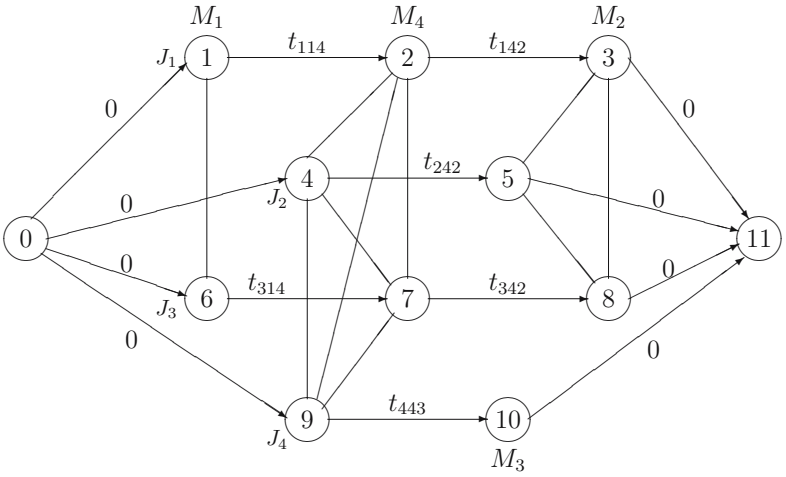


Figure 4.12: Disjunctive graph for a job-shop with transportation

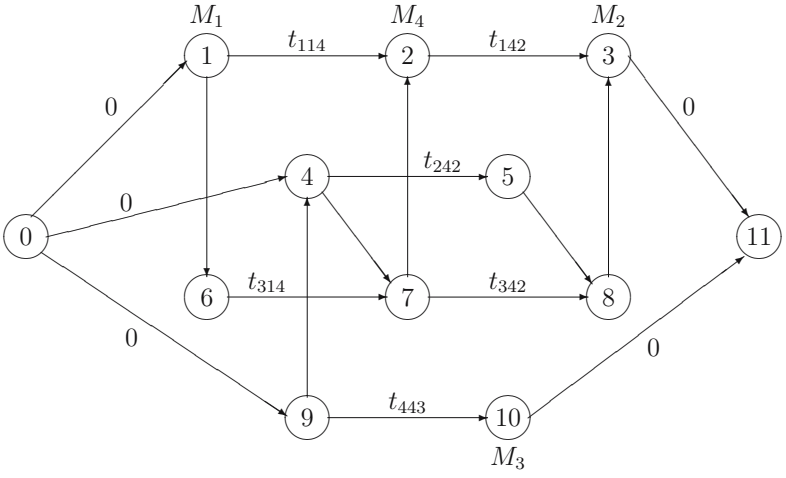


Figure 4.13: A complete selection

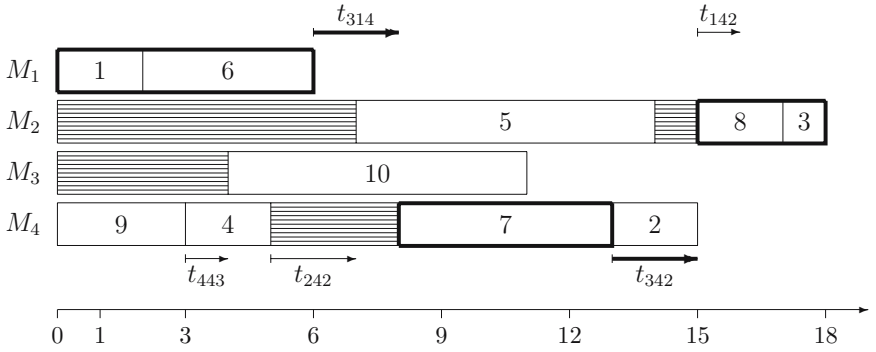


Figure 4.14: A corresponding schedule

The corresponding disjunctive graph can be found in Figure 4.12. A complete consistent selection is shown in Figure 4.13 and the corresponding earliest start schedule is depicted in Figure 4.14.

The makespan of this schedule is $C_{\max} = 18$, a critical path is $0 \to 1 \to 6 \xrightarrow{t_{314}} 7 \xrightarrow{t_{342}} 8 \to 3 \to 11$ with length $p_1 + p_6 + t_{314} + p_7 + t_{342} + p_8 + p_3 = 18$. Note that compared with the schedule without transportation times in Figure 4.3, the makespan increased by 3 time units. □

4.6.3 Problems with transportation conflicts

In the following we study the job-shop problem with transportation times where conflicts between different transportations may arise. This situation occurs if we have a limited number $r < N$ of robots $R_1, \ldots, R_r$ or if we have an environment with multi-purpose robots, i.e. some jobs may not be transported by any robot. In these environments it may happen that more jobs require transportation than free robots are available. Then it has to be decided which job is transported by which robot and which job has to wait before its transportation.

In order to extend the disjunctive graph model to these situations we have to integrate the possible transport conflicts for the robots into the model. In the model of Section 4.6.2 the transportation times are modeled by weights on the corresponding arcs in the graph, which implies that these times are only taken into account as delays between different operations. Implicitly, it is assumed that all these transportations can be carried out simultaneously. In the situation with conflicts we have to ensure that the transportation stage becomes a bottleneck stage which will be done by introducing transport operations as additional vertices in the disjunctive graph and requiring that these operations have to be processed by the robots.

In the following we consider the job-shop problem with transportation and empty moving times and multi-purpose transport robots $R_1, \ldots, R_r$. Each job J_j consists of n_j operations O_{ij} $(i = 1, \ldots, n_j)$ with chain precedence constraints

$O_{1j} \to O_{2j} \to \ldots \to O_{n_j,j}$. Furthermore, for each j a set $\mu_j^R \subseteq \{R_1, \ldots, R_r\}$ is given containing all robots by which job J_j may be transported.

In order to extend the disjunctive graph model to these problems we proceed as follows. For each job J_j $(j = 1, \ldots, N)$ we introduce $n_j - 1$ so-called **transport operations** T_{ij} $(i = 1, \ldots, n_j - 1)$ with precedences $O_{ij} \to T_{ij} \to O_{i+1,j}$. The processing time of T_{ij} is equal to the transportation time of job J_j from machine μ_{ij} to $\mu_{i+1,j}$, i.e. $p(T_{ij}) = t_{jkl}$, when $\mu_{ij} = M_k$, $\mu_{i+1,j} = M_l$. The robots may be considered as additional "machines" which have to process the transport operations (where transport operation T_{ij} can only be processed by one of the robots in μ_j^R). To construct a feasible schedule for the job-shop problem with multi-purpose robots we have to

- assign each transport operation T_{ij} to a robot from the set μ_j^R,
- determine feasible starting times for all operations O_{ij} on the machines M_k, and
- determine feasible starting times for all transport operations T_{ij} on the assigned robots.

As before, we want to use the disjunctive graph model to construct optimal schedules. This means we determine orders for all operations processed on the same machine and orders for all transport operations assigned to the same robot. Since the possible conflicts between transport operations on the same robot depend on the robot assignment, such an assignment has to be given before a disjunctive graph can be defined. Thus, the problem may be solved with a two-stage approach where at first robots are assigned to the transport operations and afterwards the scheduling problem is solved. In the following we assume that we are given a complete robot assignment ρ where each transport operation T_{ij} is assigned to a robot $\rho(T_{ij}) \in \mu_j^R$. Let G_{kl} $(k, l = 1, \ldots, m)$ be the set of all transport operations with the same transport routing, i.e. we have

$$T_{ij} \in G_{kl} \text{ if and only if } \mu_{ij} = M_k,\ \mu_{i+1,j} = M_l.$$

Then the **disjunctive graph** $G = (V, C, D_M \cup D_R)$ associated with ρ is defined as follows.

- The set of vertices $V := \{O_{ij} \mid j = 1, \ldots, N;\ i = 1, \ldots, n_j\} \cup \{T_{ij} \mid j = 1, \ldots, N; i = 1, \ldots, n_j - 1\} \cup \{0, *\}$ represents all operations and transport operations of the jobs and two dummy nodes 0 and $*$. The source 0 and the sink $*$ are used to define the start and the end of the operations. The operations O_{ij} are weighted with their processing times p_{ij}, the transport operations T_{ij} with their transportation times $p(T_{ij})$, and the weights of the dummy nodes are 0.
- The set of **conjunctions** C represents the precedence constraints $O_{ij} \to T_{ij} \to O_{i+1,j}$ for all jobs $j = 1, \ldots, N$ and $i = 1, \ldots, n_j - 1$, weighted

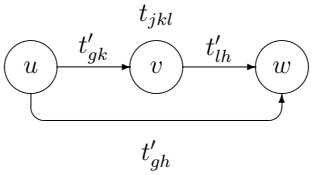
with 0. Additionally, there are arcs between the source and all operations without any predecessor, and between all operations without any successor and the sink.

- The set of **machine disjunctions** D_M consists of all pairs of operations which have to be processed on the same machine and are not linked by conjunctions. An undirected arc $O_{ij} - O_{uv} \in D_M$ with $\mu_{ij} = \mu_{uv} = M_k$ represents the two possible orders in which the operations O_{ij} and O_{uv} may be processsed on M_k and is weighted with 0.

- The set of **robot disjunctions** D_R consists of all pairs of transport operations which are not linked by conjunctions and are assigned to the same robot. An undirected arc $T_{ij} - T_{uv} \in D_R$ represents the two possible orders in which the transport operations T_{ij} and T_{uv} may be processsed on the robot and is weighted with the pair (t'_{kl}, t'_{gh}) of empty moving times, when $T_{ij} \in G_{hk}, T_{uv} \in G_{lg}$.

To solve the scheduling problem we have to turn all undirected arcs in $D_M \cup D_R$ into directed ones. Concerning an undirected arc in D_R weighted with a pair of empty moving times, at this point the directed arc gets the corresponding unique weight. If we orient a disjunction $T_{ij} - T_{uv}$ with $T_{ij} \in G_{hk}, T_{uv} \in G_{lg}$ in the direction $T_{ij} \rightarrow T_{uv}$, it gets the weight t'_{kl}, and if we orient it in the other direction, it gets the weight t'_{gh}. In order to distinguish between machines and robots we call a set $\mathcal{S}_M$ of fixed machine disjunctions a **machine selection**, a set of fixed robot disjunctions $\mathcal{S}_R$ a **robot selection** and their union $\mathcal{S} = \mathcal{S}_M \cup \mathcal{S}_R$ a **selection**. If all disjunctive arcs have been fixed and the corresponding graph $G(\mathcal{S}) = (V, C \cup \mathcal{S})$ is acyclic, $\mathcal{S}$ is called a **complete consistent selection**. Given a complete consistent selection $\mathcal{S}$ a corresponding schedule can be constructed as in Section 4.6.2 by longest path calculations and starting each operation as early as possible. The makespan $C_{\max}(\mathcal{S})$ of the schedule corresponding to $\mathcal{S}$ is given by the length of a longest path from 0 to $*$ in $G(\mathcal{S})$.

Since empty moving times only occur between transportation times which are processed consecutively on the same robot, all transitive arcs corresponding to fixed robot disjunctions must not be taken into account in the longest path calculations. But, as in Section 4.6.2, we can show that paths consisting of transitive arcs are not longer than paths consisting of non-transitive arcs, i.e. the transitive arcs do not change the longest path lengths. Let u, v, w be three transport operations which are assigned to the same robot where the corresponding robot disjunctions are directed to $u \rightarrow v \rightarrow w$. Assuming that operation v belongs to job j, we have the following situation:



Due to the assumptions (4.32) for the empty moving times we have

$$t'_{gh} \leq t'_{gk} + t'_{kl} + t'_{lh} \leq t'_{gk} + \min_{i=1}^{N} \{t_{ikl}\} + t'_{lh} \leq t'_{gk} + t_{jkl} + t'_{lh}, \tag{4.33}$$

i.e. the transitive arc $u \to w$ is not longer than the path $u \to v \to w$ consisting of non-transitive arcs. By transitivity the same property also holds for all paths consisting of more than three transport operations, which shows that transitive arcs do not have to be removed from the disjunctive graph.

Example 4.4: We consider a job-shop instance with $m = 4$ machines, $r = 2$ multi-purpose robots, $N = 3$ jobs, 8 "ordinary" operations and 5 transport operations. Let $\mu_1^R = \mu_2^R = \{R_1, R_2\}$ and $\mu_3^R = \{R_2\}$. The corresponding disjunctive graph for the robot assignment $\rho(T_{11}) = \rho(T_{12}) = R_1$, $\rho(T_{21}) = \rho(T_{13}) = \rho(T_{23}) = R_2$ can be found in Figure 4.15. All transport operations which do not belong to the same job and which are assigned to the same robot are linked by a robot disjunction weighted with the corresponding empty moving times. For example, the disjunction on R_1 between the transport operations $T_{11} \in G_{12}$ and $T_{12} \in G_{23}$ is weighted with the pair (t'_{22}, t'_{31}).

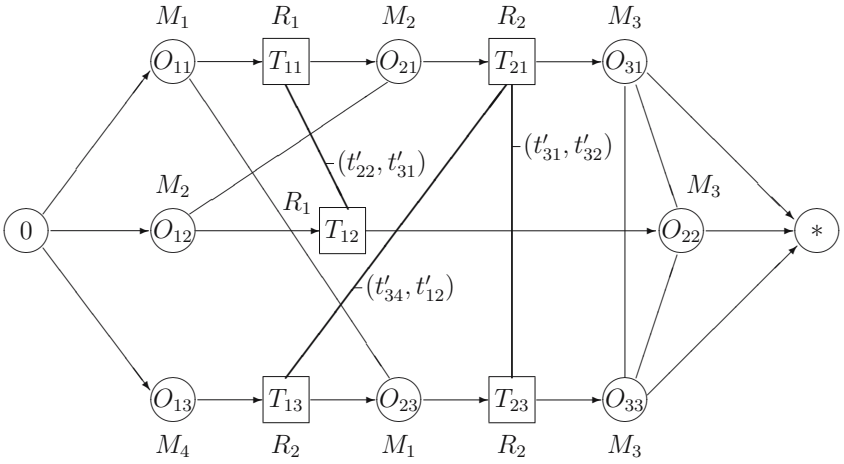


Figure 4.15: Disjunctive graph for a job-shop with $r = 2$ robots

If we only have a single robot which has to perform all transportations, we get the disjunctive graph in Figure 4.16 (for clarity not all empty moving times are drawn). In Figure 4.17 a complete selection for this graph is shown in which the machine disjunctions are oriented into $O_{11} \to O_{23}$ on M_1, into $O_{21} \to O_{12}$ on M_2, into $O_{22} \to O_{31} \to O_{33}$ on M_3, and the robot sequence is given by $T_{11} \to T_{13} \to T_{12} \to T_{21} \to T_{23}$. A corresponding schedule with the critical path $O_{13} \to T_{13} \xrightarrow{t'_{12}} T_{12} \xrightarrow{t'_{32}} T_{21} \to O_{31} \to O_{33}$ can be found in Figure 4.18. □

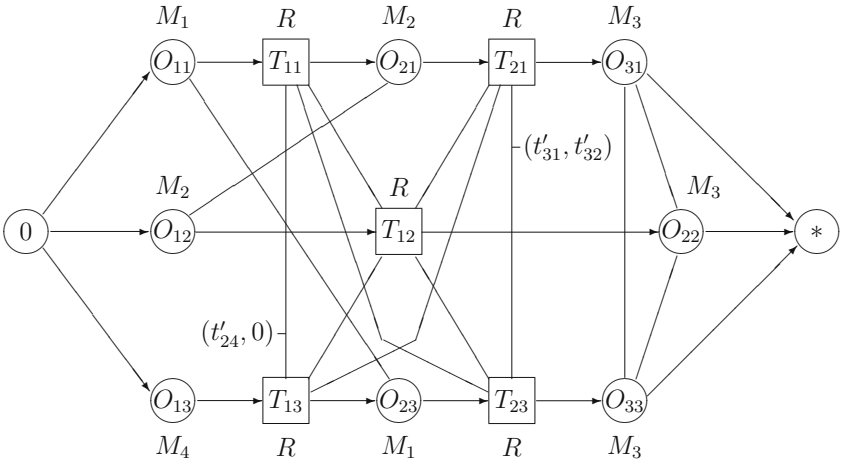


Figure 4.16: Disjunctive graph for a job-shop with a single robot

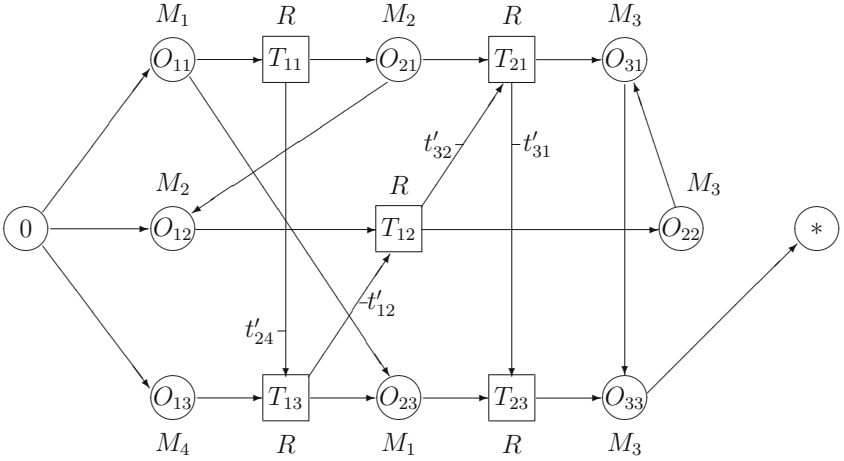


Figure 4.17: A complete selection

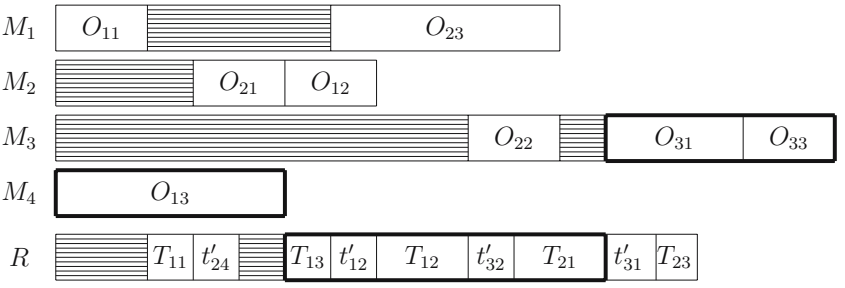


Figure 4.18: A corresponding schedule

Note that if we have only a single robot and no empty moving times (i.e. $t'_{kl} = 0$), the problem is equivalent to a classical job-shop problem with $m + 1$ machines, where the robot corresponds to an additional machine. Since this machine has to process many more operations than the other machines (each second operation of a job), it is also called a bottleneck machine.

Up to know we considered the scheduling problem which arises in the job-shop problem with multi-purpose robots for a fixed robot assignment. This scheduling problem consists of two parts: on the one hand, the jobs have to be scheduled on the machines, and, on the other hand, the transport operations have to be scheduled on the assigned robots. In the following we go one step further and consider the situation for the robots when additionally the machine orders are given (i.e. a machine selection is fixed).

Such a hierarchical decomposition of the problem may be useful in different situations. Since for the machines we have a similar situation as in a classical job-shop, several techniques known for the job-shop problem may be applied to the machines (like constraint propagation, lower bounds, heuristic methods). Thus, if we want to integrate a transportation stage into existing procedures, it remains to handle the robots appropriately.

In the following we assume that we are given a robot assignment and a machine selection $\mathcal{S}_M$ represented by the graph $G(\mathcal{S}) = (V, C \cup \mathcal{S}_M)$. We are interested in an optimal robot selection $\mathcal{S}_R$ respecting the precedence relations C and the orders on the machines defined by $\mathcal{S}_M$. Let V' be the set of all transport operations and define the length of a path between two operations i, j in $G(\mathcal{S})$ as the sum of processing times of all operations on the path (p_i, p_j excluded).

With each transport operation $j \in V'$ we may associate a **head** and a **tail**. Recall that a head r_j is a lower bound for the earliest possible starting time of j, and a tail q_j is a lower bound for the time period between the completion time of j and the optimal makespan. For a given selection $\mathcal{S}$ heads r_j may be defined by the lengths l_{0j} of longest paths from the source 0 to node j in $G(\mathcal{S})$ since operation j cannot be started before all predecessors in $G(\mathcal{S})$ are scheduled. Symmetrically, tails q_j may be defined by the lengths l_{j*} of longest paths from j to $*$ in $G(\mathcal{S})$.

Since the given precedence constraints C and the fixed machine disjunctions $\mathcal{S}_M$ have to be respected in a feasible robot selection, some additional constraints for the transport operations can be derived from the graph $G(\mathcal{S})$. Obviously, if for transport operations $i, j \in V'$ a path from i to j in $G(\mathcal{S})$ exists, i has to be processed before j. Furthermore, the longest path lengths l_{ij} between i and j define minimal finish-start **time-lags** between the completion time C_i of i and the starting time S_j of j (e.g. between two consecutive transportations of the same job the job has to be processed on a machine for a certain time period). Let C' be the set of all precedence relations for the transport operations (including the transitive ones) induced in this way. For all $i \to j \in C'$ we have to fulfill $S_j - C_i \geq l_{ij}$.

All pairs of transport operations $i, j \in V'$ which are not linked by a conjunction

from C', but are assigned to the same robot, cannot be processed simultaneously. This can again be modeled by a set D' of disjunctions. As previously described, these disjunctions $i-j \in D'$ are weighted with sequence-dependent **setup times** (s_{ij}, s_{ji}). The setup time s_{ij} arises if i is processed directly before j on the same robot (corresponding to an empty move between the two transportations).

Summarizing, to determine an optimal robot selection $\mathcal{S}_R$ for a fixed machine selection $\mathcal{S}_M$ we get the following scheduling problem for the transport operations: schedule all transport operations $j \in V'$ on their assigned robots in such a way that the release dates r_j, the minimal time-lags l_{ij} and sequence-dependent setup times s_{ij} are respected and the extended makespan $\max_{j \in V'}\{C_j + q_j\}$ is minimized. Note that this objective value equals the makespan $C_{\max}(\mathcal{S})$ of the complete selection $\mathcal{S} = \mathcal{S}_M \cup \mathcal{S}_R$ since all restrictions of the job-shop problem are incorporated into the constraints for the problem of the robots.

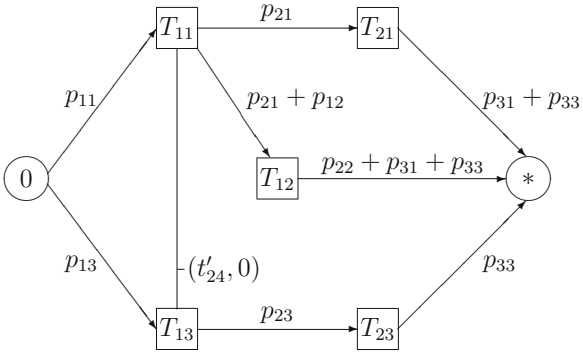
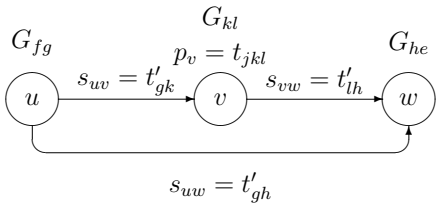


Figure 4.19: The robot problem for a fixed machine selection

Example 4.5: Consider again Example 4.4 with a single robot and the machine selection from Figure 4.17. Then we get a situation for the robot as presented in Figure 4.19. Besides the chain precedences of the jobs we get a conjunction $T_{11} \rightarrow T_{12}$ weighted with the time-lag $p_{21} + p_{12}$ induced by the machine orientation $O_{21} \rightarrow O_{12}$. Between all transport operations which are not linked by a conjunction, a disjunction weighted with the corresponding pair of empty moving times exists (in Figure 4.19 for clarity only the disjunction $T_{11} - T_{13}$ is shown). □

Finally, we derive some special properties which are satisfied for job-shop problems with transportation. These properties will be used in connection with algorithms for the robot in later sections. Let u, v, w be three transport operations. Then we have a situation as follows where the setup times result from the corresponding empty moving times.



The setup times satisfy the weak triangle inequality $s_{uw} \leq s_{uv} + p_v + s_{vw}$ since, due to the assumptions (4.32) for the empty moving times, we have

$$s_{uw} = t'_{gh} \leq t'_{gk} + t'_{kl} + t'_{lh} \leq t'_{gk} + \min_{i=1}^{N} \{t_{ikl}\} + t'_{lh} \leq t'_{gk} + t_{jkl} + t'_{lh} = s_{uv} + p_v + s_{vw}.$$

Furthermore, we have the following triangle inequalities which link the setup times with the minimal time-lags:

$$s_{uw} \leq s_{uv} + p_v + l_{vw} \text{ for all } u - v, u - w \in D', v \to w \in C', \tag{4.34}$$
$$s_{uw} \leq l_{uv} + p_v + s_{vw} \text{ for all } v - w, u - w \in D', u \to v \in C'. \tag{4.35}$$

To prove the first inequality, we distinguish the following three cases:

- If v, w are transport operations belonging to the same job, this job has to be transported from machine M_l to machine M_h in between. This means that we either have a direct move from M_l to M_h with duration t_{jlh} or a sequence of moves between machines $M_l = M_{k_1}, M_{k_2}, \ldots, M_{k_\mu} = M_h$ with duration $\sum_{\lambda=1}^{\mu-1} t_{jk_\lambda k_{\lambda+1}} \geq t_{jhl}$ due to the triangle inequality (4.32). Since transportation times are not smaller than the corresponding empty moving times, we have $l_{vw} \geq t_{jlh} \geq t'_{lh}$, which implies

$$s_{uw} = t'_{gh} \leq t'_{gk} + t'_{kl} + t'_{lh} \leq s_{uv} + p_v + l_{vw}.$$

- If v, w belong to different jobs and the path in $G(\mathcal{S})$ between v and w does not contain a job arc, all arcs on the path must be directed machine disjunctions, i.e. we have $M_l = M_h$. Thus,

$$s_{uw} = t'_{gh} \leq t'_{gk} + t_{jkl} + t'_{lh} = t'_{gk} + t_{jkl} + t'_{ll} = s_{uv} + p_v \leq s_{uv} + p_v + l_{vw}$$

  due to $t'_{ll} = 0$ and $l_{vw} \geq 0$.

- The third case in which v, w belong to different jobs and the path in $G(\mathcal{S})$ between v and w contains a job arc can be proved by combining the first two cases and using transitivity.

The second inequality can be proved in a similar way.

4.6.4 Constraint propagation

In this subsection we describe constraint propagation techniques for the job-shop problem with transport robots. These techniques may be used to derive additional conjunctions and to strengthen heads, tails and time-lags. They may be useful in connection with destructive lower bound calculations or in connection with branch-and-bound algorithms based on the disjunctive graph model where iteratively disjunctive arcs are fixed in a branching step.

Since the machines $M_1, \ldots, M_m$ can be considered as disjunctive resources, all techniques for disjunctive sets from Section 3.2.4 may be applied to the machines. On the other hand, the situation on the robots is more involved since additionally time-lags and setup times have to be taken into account. In the following we consider the subproblem for a single robot where on the machine level some (or all) machine disjunctions may already be fixed (cf. the description on page 226).

Recall that in the robot problem we are given a set $V := \{1, \ldots, n\} \cup \{0, *\}$ of transport operations where 0 and $*$ are dummy operations with $p_0 = p_* = 0$. All operations $j \in V$ have to be processed without preemptions for p_j time units on a single robot. Operation j cannot be started before its release date (head) r_j and stays in the system for q_j time units (tail) after processing. Furthermore, minimal finish-start time-lags $i \xrightarrow{l_{ij}} j \in C$ with integers $l_{ij} \geq 0$ are given for some pairs of operations. If $i \xrightarrow{l_{ij}} j \in C$ holds, operation j cannot be started earlier than l_{ij} time units after the completion of operation i. By these time-lags precedence relations (conjunctions) are induced which will be denoted by $i \to j \in C$. We assume that all transitive arcs $i \to j$ (these are arcs where an operation h with $i \to h \to j$ exists) are contained in the set C and that the time-lags are transitively adjusted, i.e. $l_{ih} + p_h + l_{hj} \leq l_{ij}$ holds. Furthermore, we regard the heads and tails as special time-lags by setting $l_{0j} := r_j$ and $l_{j*} := q_j$ for all operations $j = 1, \ldots, n$.

Since on a single robot two operations cannot be processed simultaneously, all pairs of operations i, j which are not linked by a conjunction $i \to j$ or $j \to i$ are contained in the set D of disjunctions. These disjunctions $i - j \in D$ are weighted with pairs (s_{ij}, s_{ji}) of sequence-dependent setup times with the following meaning: If j is directly processed after operation i, a setup s_{ij} between the completion time of i and the starting time of j has to be considered. If, on the other hand, i is processed directly after j, a setup s_{ji} occurs. We assume that the setup times satisfy the weak triangle inequality $s_{ih} + p_h + s_{hj} \geq s_{ij}$ for all disjunctions. Furthermore, minimal time-lags and setups are supposed to satisfy the triangle inequalities (4.34) and (4.35), i.e.

$$l_{ih} + p_h + s_{hj} \geq s_{ij} \text{ and } s_{ih} + p_h + l_{hj} \geq s_{ij}. \tag{4.36}$$

Note that the inequality $l_{ih} + p_h + l_{hj} \geq s_{ij}$ is not needed since the time-lags are transitively adjusted and therefore for conjunctions $i \to h, h \to j \in C$ also $i \to j \in C$ holds (i.e. no disjunction $i - j \in D$ exists).

The problem is to determine feasible completion times C_j for all operations $j \in V$ which respect the release dates r_j, the minimal time-lags l_{ij}, the setup times s_{ij}, and minimize the extended makespan $\max_{j=1}^{n} \{C_j + q_j\}$, which equals the completion time C_* of the dummy operation $*$.

In connection with the robot problem constraint propagation techniques may

- derive additional precedence constraints (conjunctions C), and
- improve heads r_j, tails q_j and minimal time-lags l_{ij}.

Given an instance of the robot problem and a threshold value T for the extended makespan, we define a **distance matrix** $d = (d_{ij})_{i,j \in V}$ like in Section 3.2.2 by

$$d_{ij} = \begin{cases} 0, & \text{if } i = j \\ p_i + l_{ij}, & \text{if } i \to j \in C \\ p_i - T, & \text{otherwise.} \end{cases} \tag{4.37}$$

For each pair of operations $i, j \in V$ the value d_{ij} is a lower bound for the difference $S_j - S_i$ of their starting times (minimal start-start time-lag). If $i \to j \in C$ holds, we have to satisfy $S_j - S_i \geq p_i + l_{ij}$, which is ensured by the second condition in (4.37). Furthermore, in each feasible schedule with $S_0 = 0$ and $S_* \leq T$ we must have $S_i - S_j \leq S_* - p_i - S_0 \leq T - p_i$, which implies $S_j - S_i \geq p_i - T$ (cf. the third condition in (4.37)). Heads and tails are regarded as special time-lags $l_{0j} = r_j, l_{i*} = q_i$ and are incorporated into d by the second condition in (4.37).

As in Section 3.2.2 the distance matrix may be transitively adjusted by the Floyd-Warshall algorithm, i.e. the entry d_{ij} may be replaced by $\max\{d_{ij}, d_{ik} + d_{kj}\}$ for all triples $i, j, k \in V$. If, after applying this algorithm, $d_{ii} > 0$ holds for some operation $i \in V$, a positive cycle is indicated, i.e. no feasible schedule for the threshold T exists.

Additional conjunctions may be derived from the adjusted distance matrix by testing the following conditions (which generalize the situation without setup times):

- **Distance update**: If $d_{ij} > -p_j$ holds for a disjunction $i - j \in D$, then we may introduce the conjunction $i \to j \in C$ and set $d_{ij} := \max\{d_{ij}, p_i + s_{ij}\}$ (cf. (3.10)).
- **Direct arcs**: If $d_{0j} + p_j + s_{ji} + d_{i*} > T$ holds for a disjunction $i - j \in D$, then the conjunction $i \to j \in C$ may be fixed.
- **Triple arcs**: If the three conditions

$$\begin{aligned} d_{0j} + p_j + s_{ji} + p_i + s_{ik} + d_{k*} &> T \\ d_{0j} + p_j + s_{jk} + p_k + s_{ki} + d_{i*} &> T \\ d_{0k} + p_k + s_{kj} + p_j + s_{ji} + d_{i*} &> T \end{aligned}$$

hold for a disjunction $i - j \in D$ and another operation $k \in V$, then the conjunction $i \to j \in C$ may be fixed.

Fixing all direct arcs and all triple arcs can be done in $O(n^2)$ and $O(n^3)$, respectively. If a disjunction $i - j \in D$ is replaced by a conjunction $i \xrightarrow{l_{ij}} j \in C$, the corresponding time-lag l_{ij} is always set to the weight s_{ij} of the pair (s_{ij}, s_{ji}), i.e. $l_{ij} := s_{ij}$. Thus, the triangle inequalities (4.36) remain valid after applying constraint propagation.

In the following we will use ideas developed for the traveling salesman problem with time windows where the impact of processing times, setup times and time-lags is considered simultaneously. This method may be used to detect infeasibility or to improve heads and tails. In order to have a unique notation for the time which has to be considered after the completion time of operation i, if operation j is planned directly after i, we define

$$t_{ij} := \begin{cases} l_{ij}, & \text{if } i \to j \in C \\ s_{ij}, & \text{if } i - j \in D \end{cases} \tag{4.38}$$

for all operations $i, j \in V$ where i may be processed before j, i.e. for which the relation $j \to i$ is not contained in C.

Given a threshold value T, we consider the feasibility problem for the robot with deadlines $d_j := T - q_j$. A feasible schedule for this problem has to respect

- the time windows $[r_j, d_j]$ of all operations $j \in V$,
- all precedence relations $i \to j \in C$ with the corresponding minimal time-lags l_{ij}, and
- all setup times s_{ij}.

In the following we consider a relaxation in which we relax all minimal time-lags between operations i, j which are not planned consecutively in the schedule. By showing that no feasible solution for this relaxation exists, we may conclude that no feasible schedule for the original problem with threshold T exists.

Note that a feasible schedule for this relaxation is only feasible for the original problem if the triangle inequality $t_{ij} \leq t_{ih} + p_h + t_{hj}$ holds for all operations i, j which are planned consecutively. Since the inequalities $l_{ij} > l_{ih} + p_h + l_{hj}$, $l_{ij} > s_{ih} + p_h + s_{hj}$, $l_{ij} > l_{ih} + p_h + s_{hj}$ or $l_{ij} > s_{ih} + p_h + l_{hj}$ for some $i, j, h \in V$ may hold, in general this triangle inequality is not valid.

The resulting problem is a generalization of the traveling salesman problem with time windows. To solve the relaxation we generalize the dynamic programming approach for the traveling salesman problem presented in Example 2.18.

We consider a fixed subset $V^0 \subseteq V \setminus \{0\}$. A subset $S \subseteq V^0$ is called **closed** (with respect to conjunctions) if for each operation $j \in S$ all predecessors $i \in V^0$ with $i \to j \in C$ are also contained in S. For a closed set $S \subseteq V^0$ and an operation

$j \in S$ let $F(S,j)$ be the earliest starting time for operation j when all operations in the set $S \setminus \{j\}$ are processed before j. Two possibilities exist for a feasible schedule realizing $F(S,j)$:

- either operation j starts at its release date r_j, or
- operation j starts at time $S_i + p_i + t_{ij}$ directly after another operation i.

In the first case we have $F(S,j) = r_j$, in the second we have to select the minimal value $F(S \setminus \{j\}, i) + p_i + t_{ij}$ among all operations $i \in S \setminus \{j\}$ which may be processed last in the set $S \setminus \{j\}$. If this minimal value is larger than the deadline $d_j - p_j$ of operation j, the corresponding schedule with operation j as a last operation is infeasible with respect to the time windows, i.e. in this case we may set $F(S,j) := \infty$.

Initially, we set $F(\{j\}, j) := r_j$ for all operations j without predecessor (corresponding to the closed sets $\{j\}$ consisting of only one operation). For all other operations j the set $\{j\}$ is not closed, i.e. we may set $F(\{j\}, j) := \infty$. Thus, the values $F(S,j)$ can be calculated recursively as follows.

Initialization:

$$F(\{j\}, j) := \begin{cases} r_j, & \text{if } j \text{ has no predecessor } i \text{ with } i \to j \in C \\ \infty, & \text{otherwise} \end{cases}$$

$$\text{for all } j \in V^0.$$

Recursion:

$$F(S \cup \{j\}, j) =$$

$$\begin{cases} \max[r_j, \min\limits_{i \in S}\{F(S,i) + p_i + t_{ij}\}] & , \text{ if } \min\limits_{i \in S}\{F(S,i) + p_i + t_{ij}\} \le d_j - p_j \\ \infty & , \text{ otherwise} \end{cases}$$

$$\text{for all } S \subseteq V^0 \text{ and all } j \notin S \text{ whose predecessors are all contained in } S.$$

For all other j the set $S \cup \{j\}$ is not closed and we may also assume $F(S \cup \{j\}, j) := \infty$. The optimal solution value for the fixed set V^0 is given by $F(V^0, *)$. If $F(V^0, *) = \infty$, no feasible schedule for the set V^0 exists.

We have to calculate $F(S,j)$ for all closed subsets $S \subseteq V^0$ and all operations $j \in S$. Since each computation can be done in $O(n)$ time, the running time of the whole dynamic programming procedure is $O(n^2 \cdot 2^n)$. The storage requirement to store all $F(S,j)$-values is $O(n \cdot 2^n)$, which can be reduced to $O(n \cdot \binom{n}{\lceil n/2 \rceil})$ by generating all closed sets S in the recursion according to non-increasing cardinalities (cf. Example 2.18). This can be seen as follows. In order to calculate the objective value for a subset S with cardinality $k \in \{2, \ldots, n\}$, only subsets $S \setminus \{j\}$ of cardinality $k-1$ are needed. Thus, in iteration k we only have to store the closed subsets of cardinalities $k-1$ and k. When we proceed with subsets

of cardinality $k+1$, all subsets with cardinality $k-1$ can be removed. Since at most $\binom{n}{k}$ subsets of cardinality k exist, the maximal storage requirement is given by $\max_{k=1}^{n}\{\binom{n}{k}+\binom{n}{k-1}\} \leq 2\binom{n}{\lceil n/2\rceil}$ because the maximum is achieved for $k=\lceil\frac{n}{2}\rceil$. Thus, the storage requirement is bounded by $O(n\cdot\binom{n}{\lceil n/2\rceil})$.

Although the worst-case complexity of the proposed algorithm is exponential, the practical complexity is much smaller. This is due to the fact that many subsets $S\subseteq V^0$ are not closed with respect to the precedence relations, i.e. in practice much fewer than 2^n subsets have to be considered.

If we apply the dynamic programming algorithm to the whole set V, the optimal value $F(V,*)$ provides a (constructive) lower bound for the optimal objective value of the robot problem. On the other hand, the algorithm may also be used in a destructive way. If no feasible solution for the relaxation exists, no feasible solution for the robot problem with threshold T exists. Thus, in a destructive lower bound procedure we may calculate the largest T where infeasibility can be proved.

Since the dynamic programming procedure may be very time-consuming for larger n, in the following we describe some local considerations in which we apply the algorithm only to subsets V^0 of all operations in V. One possibility is to consider an interval $[a,b]$ and all operations which may be (at least partially) processed in this interval. This means we apply the algorithm to the set $V^{[a,b]} := \{j\in V \mid [r_j,d_j]\cap[a,b]\neq\emptyset\}$. If no feasible solution for this subset exists, we may conclude that no feasible solution exists for the whole set.

Another application of the dynamic programming procedure may improve heads and tails of the operations as follows. For each operation j we consider the set $S^j := \{i \mid i\to j\in C\}\cup\{j\}$ consisting of all predecessors of operation j (including the transitive predecessors) and j itself. Since the value $F(S,j)$ defines the minimal starting time for operation j when all operations in the set $S\setminus\{j\}$ are processed before, the value $F(S^j,j)$ is a lower bound for the starting time of operation j in each feasible schedule. Thus, if $F(S^j,j)>r_j$ holds, we may increase the head of j to the corresponding value. Note that during the calculation of the $F(S^j,j)$-values, the values $F(S^i,i)$ for all operations $i\in S^j$ are also calculated. Hence, the algorithm does not have to be called individually for each operation and heads of various operations may be improved simultaneously. Symmetrically, by interchanging the role of heads and tails, tails q_j may be increased during a backward calculation for sets S^j consisting of all successors of operation j and j itself.

All the presented constraint propagation techniques may be integrated into several (more or less time-consuming) procedures which are applied for a given threshold value T. A basic procedure is shown in Figure 4.20.

This procedure may be extended to a procedure which repeats Steps 3 and 4 until no further direct or triple arcs can be found. Since at most $O(n^2)$ arcs may be fixed and updating the distance matrix can be done in $O(n^3)$, both procedures are polynomial and their worst-case complexity is $O(n^5)$.

1. Define the distance matrix d according to (4.37).
2. Calculate the transitive closure of d taking into account all distance updates. If a positive cycle is detected, infeasibility is stated.
3. Fix all direct arcs and update d as in Step 2 until no further direct arcs can be found.
4. Fix all triple arcs and update d as in Step 2 until no further triple arcs can be found.

Figure 4.20: Procedure `ConstraintPropagation` (T)

In order to improve heads and tails with the dynamic programming algorithm, we may proceed as follows.

1. For all operations $j \in V$ determine the sets
$$S^j := \{i \mid i \to j \in C\} \cup \{j\}$$
consisting of all (including the transitive) predecessors of operation j and j itself.
2. Consider the operations according to a list in which the operations are sorted with respect to non-increasing cardinalities of the sets S^j. For each operation j in the list calculate $F(S^j, j)$ with the dynamic programming algorithm. If no feasible solution for subset S^j exists, infeasibility is detected. Otherwise, during the calculation of the value $F(S^j, j)$, all values $F(S^i, i)$ for predecessors i of j have also been calculated. Set $r_i := F(S^i, i)$ for all $i \in S^j$ and eliminate all considered operations i from the list. If after considering all operations some heads have been improved, transform these heads back into the distance matrix d and update d as in Step 2 from procedure `ConstraintPropagation` (T).
3. Symmetrically, improve tails q_j by considering the operations according to non-increasing cardinalities of their successor sets
$$\hat{S}^j := \{i \mid j \to i \in C\} \cup \{j\}.$$
If after considering all operations some tails have been improved, transform these tails back into the distance matrix d and update d as in Step 2 from procedure `ConstraintPropagation` (T).

Again, this procedure may be extended to a procedure which repeats Steps 2 and 3 until no further heads or tails can be improved. In a further extension after transforming back the improved heads and tails in Steps 2 and 3, the first constraint programming techniques may be applied. Note that these procedures are not polynomial, since the dynamic programming algorithm has worst-case complexity $O(n^2 \cdot 2^n)$.

4.6.5 Lower bounds

In this subsection we describe the computation of lower bounds for the job-shop problem with transport robots. We assume that we are given a robot assignment ρ where each transport operation is assigned to a robot. If all empty moving times are relaxed, we get a classical job-shop problem, i.e. all lower bound methods known for the job-shop problem may be used. On the other hand, in order to cover the empty moving times, we have to consider the situation on the robots more carefully. For this reason, in the following we again concentrate on the robot problem from the previous subsection taking into account setup times s_{ij} (corresponding to empty moving times) and time-lags l_{ij} (induced by the job chains and fixed machine disjunctions).

Concerning constructive lower bounds it is very difficult to find a good relaxation for the robot problem which is easy to solve. Unfortunately, relaxing one of the constraints in the robot problem does not make the problem much easier (since the single-machine problem with either heads and tails, setup times or minimal time-lags is already NP-hard). Thus, the only way to obtain polynomially solvable relaxations is to drop several constraints. For example, if we relax the setup times s_{ij}, the minimal time-lags l_{ij}, and additionally allow preemption, the resulting problem is polynomially solvable. But since in this relaxation many constraints have been dropped, the resulting lower bounds are quite poor.

For this reason we concentrate on destructive lower bounds. Given a threshold value T for the robot problem we define deadlines $d_j := T - q_j$ and consider the corresponding instance of the feasibility problem. One possibility to detect infeasibility for such an instance is to use the constraint propagation techniques from the previous subsection. Another possibility is to use linear programming (which will be described in the following).

Note that a destructive approach for the robot can also easily be integrated into a destructive approach for the whole job-shop problem taking into account the machines and the robot individually.

In the following we will describe an integer linear programming formulation for the robot problem where the precedence constraints are decomposed into chains. The decision variables correspond to single schedules for each chain. By relaxing the integrality constraints we get a linear program in which different schedules for each chain may be mixed (in different fractions). Since this linear program contains an exponential number of variables, we propose a column generation approach.

We assume that a decomposition of the precedence constraints C into γ chains is given where for $\varrho = 1, \ldots, \gamma$ the chain Γ_ϱ has the form $\varrho_1 \to \varrho_2 \to \ldots \to \varrho_{n_\varrho}$ with chain length n_ϱ. A natural way for such a decomposition into chains is the decomposition which is induced by the job-shop structure, (i.e. chain Γ_ϱ consists of the $n_\varrho - 1$ transport operations belonging to job $\varrho \in \{1, \ldots, N\}$). Furthermore, we may have some additional precedence constraints between operations belonging to different chains (induced by fixed machine disjunctions).

Obviously, a feasible robot schedule decomposes into corresponding "chain-schedules" for all operations belonging to the same chain. These schedules will be the basic elements of the approach. A schedule for chain Γ_ϱ (described by the completion times of the operations) will be called a **feasible chain-schedule** if it

- respects the time windows $[r_j, d_j]$ of all operations j belonging to chain Γ_ϱ,
- respects all precedences $\varrho_1 \to \varrho_2 \to \ldots \to \varrho_{n_\varrho}$ in chain Γ_ϱ, and
- respects the time-lags $l_{\varrho_k,\varrho_{k+1}}$ between consecutive operations ϱ_k, ϱ_{k+1} belonging to chain Γ_ϱ.

Note that as in the relaxation on page 231 time-lags l_{ij} between non-consecutive operations i, j are relaxed. We denote by $\mathcal{S}_\varrho$ the set of all feasible chain-schedules σ for chain Γ_ϱ and define for each schedule $\sigma \in \mathcal{S}_\varrho$ binary coefficients

$$a_{j\sigma t} = \begin{cases} 1, & \text{if operation } j \text{ finishes at time } t \text{ in } \sigma \\ 0, & \text{otherwise} \end{cases}$$

$$b_{\sigma t} = \begin{cases} 1, & \text{if an operation is processed in } \sigma \text{ in the interval } [t-1, t] \\ 0, & \text{otherwise} \end{cases}$$

for all operations j belonging to chain Γ_ϱ and $t = 1, \ldots, T$.

Furthermore, we introduce binary variables for every chain-schedule $\sigma \in \mathcal{S}_\varrho$ $(\varrho = 1, \ldots, \gamma)$

$$x_\sigma = \begin{cases} 1, & \text{if chain } \Gamma_\varrho \text{ is scheduled according to schedule } \sigma \in \mathcal{S}_\varrho \\ 0, & \text{otherwise.} \end{cases}$$

For each operation $j \in V$ let $\Gamma(j)$ be the chain to which j belongs. Then the feasibility problem for the robot with threshold T may be formulated as a 0-1-program as follows.

$$\sum_{\sigma \in \mathcal{S}_\varrho} x_\sigma = 1 \qquad (\varrho = 1, \ldots, \gamma) \quad (4.39)$$

$$\sum_{\varrho=1}^{\gamma} \sum_{\sigma \in \mathcal{S}_\varrho} b_{\sigma t} x_\sigma \leq 1 \qquad (t = 1, \ldots, T) \quad (4.40)$$

$$\sum_{\sigma \in \mathcal{S}_{\Gamma(j)}} \sum_{t=r_j+p_j}^{d_j} t\, a_{j\sigma t} x_\sigma - \sum_{\sigma \in \mathcal{S}_{\Gamma(i)}} \sum_{t=r_i+p_i}^{d_i} t\, a_{i\sigma t} x_\sigma \geq l_{ij} + p_j \quad (i \to j \in \tilde{C}) \quad (4.41)$$

$$\sum_{\sigma \in \mathcal{S}_{\Gamma(i)}} a_{i\sigma t} x_\sigma + \sum_{\sigma \in \mathcal{S}_{\Gamma(j)}} \sum_{\tau \in I_t^{i,j}} a_{j\sigma\tau} x_\sigma \leq 1 \quad \begin{matrix} (i - j \in D; \\ t = r_i + p_i, \ldots, d_i) \end{matrix} \quad (4.42)$$

$$x_\sigma \in \{0, 1\} \qquad (\sigma \in \bigcup_{\varrho=1}^{\gamma} \mathcal{S}_\varrho) \quad (4.43)$$

Due to the equalities (4.39) exactly one schedule $\sigma \in \mathcal{S}_\varrho$ is chosen for each chain $\Gamma_\varrho\,(\varrho = 1, \ldots, \gamma)$. The capacity conditions (4.40) ensure that in every time period $[t-1, t]$ at most one operation is scheduled. Constraints (4.41) take care of the minimal time-lags by requiring $C_j - C_i \geq l_{ij} + p_j$ for all $i \xrightarrow{l_{ij}} j \in C$. Since the minimal time-lags between consecutive operations of the same chain are already satisfied by definition in a feasible chain-schedule $\sigma \in \mathcal{S}_\varrho$, in (4.41) it is sufficient to consider only constraints corresponding to the reduced set

$$\tilde{C} := \{i \to j \in C \mid i, j \text{ are not consecutive operations in a chain}\}.$$

The weighted disjunctions $i - j \in D$ are taken into account by conditions (4.42) as follows. Assuming that operation i finishes at time $t \in [r_i + p_i, d_i]$, we have two possibilities for scheduling operation j (cf. Figure 4.21):

- If j is scheduled after i, the earliest completion time of j is $C_j = t + s_{ij} + p_j$,
- if j is scheduled before i, the latest completion time of j is $C_j = t - p_i - s_{ji}$.

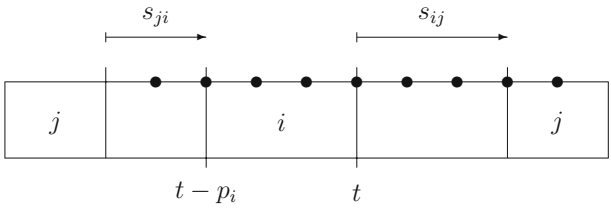


Figure 4.21: Two possibilities for scheduling operations i, j

Thus, operation j cannot be finished in the interval

$$I_t^{i,j} := [t - p_i - s_{ji} + 1, t + s_{ij} + p_j - 1] \cap [r_j + p_j, d_j].$$

If we have $a_{i\sigma t} = 1$, which means that i finishes at time t in σ, all possible completion times of operation j in the interval $I_t^{i,j}$ (as indicated by the black cycles in the figure) are blocked by constraints (4.42).

In order to reduce the number of constraints some superfluous ones may be omitted:

- The constraints in (4.40) only have to be considered for intervals $[t-1, t]$ in which at least two operations may be processed, i.e. for all t with $|\{j \mid [r_j, d_j] \cap [t-1, t] \neq \emptyset\}| \geq 2$. Since in all other time periods at most one operation may be processed, the corresponding capacity constraints are automatically fulfilled for each collection of feasible chain-schedules.
- Besides the constraints corresponding to consecutive operations in a chain all constraints in (4.41) can be dropped which correspond to time-lags induced by transitivity (i.e. all $i \xrightarrow{l_{ij}} j \in \tilde{C}$ where an operation h exists with $l_{ij} = l_{ih} + p_h + l_{hj}$).

- In the disjunctive constraints (4.42) it is sufficient to cover each disjunction $i - j \in D$ once: either for operation i as (i, j) for all $t \in [r_i + p_i, d_i]$ or for operation j as (j, i) for all $t \in [r_j + p_j, d_j]$. In order to have a smaller number of constraints, we may choose the operation with the smaller time window. Let $\tilde{D}$ be the set of all ordered pairs (i, j) with $i - j \in D$ which are considered in (4.42) for $t \in [r_i + p_i, d_i]$.

  Furthermore, we may omit all constraints for combinations $(i, j) \in \tilde{D}$ with time periods $t \in [r_i + p_i, d_i]$ for which the interval $I_t^{i,j}$ is empty. This case occurs if no time periods which are blocked by operation i finishing at time t are contained in the time window $[r_j + p_j, d_j]$ of operation j.

For the lower bound calculation we consider the fractional relaxation of (4.39) to (4.43). Since each variable x_σ is already bounded by 1 due to (4.39), it is sufficient to replace the 0-1-constraints (4.43) by the condition

$$x_\sigma \geq 0 \qquad \text{for all } \sigma \in \bigcup_{\varrho=1}^{\gamma} \mathcal{S}_\varrho.$$

If this relaxation has no feasible fractional solution, then $T + 1$ is a valid lower bound for the robot problem. The largest bound obtained by this relaxation will be called the **chain-packing bound**.

Note that if we decompose the precedence constraints into single nodes instead of chains and cover all conjunctions by conditions (4.41), the packing formulation (4.39) to (4.43) is equivalent to a formulation based on so-called **time-indexed variables** (cf. the LP-formulation of the RCPSP in Section 2.3.4). These binary variables are defined for all operations $j \in V$ and all time periods $t = 1, \ldots, T$ by

$$x_{jt} = \begin{cases} 1, & \text{if operation } j \text{ finishes at time } t \\ 0, & \text{otherwise.} \end{cases}$$

It can be shown that the chain-packing bound is at least as strong as the bound based on these time-indexed variables, since each feasible fractional chain-packing schedule induces a feasible fractional time-indexed solution

$$x_{jt} := \sum_{\sigma \in \mathcal{S}_{\Gamma(j)}} a_{j\sigma t} x_\sigma,$$

where x_{jt} is the sum of portions of operation j which finish at time t in the chain-packing schedule.

Thus, the lower bound based on the time-indexed variables cannot be larger than the chain-packing bound. In the following example it can be seen that the chain-packing bound may produce better bounds than the time-indexed relaxation.

Example 4.6: Consider the example with 7 operations, 3 chains and unit processing times shown in Figure 4.22. We consider the feasibility problem for

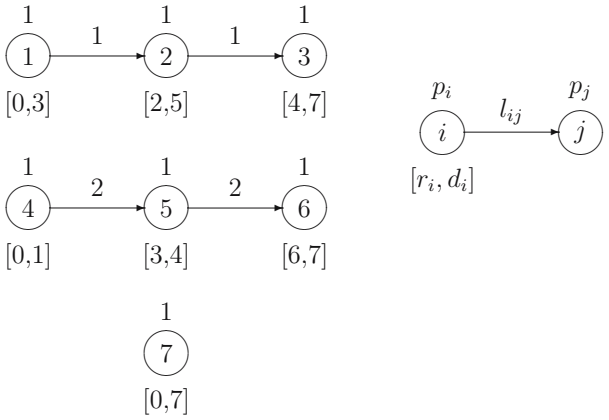
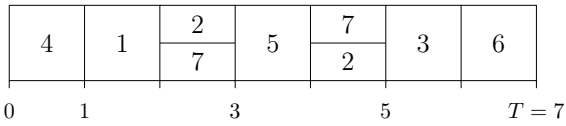


Figure 4.22: Example for the chain-packing bound

the threshold value $T = 7$. In each feasible chain-packing schedule (fractional or non-fractional) the chain consisting of operations 4,5 and 6 has to be planned non-fractionally in the time windows [0,1], [3,4] and [6,7]. Obviously, no feasible chain-schedule exists for the chain $1 \rightarrow 2 \rightarrow 3$ which is compatible with the other chain, i.e. the feasibility problem does not have a solution. On the other hand, consider the fractional time-indexed solution

$$x_{41} = x_{54} = x_{67} = 1, x_{12} = x_{36} = 1, x_{23} = x_{25} = x_{73} = x_{75} = 0.5.$$



It can easily be checked that the two conditions for the conjunctions $1 \rightarrow 2$ and $2 \rightarrow 3$ in (4.41) are satisified by this solution. For example, for the conjunction $1 \rightarrow 2$ we have

$$\sum_{t=r_2+p_2}^{d_2} t\, x_{2t} - \sum_{t=r_1+p_1}^{d_1} t\, x_{1t} = 3x_{23} + 5x_{25} - 2x_{12} = 2 \geq l_{12} + p_2 = 2,$$

the condition for $2 \rightarrow 3$ can be verified analogously. Hence this solution is feasible for the fractional time-indexed relaxation. □

In the following we describe how the chain-packing bound can be calculated in an efficient way. By introducing two additional variables, λ_1, λ_2, the fractional relaxation of the feasibility problem (4.39) to (4.43) can be formulated as a linear program:

$$\min \quad \lambda_1 + \lambda_2 \tag{4.44}$$

s.t.

$$\sum_{\sigma \in \mathcal{S}_\varrho} x_\sigma = 1 \qquad (\varrho = 1, \ldots, \gamma) \tag{4.45}$$

$$\lambda_1 - \sum_{\varrho=1}^{\gamma} \sum_{\sigma \in \mathcal{S}_\varrho} b_{\sigma t} x_\sigma \geq -1 \qquad (t = 1, \ldots, T) \tag{4.46}$$

$$\sum_{\sigma \in \mathcal{S}_{\Gamma(j)}} \sum_{t=r_j+p_j}^{d_j} t\, a_{j\sigma t} x_\sigma - \sum_{\sigma \in \mathcal{S}_{\Gamma(i)}} \sum_{t=r_i+p_i}^{d_i} t\, a_{i\sigma t} x_\sigma \geq l_{ij} + p_j \quad (i \to j \in \tilde{C}) \tag{4.47}$$

$$\lambda_2 - \sum_{\sigma \in \mathcal{S}_{\Gamma(i)}} a_{i\sigma t} x_\sigma - \sum_{\sigma \in \mathcal{S}_{\Gamma(j)}} \sum_{\tau \in I_t^{i,j}} a_{j\sigma\tau} x_\sigma \geq -1 \quad \begin{array}{l}((i,j) \in \tilde{D};\\ t = r_i + p_i, \ldots, d_i)\end{array} \tag{4.48}$$

$$x_\sigma \geq 0 \qquad (\sigma \in \bigcup_{\varrho=1}^{\gamma} \mathcal{S}_\varrho) \tag{4.49}$$

$$\lambda_1, \lambda_2 \geq 0. \tag{4.50}$$

In this formulation the capacity of the machine and the weighted disjunctions may be violated, which is compensated by the two "penalty variables", λ_1 and λ_2. Note that we do not need an additional variable λ_3 in constraints (4.47), since it is easy to determine solutions for the subproblem described by conditions (4.45) and (4.47). In this subproblem we only have to calculate feasible chain-schedules respecting all minimal time-lags l_{ij}, which can be done as follows. For each operation j the length $\ell(j)$ of a longest path from 0 to j in the precedence graph is determined taking into account all processing times and minimal time-lags of its predecessors (cf. the construction of a feasible schedule defined by a complete selection in the disjunctive graph model in Section 4.6.3).

By scheduling all operations as early as possible at time $\ell(j)$ we get a schedule in which all time-lags are respected, but the machine capacity and some disjunctions may be violated. If we decompose this schedule into a collection of γ chain-schedules, we obtain a solution which is feasible according to conditions (4.47).

This means that the vector x in a feasible solution $(x, \lambda_1, \lambda_2)$ for (4.45) to (4.50) only has to fulfill all minimal time-lags l_{ij} according to (4.41); conditions (4.40) and (4.42) may be violated. Then the variable λ_1 in (4.46) denotes the maximal violation of the load of the machine, the variable λ_2 in (4.48) denotes the maximal violation of a weighted disjunction. A feasible solution for the fractional relaxation of (4.39) to (4.43) exists if and only if the linear program (4.44) to (4.50) has an optimal solution value with $\lambda_1^* = 0$ and $\lambda_2^* = 0$, i.e. $\lambda_1^* + \lambda_2^* = 0$.

Unfortunately, the number of variables in this LP-formulation grows exponentially with the number of operations. However, as described in Section 2.3.5, we can work with only a few of them at a time and only generate new feasible

sets when they are really needed. In the following we apply such a **delayed column generation approach** and describe the problem-specific details.

For this purpose the corresponding dual problem to the primal problem (4.44) to (4.50) is required. If we denote the dual variables corresponding to constraints (4.45) by z_ϱ, those corresponding to (4.46) by y_t, and the variables belonging to conditions (4.47) and (4.48) by $y_{i\to j}$ and $y_{(i,j),t}$, respectively, the dual linear program has the form

$$\max \quad \sum_{\varrho=1}^{\gamma} z_\varrho - \sum_{t=1}^{T} y_t + \sum_{i\to j\in\tilde{C}} (l_{ij}+p_j) y_{i\to j} - \sum_{(i,j)\in\tilde{D}} \sum_{t=r_i+p_i}^{d_i} y_{(i,j),t} \tag{4.51}$$

s.t.

$$\sum_{t=1}^{T} y_t \quad \leq \quad 1 \tag{4.52}$$

$$\sum_{(i,j)\in\tilde{D}} \sum_{t=r_i+p_i}^{d_i} y_{(i,j),t} \quad \leq \quad 1 \tag{4.53}$$

$$\begin{aligned} z_\varrho - \sum_{t=1}^{T} b_{\sigma t} y_t + \sum_{\substack{i\in\Gamma_\varrho \\ \{j|j\to i\in\tilde{C}\}}} \sum_{t=r_i+p_i}^{d_i} t\, a_{i\sigma t} y_{j\to i} - \sum_{\substack{i\in\Gamma_\varrho \\ \{j|i\to j\in\tilde{C}\}}} \sum_{t=r_i+p_i}^{d_i} t\, a_{i\sigma t} y_{i\to j} \\ - \sum_{\substack{i\in\Gamma_\varrho \\ \{j|(i,j)\in\tilde{D}\}}} \sum_{t=r_i+p_i}^{d_i} a_{i\sigma t} y_{(i,j),t} - \sum_{\substack{i\in\Gamma_\varrho \\ \{j|(j,i)\in\tilde{D}\}}} \sum_{t=r_j+p_j}^{d_j} \sum_{\tau\in I_t^{j,i}} a_{i\sigma\tau} y_{(j,i),t} \leq 0 \\ (\varrho = 1,\ldots,\gamma;\ \sigma\in\mathcal{S}_\varrho) \end{aligned} \tag{4.54}$$

$$\begin{aligned} y_t \quad &\geq \quad 0 \qquad (t=1,\ldots,T) && (4.55)\\ y_{i\to j} \quad &\geq \quad 0 \qquad (i\to j\in\tilde{C}) && (4.56)\\ y_{(i,j),t} \quad &\geq \quad 0 \qquad ((i,j)\in\tilde{D};\ t=r_i+p_i,\ldots,d_i) && (4.57)\\ z_\varrho \quad &\in \quad \mathbb{R} \qquad (\varrho=1,\ldots,\gamma) && (4.58) \end{aligned}$$

Conditions (4.52) and (4.53) are the dual constraints corresponding to the dual variables λ_1 and λ_2, respectively. In order to derive (4.54), we must collect for each x_σ the correct terms in the corresponding column. For example, for each fixed $i\in\Gamma_\varrho$ we have to consider (4.47) for all j with $i\to j\in C$ and all j with $j\to i\in C$ (the same for disjunctions $(i,j),(j,i)\in\tilde{D}$ in (4.48)).

In each iteration of the revised simplex method we work with a restricted set of columns which contains a basis, the variables λ_1,λ_2 and all slack variables according to inequalities (4.46) to (4.48). Initially, for each chain $\Gamma_\varrho\,(\varrho=1,\ldots,\gamma)$ we calculate a chain-schedule $\sigma^{(\varrho)}\in\mathcal{S}_\varrho$ which is compatible with the other chain-schedules with respect to the precedences and the minimal time-lags l_{ij}

according to (4.47). As previously described, this can be done by longest path calculations in the precedence graph and scheduling each operation as early as possible. We set $x_{\sigma^{(\varrho)}} = 1$ for all these γ chain-schedules $\sigma^{(\varrho)}$ and determine the maximal violations λ_1 and λ_2 in inequalities (4.46) and (4.48). It is easy to see that we obtain a feasible basic non-fractional starting solution for (4.45) to (4.50) if we choose the γ variables $x_{\sigma^{(\varrho)}}$, the penalty variables λ_1, λ_2 and slack variables for the remaining inequalities in (4.46) to (4.48) as basic variables.

In the general step of the procedure the linear program is solved to optimality with the current set of columns. Then the multipliers $z_\varrho, y_t, y_{i\to j}, y_{(i,j),t}$ which correspond to the current basis are calculated. If these values satisfy all constraints (4.52) to (4.58), the current value $\lambda_1 + \lambda_2$ is optimal. In the case $\lambda_1 + \lambda_2 > 0$ we state infeasibility, otherwise the fractional relaxation has a feasible solution. If the multipliers do not satisfy all dual constraints (4.52) to (4.58), only condition (4.54) may be violated for a column which is not in the current working set. The other conditions must be fulfilled since λ_1, λ_2 and all slack variables according to (4.46) to (4.48) are kept in the working set and the corresponding linear program has been solved to optimality. This implies that condition (4.54) may only be violated for a column σ which is not contained in the current set of columns. Such a violated constraint (4.54) corresponds to a column σ which may be used as an entering column.

To use a column generation approach, calculating one such violated constraint should be possible without explicitly enumerating every non-basic column. A procedure for this so-called **pricing problem** will be described next. This problem has the nice property that it decomposes into γ subproblems which can be solved independently for each chain. Thus, we may consider all chains Γ_ϱ successively and may obtain more than one chain-schedule corresponding to an entering column in each step. Afterwards all these columns are inserted into the working set of current columns and the linear program is resolved to optimality. If the size of the working set exceeds a certain value, some non-basic columns are deleted.

For each chain-schedule $\sigma \in \mathcal{S}_\varrho$ $(\varrho = 1, \ldots, \gamma)$ we define the costs

$$\begin{aligned} c(\sigma) \;:=\; & \sum_{t=1}^{T} b_{\sigma t} y_t - \sum_{\substack{i\in\Gamma_\varrho \\ \{j \mid j\to i\in\tilde{C}\}}} \sum_{t=r_i+p_i}^{d_i} t\, a_{i\sigma t} y_{j\to i} + \sum_{\substack{i\in\Gamma_\varrho \\ \{j \mid i\to j\in\tilde{C}\}}} \sum_{t=r_i+p_i}^{d_i} t\, a_{i\sigma t} y_{i\to j} \\ & + \sum_{\substack{i\in\Gamma_\varrho \\ \{j \mid (i,j)\in\tilde{D}\}}} \sum_{t=r_i+p_i}^{d_i} a_{i\sigma t} y_{(i,j),t} + \sum_{\substack{i\in\Gamma_\varrho \\ \{j \mid (j,i)\in\tilde{D}\}}} \sum_{t=r_j+p_j}^{d_j} \sum_{\tau\in I_t^{j,i}} a_{i\sigma\tau} y_{(j,i),t}. \end{aligned} \tag{4.59}$$

If the costs $c(\sigma) - z_\varrho$ are negative, then for $\sigma \in \mathcal{S}_\varrho$ the dual restriction (4.54) is violated and the corresponding variable x_σ is eligible to enter the basis. To find such a column $\sigma \in \mathcal{S}_\varrho$ for chain Γ_ϱ or to prove that none exists, it is sufficient to calculate a chain-schedule $\sigma^* \in \mathcal{S}_\varrho$ with minimal costs, i.e. we determine a

schedule $\sigma^* \in \mathcal{S}_\varrho$ with $c(\sigma^*) = \min\{c(\sigma) \mid \sigma \in \mathcal{S}_\varrho\}$. If $c(\sigma^*) < z_\varrho$ holds, we have found an entering column, otherwise no improving column for chain Γ_ϱ exists.

We consider a chain-schedule $\sigma \in \mathcal{S}_\varrho$ and collect all contributions of an operation i of chain Γ_ϱ to the costs $c(\sigma)$ in (4.59). If operation i finishes at time $t' \in [r_i+p_i, d_i]$ in σ, we have $a_{i\sigma t} = 1$ only for $t = t'$ and $b_{\sigma t} = 1$ for all $t \in [t'-p_i+1, t']$. The contribution to the last term of (4.59) is given by the sum of all dual variables $y_{(j,i),t}$ for which t' is contained in the interval $I_t^{j,i}$. Since $t' \in I_t^{j,i}$ if and only if $t \in I_{t'}^{i,j}$, this sum is $\sum_{\{j|(j,i)\in\tilde{D}\}} \sum_{t\in I_{t'}^{i,j}} y_{(j,i),t}$. Summarizing all terms we get the following costs for operation i finishing at time t':

$$\begin{aligned} c(i,t') \;=\; & \sum_{t=t'-p_i+1}^{t'} y_t - t' \sum_{\{j|j\to i\in\tilde{C}\}} y_{j\to i} + t' \sum_{\{j|i\to j\in\tilde{C}\}} y_{i\to j} \\ & + \sum_{\{j|(i,j)\in\tilde{D}\}} y_{(i,j),t'} \quad + \sum_{\{j|(j,i)\in\tilde{D}\}} \sum_{t\in I_{t'}^{i,j}} y_{(j,i),t}. \end{aligned}$$

Hence, the costs $c(\sigma)$ for a chain-schedule σ with completion times C_i^σ are given by

$$c(\sigma) = \sum_{i\in\Gamma_\varrho} c(i, C_i^\sigma).$$

The pricing algorithm is based on dynamic programming and uses a forward recursion. For each chain Γ_ϱ we have to calculate a chain-schedule $\sigma^* \in \mathcal{S}_\varrho$ which minimizes $c(\sigma)$. In order to simplify the notation, in the following we assume that chain Γ_ϱ has the form $1 \to 2 \to \ldots \to n_\varrho$. Since the order of all operations i belonging to chain Γ_ϱ is fixed, the problem is to find completion times C_i^σ in the given time windows $[r_i + p_i, d_i]$ which respect the time-lags between consecutive operations and minimize the sum objective $c(\sigma) = \sum_{i\in\Gamma_\varrho} c(i, C_i^\sigma)$.

Let $F_k(t)$ be the minimal costs for all feasible chain-schedules $\sigma \in \mathcal{S}_\varrho$ consisting of the first k operations of chain Γ_ϱ, where operation k finishes not later than time t. For a chain-schedule realizing $F_k(t)$ there are two possibilities:

- either operation k finishes before time t, i.e. it finishes not later than time $t-1$, or
- operation k finishes exactly at time t.

In the first case the costs $F_k(t)$ are equal to $F_k(t-1)$. In the second case we have to select the best chain-schedule for the first $k-1$ operations where $k-1$ finishes not later than time $t - p_k - l_{k-1,k}$. Then the costs are equal to $F_{k-1}(t - p_k - l_{k-1,k}) + c(k,t)$. Since operation k cannot be finished before time $r_k + p_k$, the costs $F_k(t)$ are set to infinity for $t < r_k + p_k$. Furthermore, since operation k cannot be finished later than time d_k, we have $F_k(t) = F_k(d_k)$ for all $t > d_k$. These values do not have to be explicitly stored if in the second case

the minimum of the time $t - p_k - l_{k-1,k}$ with the corresponding deadline d_{k-1} is taken.

Initially, we have $F_1(t) = \infty$ for $t < r_1 + p_1$ since operation 1 cannot be finished before time r_1+p_1. For all $t \in [r_1+p_1, d_1]$ the value $F_1(t)$ is given by the minimum of $c(1,t)$ (corresponding to the situation that operation 1 finishes exactly at time t) and $F_1(t-1)$ (corresponding to the situation that operation 1 finishes before time t). Summarizing, the $F_k(t)$-values can be calculated recursively as follows.

Initialization:

$$F_1(t) := \begin{cases} \infty & , \quad t < r_1 + p_1 \\ \min\{F_1(t-1),\ c(1,t)\}, & t \in [r_1 + p_1, d_1] \end{cases}$$

Recursion:

$$F_k(t) = \begin{cases} \infty & , \ t < r_k + p_k \\ \min\{F_k(t-1), & \\ \quad F_{k-1}(\min\{t - p_k - l_{k-1,k}, d_{k-1}\}) + c(k,t)\}, & t \in [r_k + p_k, d_k] \end{cases}$$

$$\text{for } k = 2, \ldots, n_\varrho.$$

The optimal solution value $c(\sigma^*)$ is given by $F_{n_\varrho}(d_{n_\varrho})$. We have to calculate $F_k(t)$ for each of the n_ϱ operations k in chain Γ_ϱ and for all time periods $t \in [r_k + p_k, d_k]$. Since each computation can be done in constant time, the complexity of the dynamic programming procedure is $O(n_\varrho \Delta_\varrho^{\max}) = O(n_\varrho T)$ where $\Delta_\varrho^{\max} := \max_{k \in \Gamma_\varrho} \{d_k - p_k - r_k\}$ denotes the maximal length of a time window of an operation belonging to chain Γ_ϱ.

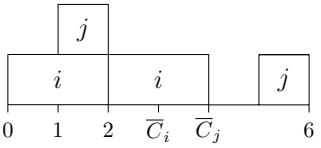
Finding a chain-schedule $\sigma^* \in \mathcal{S}_\varrho$ which realizes $F_{n_\varrho}(d_{n_\varrho})$ can be done by a backward construction. We start with time $t := d_{n_\varrho}$ and $k := n_\varrho$ as the last operation of chain Γ_ϱ. If $F_k(t) = F_k(t-1)$, then an optimal schedule exists where k finishes not later than time $t-1$. Then we reduce t by one and continue. Otherwise, $F_k(t) < F_k(t-1)$ and operation k finishes exactly at time t in an optimal chain-schedule. Then we replace t by $\min\{t - p_k - l_{k-1,k}, d_{k-1}\}$ and continue with the predecessor $k-1$. This backward construction finds an optimal schedule in $O(d_{n_\varrho} - r_1 + n_\varrho) = O(T + n_\varrho)$ steps since in each step we either reduce the time or move to the preceding operation.

The complexity of the dynamic programming procedure is the reason why we do not require that time-lags between non-consecutive operations also have to be satisfied in a feasible chain-schedule. If this condition had to be fulfilled, it would not be sufficient to store only the completion time t of the last scheduled operation k. Then we have to store all completion times $t_1, \ldots, t_{k-1}$ for the operations $1, \ldots, k-1$ in order to calculate the minimal feasible starting time of operation k which is given by the maximal value $t_i + l_{ik}$ of the operations $i \in \{1, \ldots, k-1\}$ with $i \to k \in C$. This extension increases the complexity of the dynamic programming procedure by the factor T which may lead to problems for larger T-values.

Finally, we consider an improvement of the chain-packing bound introduced before. A disadvantage of the previous formulation is that setups and time-lags between different chains are only weakly taken into account in the fractional relaxation. Therefore, we will strengthen the formulation by considering schedules for larger subgraphs than single chains. The structure of the resulting linear program is similar to the previous one, i.e. it can be solved with similar techniques.

The previous linear programming formulation is based on a decomposition of the precedence constraints C into single chains. While the time-lags between consecutive operations in these chains have to be respected in all considered schedules, the other time-lags and the setup times are only weakly taken into account in the fractional relaxation (by the inequalities (4.41) and (4.42)).

In order to get an impression how conjunctions are taken into account in the constraints of the fractional relaxation, we consider the following example with one conjunction $i \to j \in C$ and $p_i = 2, p_j = 1, l_{ij} = 0$. Let x be the fractional solution with $x_{\sigma_k} = 0.5$ for $k = 1, \ldots, 4$ in which the operations i and j are scheduled according to the schedules $\mathcal{S}^i := \{\sigma_1, \sigma_2\}$ and $\mathcal{S}^j := \{\sigma_3, \sigma_4\}$ where $C_i^{\sigma_1} = 2, C_i^{\sigma_2} = 4, C_j^{\sigma_3} = 2, C_j^{\sigma_4} = 6$:



Obviously, this solution x satisfies the constraints (4.39) and (4.40). Furthermore, it respects the constraint (4.41) for $i \to j \in C$ due to

$$\sum_{\sigma \in \mathcal{S}^j} C_j^\sigma x_\sigma - \sum_{\sigma \in \mathcal{S}^i} C_i^\sigma x_\sigma = 4 - 3 = 1 = l_{ij} + p_j.$$

At this example we see that the constraints (4.41) only ensure that the "average completion times" $\bar{C}_j := \sum_{\sigma \in \mathcal{S}_{\Gamma(j)}} C_j^\sigma x_\sigma$ satisfy the inequalities $\bar{C}_j - \bar{C}_i \geq l_{ij} + p_j$ for $i \to j \in C$. These inequalities can often easily be satisfied by solutions x in which several operations are split into many fractions (i.e. where we have a lot of small x_σ-values). Hence, the set of feasible solutions for the fractional relaxation is much larger than the set of feasible solutions for the 0-1-program and the corresponding bounds are weak.

In the following we show how the fractional relaxation can be strengthened by considering more complicated subgraphs instead of chains. For this purpose we partition the set C into λ subsets $\Lambda_1, \ldots, \Lambda_\lambda$, where each Λ_ϕ $(\phi = 1, \ldots, \lambda)$ consists of a fixed number q of chains $\Gamma_\varrho^{(\phi)} : \varrho_1^{(\phi)} \to \varrho_2^{(\phi)} \to \ldots \to \varrho_{n_\varrho^{(\phi)}}^{(\phi)}$ for $\varrho = 1, \ldots, q$. In order to obtain such a decomposition we may consider the first q chains as the first subset, the second q chains as the second subset, etc.

Anologously to the definition of a feasible chain-schedule, a schedule σ for a subset Λ_ϕ will be called a **feasible *q*-chain-schedule** if it

- respects the time windows $[r_j, d_j]$ of all operations j in the subset Λ_ϕ,
- respects all precedences $i \rightarrow j \in C$ between operations i, j belonging to subset Λ_ϕ (not only between operations belonging to the same chain, but also between operations belonging to different chains),
- respects all setups s_{ij} between operations $i, j \in \Lambda_\phi$, and
- respects all time-lags l_{ij} between operations $i, j \in \Lambda_\phi$ consecutively planned in σ.

Thus, we again relax time-lags l_{ij} between operations $i, j \in \Lambda_\phi$ which are not planned consecutively. The structure of the linear programming formulation (4.39) to (4.43) remains the same, only the constraints have to be modified. Especially, some inequalities in the linear program can be omitted since the feasibility constraints which are induced by these inequalities are already incorporated into the definition of feasible q-chain-schedules. In conditions (4.39) we have to choose a q-chain-schedule for each subset Λ_ϕ ($\phi = 1, \ldots, \lambda$). The capacity conditions (4.40) remain the same and constraints (4.41) are defined for all time-lags $i \xrightarrow{l_{ij}} j \in C$ which are not induced by transitivity. Note that contrary to the situation where only single chains are considered, conditions (4.41) are not automatically fulfilled for consecutive operations of the same chain in a feasible q-chain-schedule. Thus, these conditions are not superfluous. But, since in the definition of a feasible q-chain-schedule we require that all setup times between operations in the same subset Λ_ϕ are satisfied, constraints (4.42) corresponding to these disjunctions can be dropped. Especially, if we consider all chains together in one subset (i.e. we have $q = \gamma$), all disjunctive constraints can be dropped.

As before we consider the fractional relaxation of this formulation and call the largest bound obtained by this relaxation the ***q*-chain-packing bound**. This relaxation can again be solved by column generation techniques and an adapted dynamic programming algorithm to solve the pricing problem.

4.6.6 Heuristic methods

In this subsection we describe foundations for heuristic algorithms for the job-shop problem with transportation where only a single robot is available for all transportations. Especially, we derive some problem-specific properties which are useful to define appropriate neighborhoods for local search algorithms.

As described in Section 4.6.3, we represent feasible solutions by complete consistent selections $\mathcal{S} = \mathcal{S}_M \cup \mathcal{S}_R$ in the disjunctive graph where all machine and robot disjunctions are fixed such that the resulting graph $G(\mathcal{S}) = (V, C \cup \mathcal{S})$

is acyclic. We consider the situation in which a complete selection $\mathcal{S}$ and the corresponding earliest start schedule S with makespan $C_{\max}(S)$ are given and we want to improve this solution. In order to define appropriate neighborhood structures we use a block approach like in Section 4.2. With the definition of blocks it can be stated that only certain changes of a schedule S may have a chance to improve the current makespan $C_{\max}(S)$ and in the search process only such solutions will be considered as candidates for neighbor solutions.

Let P^S be a critical path in $G(\mathcal{S})$. A sequence $u_1, \ldots, u_k$ of at least two successive nodes (i.e. $k > 1$) on P^S is called

- a **machine block** if the operations $u_1, \ldots, u_k$ are sequenced consecutively on the same machine, and enlarging the subsequence by one operation leads to a subsequence which does not fulfill this property,
- a **robot block** if the operations $u_1, \ldots, u_k$ are transport operations which are sequenced consecutively on the robot, no conjunction exists between consecutive operations and enlarging the subsequence by one operation would violate one of the previous properties.

For example, in the schedule shown in Figure 4.18 for Example 4.4 the path $O_{13} \to T_{13} \xrightarrow{t'_{12}} T_{12} \xrightarrow{t'_{32}} T_{21} \to O_{31} \to O_{33}$ is a critical path with the machine block (O_{31}, O_{33}) on M_3 and the robot block (T_{13}, T_{12}, T_{21}). Another critical path is $O_{11} \to T_{11} \to O_{21} \to O_{12} \to T_{12} \xrightarrow{t'_{32}} T_{21} \to O_{31} \to O_{33}$ with the machine blocks (O_{21}, O_{12}) on M_2, (O_{31}, O_{33}) on M_3 and the robot block (T_{12}, T_{21}).

The following theorem is a generalization of Theorem 4.2 for the classical job-shop problem. Note that in the proof the triangle inequality (4.32) for the empty moving times is necessary.

Theorem 4.7 Let $\mathcal{S}$ be a complete selection with makespan $C_{\max}(S)$ and let P^S be a critical path in $G(\mathcal{S})$. If another complete selection $\mathcal{S}'$ with $C_{\max}(S') < C_{\max}(S)$ exists, then in $\mathcal{S}'$

- at least one operation of some machine block on P^S has to be processed before the first or after the last operation of the block, or
- at least two transport operations of a robot block on P^S are processed in the opposite order.

Based on this result we will introduce two approaches: an one-stage and a two-stage approach.

An one-stage approach

At first we present an one-stage approach for the job-shop problem with a single transport robot. The search space is the set of all complete consistent selections

and in each iteration a neighbor is generated either by moving an operation on a machine or by moving a transport operation on the robot to another position.

Theorem 4.7 gives necessary conditions for improving a given solution. Thus, a neighborhood $\mathcal{N}_1$ may be defined as follows. For a given selection $\mathcal{S} = \mathcal{S}_M \cup \mathcal{S}_R$ and a critical path P^S neighborhood $\mathcal{N}_1(\mathcal{S})$ contains all feasible selections $\mathcal{S}' = \mathcal{S}'_M \cup \mathcal{S}'_R$ which can be constructed as follows:

- in $\mathcal{S}'_M$ an operation of a machine block on P^S is moved to the beginning or the end of the block, or
- in $\mathcal{S}'_R$ two successive transport operations of a robot block on P^S are interchanged.

Another (larger) neighborhood $\mathcal{N}_2$ may be defined as for the job-shop problem by interchanging successive operations on a critical path (cf. the neighborhood $\mathcal{N}_{ca}$ in Section 4.2). This neighborhood considers also the interchange of successive operations in the inner part of machine blocks although due to Theorem 4.7 such moves cannot result in improving solutions. However, these extra moves ensure that $\mathcal{N}_2$ is opt-connected, i.e. from an arbitrary selection it is possible to reach a globally optimal selection via a sequence of steps in $\mathcal{N}_2$. For this proof, we first show a property of successive operations within blocks on a critical path.

Property 4.3 If i and j are two successive operations (ordinary or transport) within a block of a critical path belonging to a complete selection $\mathcal{S}$, then in $G(\mathcal{S})$ besides the arc (i, j) no further path exists from i to j.

Proof: Assume to the contrary that in $G(\mathcal{S})$ a path $P = (i, u_1, \ldots, u_k, j)$; $k \geq 1$ with length ℓ exists.

- Case 1: i and j belong to a machine block.
  Since all vertices are weighted with processing times greater than zero, we get $\ell \geq p_i + p_{u_1} > p_i$, which contradicts the fact that (i, j) is an arc of a critical path in $G(\mathcal{S})$.
- Case 2: i and j belong to a robot block.
  In this case i and j are two transport operations which are processed consecutively by the robot. As a consequence, all vertices $u_1, \ldots, u_k$ have to belong to operations which are processed on the same machine (say M_l) and, thus, transport operation i has to be a transport to machine M_l and transport operation j has to be a transport away from machine M_l. This implies that the length ℓ' of the path consisting only of the arc (i, j) is equal to $\ell' = p_i + t'_{ll} = p_i$ due to $t'_{ll} = 0$. Thus, $\ell \geq p_i + p_{u_1} > \ell'$, which contradicts the fact that (i, j) is an arc of a critical path in $G(\mathcal{S})$. □

This property implies that interchanging two successive operations i, j within a block always leads to a feasible selection (a cycle containing the arc (j, i) would induce a path from i to j which is different from the arc (i, j)). Using this property we can prove

Theorem 4.8 Neighborhood $\mathcal{N}_2$ is opt-connected.

Proof: Let $\mathcal{S}$ be an arbitrary non-optimal complete selection and $\mathcal{S}^*$ an optimal selection. Due to Theorem 4.7 we know that in $\mathcal{S}$ for at least two operations of a block (machine block or robot block) on a critical path in $G(\mathcal{S})$ the corresponding disjunction is fixed in the opposite direction compared with $\mathcal{S}^*$. Consequently, also two successive operations of a block with this property exist. Since due to Property 4.3 their exchange leads to a feasible selection, by one step in $\mathcal{N}_2$ we can achieve a solution which has one more disjunction fixed in the same way as in $\mathcal{S}^*$. Iteratively applying this step leads to a globally optimal solution by a sequence of steps in the neighborhood $\mathcal{N}_2$. □

A two-stage approach

In the following we describe another approach which is based on Theorem 4.7, but which tries to deal with the different situations on the machines and the robot more individually. This algorithm is organized in two stages. While in in the outer stage an operation on a machine is moved to another position, in the inner stage the robot is optimized according to the new machine selection.

In the one-stage approach in each iteration a neighbor solution $\mathcal{S}'$ is constructed in which either the machine selection $\mathcal{S}_M$ or the robot selection $\mathcal{S}_R$ is changed, but not both. Since the robot corresponds to a bottleneck machine, the old robot selection $\mathcal{S}_R$ is often a bad solution for the new machine selection $\mathcal{S}'_M$ or $\mathcal{S}_R$ is even infeasible with respect to $\mathcal{S}'_M$ (i.e. the graph $G(\mathcal{S}')$ contains a cycle). To overcome this disadvantage we may proceed in two stages, i.e. we define a neighborhood where after changing the machine selection the robot selection is adapted before the makespan of the new solution is determined.

This means that in the second stage the robot problem from page 226 has to be considered where all machine orders are fixed. We are interested in a robot selection respecting all precedences induced by the fixed machine orders and having a minimal makespan among all robot orders which are compatible with the given machine selection.

Based on such a hierarchical decomposition of the considered problem a neighborhood may also be defined in a hierarchical way where first the machine selection is changed and then the robot selection is adapted. For a given selection $\mathcal{S} = \mathcal{S}_M \cup \mathcal{S}_R$ neighborhood $\mathcal{N}_3(\mathcal{S})$ contains all feasible selections $\mathcal{S}' = \mathcal{S}'_M \cup \mathcal{S}'_R$ which can be constructed as follows:

- in $\mathcal{S}'_M$ one operation of a machine block on P^S is moved to the beginning or to the end of the block, and
- $\mathcal{S}'_R$ is obtained from $\mathcal{S}_R$ by calculating a heuristic solution for the robot problem associated with the new machine selection $\mathcal{S}'_M$.

Of course, also combinations of the one-stage and the two-stage approach are possible (in order to intensify and diversify the search process).

4.7 Job-Shop Problems with Limited Buffers

In this section we study job-shop problems with limited buffers between the machines. In such a model blocking on a machine may occur if not sufficient buffer space is available. In the following we will introduce different buffer models and discuss foundations for the representation of solutions.

4.7.1 Problem formulation

The job-shop problem with buffers of limited capacity is a generalization of the job-shop problem with blocking. It may be formulated as follows. We have a job-shop problem with operations $i = 1, \ldots, n$, a dummy start operation 0, a dummy terminating operation $n+1$, and machines $M_1, \ldots, M_m$. As before, for each operation $i = 1, \ldots, n$ the successor operation of i is denoted by $\sigma(i)$, $J(i)$ denotes the job to which i belongs, and $\mu(i)$ is the machine on which i must be processed for p_i time units.

Furthermore, we have q buffers $B_1, \ldots, B_q$, where B_k can store at most b_k jobs. When operation i finishes processing on machine $\mu(i)$, its successor operation $\sigma(i)$ may directly start on the next machine $\mu(\sigma(i))$ if this machine is not occupied by another job. Otherwise, job $J(i)$ is stored in a specified buffer $\beta(i) \in \{B_1, \ldots, B_q\}$ associated with operation i.

However, it may happen that $\mu(\sigma(i))$ is occupied and the buffer $\beta(i)$ is full. In this case job $J(i)$ has to stay on $\mu(i)$ until a job leaves buffer $\beta(i)$ or the job occupying $\mu(\sigma(i))$ moves to another machine. During this time job $J(i)$ blocks machine $\mu(i)$ for other jobs to be processed on $\mu(i)$.

We want to find a schedule satisfying the timing and blocking constraints which minimizes the makespan.

Depending on the buffer assignment $\beta(i)$ one can distinguish different buffer models:

- We call a buffer model a **general buffer model** if any assignment $\beta(i)$ to operation i is possible.
- If the assignment $\beta(i)$ depends on the job $J(i)$ to which i belongs, i.e. if each job has an own buffer, then we speak of **job-dependent buffers**.
- In a **pairwise buffer model** $\beta(i)$ depends on machines $\mu(i)$ and $\mu(\sigma(i))$. In this model a buffer B_{kl} is associated with each pair (M_k, M_l) of machines M_k and M_l. If $\mu(i) = M_k$ and $\mu(\sigma(i)) = M_l$, then job $J(i)$ has to use B_{kl} after leaving M_k (if necessary). A pairwise buffer model is usually used in connection with the flow-shop problem. Each job has to use $B_{k,k+1}$ when moving from M_k to M_{k+1} and machine M_{k+1} is still occupied.
- If the assignment $\beta(i)$ depends on the machine $\mu(i)$ on which operation i is processed, this type of buffer model is called **output buffer model**.

An output buffer B_k for machine M_k stores all jobs which leave machine M_k and cannot directly be loaded on the following machine.

- Symmetrically, if the assignment $\beta(i)$ depends on the machine $\mu(\sigma(i))$ on which the successor operation of i is processed, this type of buffer model is called **input buffer model**. An input buffer B_k for machine M_k stores all jobs which have been finished on their previous machine and cannot be loaded onto machine M_k directly.

In the following section we will discuss different ways to represent solutions within the general buffer model. These representations are useful for solution methods like branch-and-bound algorithms or local search heuristics. In later sections we will show how these representations can be specialized in connection with specific buffer models.

4.7.2 Representation of solutions

In the following we will show that the job-shop problem with general buffers can be reduced to the blocking job-shop problem without any buffers discussed in Section 4.4. For this purpose, we differentiate between b storage places within a buffer B with capacity $b > 0$. This means that the buffer B is divided into b so-called **buffer slots** $B^1, \ldots, B^b$ where the buffer slot B^l represents the l-th storing place of buffer B. Each buffer slot may be interpreted as additional blocking machine on which entering jobs have processing time zero. For each job one has to decide whether it uses a buffer on its route or it goes directly to the next machine. If job j uses a buffer, one has to assign a buffer slot to j. After these decisions and assignments we have to solve a problem which is equivalent to a blocking job-shop problem.

Because of the described reduction, a solution of a job-shop problem with general buffers can be represented by

1. sequences $\pi^1, \ldots, \pi^m$ of the operations on the machines $M_1, \ldots, M_k$,
2. a buffer slot assignment of each operation to a buffer slot of its corresponding buffer (where an operation may also not use any buffer), and
3. sequences of the jobs on the additional blocking machines (which correspond to buffer slot sequences).

The reduction of a job-shop problem with general buffers to a blocking job-shop problem implies that the buffer slot assignment is part of the solution representation. However, this way of solution representation has several disadvantages when designing fast solution procedures for the problem: Obviously, many buffer slot assignments exist which lead to very long schedules. For example, it is not meaningful to assign a large number of jobs to the same buffer slot when other buffer slots remain empty. Also there are many buffer slot assignments which are

symmetric to each other. It would be sufficient to choose one of them. Thus, we have the problem to identify balanced buffer slot assignments and to choose one representative buffer slot assignment among classes of symmetric assignments.

To overcome these deficits one may use a different solution representation from which buffer slot assignments can be calculated in polynomial time. The basic idea of this approach is to treat the buffer as one object and not as a collection of several slots.

For this purpose, we assign to each buffer B with capacity $b > 0$ two sequences: an input sequence π_{in} and an output sequence π_{out} containing all jobs assigned to buffer B. The input sequence π_{in} is a priority list for the jobs which either enter the buffer or go directly to the next machine. The output sequence π_{out} is a corresponding priority list for the jobs which leave buffer B or go directly to the next machine.

To represent a feasible (deadlock-free) schedule, the buffer sequences π_{in} and π_{out} must be compatible with the machine sequences. This means, that two jobs in π_{in} (π_{out}) which come from (go to) the same machine have to be in the same order in the buffer and the machine sequence. Additionally, the buffer sequences must be compatible with each other. Necessary conditions for mutual compatibility of π_{in} and π_{out} are given by the next theorem which also describes conditions under which jobs do not use the buffer.

Denote by $\pi_{in}(i)$ and $\pi_{out}(i)$ the job in the i-th position of the sequence π_{in} and π_{out}, respectively.

Theorem 4.9 Let B be a buffer with capacity $b > 0$, let π_{in} be an input sequence and π_{out} be an output sequence corresponding with a feasible schedule. Then the following conditions are satisfied:

(i) If $j = \pi_{out}(i) = \pi_{in}(i + b)$ for some position i, then job j does not enter the buffer and goes directly to the next machine.

(ii) For each position i we have $\pi_{out}(i) \in \{\pi_{in}(1), \ldots, \pi_{in}(i + b)\}$.

Proof:

(i) Let i be a position such that $j = \pi_{out}(i) = \pi_{in}(i + b)$ holds. At the time job j leaves its machine, $i + b - 1$ other jobs have entered buffer B and $i - 1$ jobs have left it. Thus, $(i + b - 1) - (i - 1) = b$ jobs different from j must be in the buffer. Therefore, buffer B is completely filled and job j must go directly to the next machine.

(ii) Assume to the contrary that $j = \pi_{out}(i) = \pi_{in}(i + b + k)$ for some $k \geq 1$. Similarly to (i) we can conclude that at the time job j leaves its machine, $i + b + k - 1$ other jobs have entered buffer B and $i - 1$ jobs different from j have left it. Thus, $(i + b + k - 1) - (i - 1) = b + k$ jobs different from j must be in the buffer. Since this exceeds the buffer capacity, the sequences π_{in} and π_{out} cannot correspond to a feasible schedule. □

From this theorem we may conclude that if we have a feasible schedule, then for each buffer B the corresponding sequences π_{in} and π_{out} must satisfy the condition

$$\pi_{out}(i) \in \{\pi_{in}(1), \ldots, \pi_{in}(i+b)\} \text{ for each position } i. \tag{4.60}$$

Conversely, if (4.60) holds, then we can find a valid buffer slot assignment by the algorithm shown in Figure 4.23. This algorithm scans both sequences π_{out} and π_{in} from the first to the last position and puts all jobs onto the next machine.

```
Algorithm Buffer Slot Assignment
 1.  WHILE π_out is not empty DO
 2.     Let j be the currently first job in π_out;
 3.     IF j is in the buffer B THEN
 4.        Move j to the next machine and delete j
           from B and π_out;
 5.     ELSE
 6.       IF j is currently in the first position of π_in THEN
 7.          Put j onto the next machine and delete j from
             π_in and π_out;
 8.       ELSE
 9.          IF the buffer B is not full THEN
10.             Delete the currently first element from π_in
                and put it into B;
11.          ELSE STOP;  /* π_in, π_out are not compatible */
12.  ENDWHILE
```

Figure 4.23: Buffer slot assignment

Example 4.7: Consider the input sequence $\pi_{in} = (1, 2, 3, 4, 5, 6)$ and the output sequence $\pi_{out} = (3, 2, 5, 4, 6, 1)$ for a buffer B with capacity $b = 2$. These sequences obviously satisfy condition (4.60).

We scan π_{out} from the first to the last position. Because $\pi_{out}(1) = 3$ is not at the beginning of π_{in}, we successively delete jobs 1 and 2 from π_{in} and assign them to buffer slots B^1 and B^2, respectively. Now, both buffer slots are occupied. Since afterwards job 3 is at the beginning of π_{in}, we put job 3 onto the next machine and delete it from π_{in} and π_{out}. The new first element $\pi_{out}(2) = 2$ is in the buffer, i.e. according to Step 4 we move job 2 to the next machine and delete it from B and π_{out}. Since the next job $\pi_{out}(3) = 5$ is currently not first in π_{in}, we eliminate job 4 from π_{in} and assign it to buffer slot B^2. Afterwards, job 5 is the first element in π_{in} and we put it onto the next machine and eliminate it from both π_{in} and π_{out}. Then, job $\pi_{out}(4) = 4$ is moved from the buffer to the next machine and deleted from B and π_{out}. Since job $\pi_{out}(5) = 6$ is not at the beginning of π_{in}, the currently first element 6 from π_{in} is put into the buffer slot B^2. Afterwards, according to Step 4 job 6 is moved to the next machine.

Finally, the last element $\pi_{out}(6) = 1$, which is in the buffer, is moved to the next machine and deleted from π_{out}. As a result of the algorithm job 1 is assigned to buffer slot B^1, while jobs 2, 4 and 6 are assigned to buffer slot B^2. □

To prove that the algorithm always provides a feasible buffer slot assignment if (4.60) holds, we have to show that Step 11 is never reached. The only possibility to reach Step 11 is when

- the currently first element j of π_{out} is not in the buffer,
- j is not the currently first element in π_{in}, and
- the buffer is full.

Let i be the position of job j in the original sequence π_{out}. But then according to the three conditions job j must be in a position greater than $i + b$ in π_{in}, which contradicts condition (4.60).

The buffer slot assignment procedure not only assigns jobs to buffer slots. It also defines a sequence of all jobs assigned to the same buffer slot. This buffer slot sequence is given by the order in which the jobs are assigned to the buffer slot. The order is induced by the buffer input sequence. In the previous example, the buffer slot sequence for B^1 is (1) and for B^2 it is $(2, 4, 6)$.

Let now S be an arbitrary feasible schedule for a job-shop problem with general buffers. This schedule defines sequences $\pi^1, \ldots, \pi^m$ for the machines $M_1, \ldots, M_m$ as well as sequences π^B_{in} and π^B_{out} for all buffers B with $b > 0$. If we apply to these buffer input and output sequences the buffer slot assignment procedure, we get a blocking job-shop problem (where the buffer slots correspond to additional blocking machines). S is also a feasible solution for this blocking job-shop. This shows that a solution of the job-shop problem with buffers can be represented by machine sequence $\pi^1, \ldots, \pi^m$ and for each buffer B with $b > 0$ an input sequence π^B_{in} and an output sequence π^B_{out}.

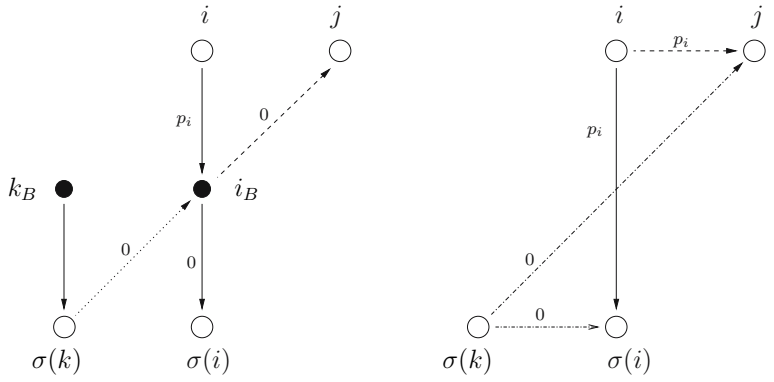
A corresponding schedule can be identified by longest path calculations in a directed graph G with machine operations and buffer-slot operations as vertices which is constructed in the following way:

- The set of vertices contains a vertex for each operation $i = 1, \ldots, n$ as well as two dummy vertices 0 and $n+1$. Additionally, for each operation i and each buffer B with $\beta(i) = B$ we have a **buffer-slot operation vertex** i_B if job $J(i)$ is assigned to the buffer by the buffer slot assignment procedure from above.
- We have the following arcs for each operation i where i is not the last operation of job $J(i)$ and i is not the last operation on machine $\mu(i)$: Associated with i and buffer B with $\beta(i) = B$ there is a direct arc $i \to \sigma(i)$ weighted with p_i if $J(i)$ is not assigned to the buffer. Furthermore, we have an arc $\sigma(i) \to j$ with weight 0 where j denotes the operation to be

processed immediately after operation i on $\mu(i)$. This arc ensures that operation j cannot start on $\mu(i)$ before the machine predecessor i has left $\mu(i)$.

If job $J(i)$ is assigned to buffer B, we introduce arcs connected with i and the buffer-slot operation vertex i_B as indicated in Figure 4.24(a). In this figure, j again denotes the operation to be processed immediately after operation i on $\mu(i)$. The buffer-slot operation k_B denotes the buffer-slot predecessor of i_B. If there is no such predecessor, the vertex i_B possesses only one incoming arc. The dotted arcs are called **blocking arcs**. The blocking arc $i_B \to j$ ensures that operation j cannot start on $\mu(i)$ before operation i has left $\mu(i)$ and the blocking arc $\sigma(k) \to i_B$ takes care that job $J(i)$ cannot enter the buffer slot before its buffer slot predecessor, which is job $J(k)$, has left the buffer slot.

- We have an arc $0 \to i$ for each first operation i of a job weighted with 0 and an arc $i \to n+1$ for each last operation i of a job weighted with p_i. Furthermore, if i is the last operation of job $J(i)$ but not the last operation on machine $\mu(i)$, there is an arc $i \to j$ weighted with p_i, where j denotes the operation to be processed immediately after i on $\mu(i)$.



(a) Buffer slot vertex i_B (b) Simplified graph

Figure 4.24: Buffer-slot vertices

This graph corresponds to the graph $G(\mathcal{S})$ introduced in Section 4.4 for a selection $\mathcal{S}$ for the job-shop problem with blocking operations where transitive arcs are left out.

If the graph G does not contain a positive cycle, let S_ν be the length of a longest path from 0 to vertex ν in G. Then the times S_ν describe a feasible schedule where S_i is the starting time of operation i and S_{i_B} indicates the time at which operation i is moved into buffer B.

If we are only interested in the starting times of the operations $i = 1, \ldots, n$ and not in the insertion times of jobs into buffers, then we can simplify G by eliminating all buffer-slot operations i_B as indicated in Figure 4.24(b). More specifically, after eliminating the buffer-slot operation i_B, the job arcs $(i, \sigma(i))$ with weight p_i and the blocking arcs $(\sigma(k), j), (\sigma(k), \sigma(i))$ with weight 0 and (i, j) with weight p_i are induced. We call the simplified graph $\overline{G}$ **solution graph**.

In the remaining subsections of this section we discuss the special buffer models introduced in Section 4.7.1.

4.7.3 Flow-shop problems with intermediate buffers

The natural way to introduce buffers in connection with the flow-shop problem is to introduce an intermediate buffer B_k between succeeding machines M_k and M_{k+1} for $k = 1, \ldots, m-1$. If π^k is the sequence of operations on machine M_k $(k = 1, \ldots, m)$, then obviously the input sequence for B_k is given by π^k and the output sequence must be π^{k+1}. Thus, the sequences $\pi^1, \ldots, \pi^m$ are sufficient to represent a solution in the case of a flow-shop with intermediate buffers.

Example 4.8: Consider a flow-shop problem with $m = 3$ machines, five jobs and two buffers with capacities $b_1 = 1$ and $b_2 = 2$.

Figure 4.25 shows a solution for the machine sequences $\pi^1 = (O_{11}, O_{12}, O_{13}, O_{14}, O_{15})$, $\pi^2 = (O_{22}, O_{21}, O_{23}, O_{25}, O_{24})$ and $\pi^3 = (O_{33}, O_{31}, O_{34}, O_{35}, O_{32})$. The numbers in the white circles denote the indices of the corresponding operations, whereas the black circles represent buffer slot operation vertices. The job chains of each job are shown vertically, where the positions of the black circles also indicate the corresponding buffer slot assignment. Figure 4.26 shows the resulting simplification after the buffer-slot vertices have been eliminated. □

It can be shown that buffer-slots can alway be assigned such that the simplified graph contains the following arcs (cf. also Figure 4.26):

- machine arcs $\pi^k(1) \to \pi^k(2) \to \ldots \to \pi^k(N)$ for $k = 1, \ldots, m$,
- job arcs $i \to \sigma(i)$ for all operations $i = 1, \ldots, n$, and
- buffer arcs $\pi^{k+1}(i) \to \pi^k(i + b_k + 1)$ for $i = 1, \ldots, N - b_k - 1$ and $k = 1, \ldots, m-1$

Due to condition (4.60) the machine sequences $\pi^1, \ldots, \pi^m$ are compatible if and only if for $k = 1, \ldots, m-1$ and each position $i = 1, \ldots, N - b_k$ the condition

$$\pi^{k+1}(i) \in \{\pi^k(1), \ldots, \pi^k(i + b_k)\} \tag{4.61}$$

holds. This is equivalent to the condition that the simplified graph contains no cycle. For each machine k condition (4.61) can be checked in $O(N)$ time. Thus, we can check in $O(Nm) = O(n)$ time whether the simplified graph contains no cycle. In this case a corresponding earliest start schedule can be calculated in $O(n)$ time because the simplified graph contains at most $O(n)$ arcs.

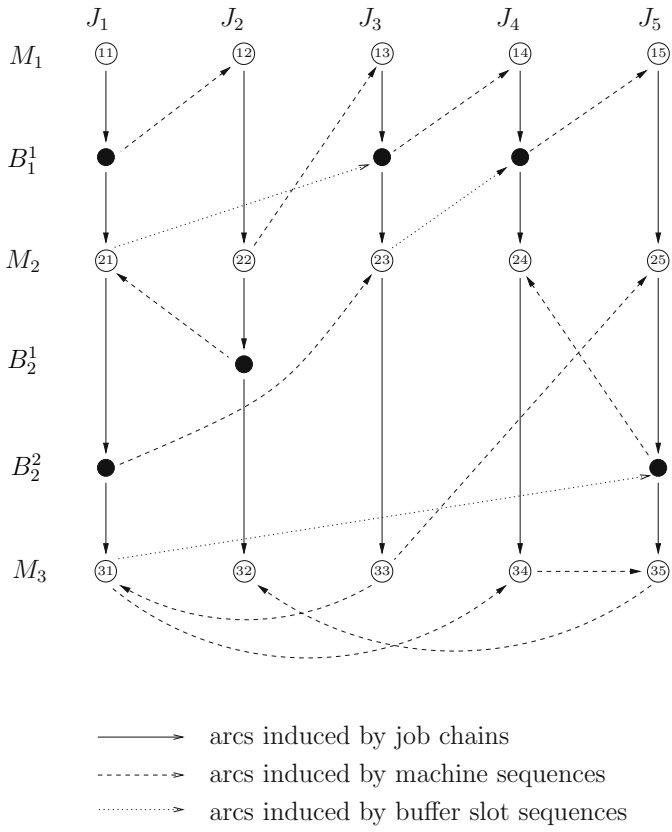


Figure 4.25: Buffer slot assignment and sequencing

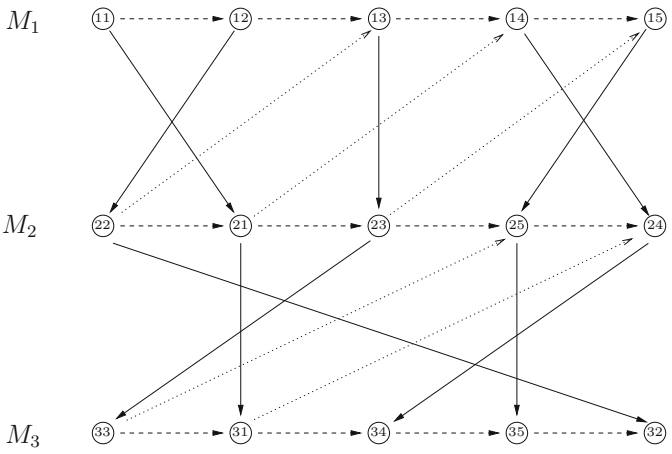


Figure 4.26: Simplified graph after eliminating the buffer-slot vertices

4.7.4 Job-shop problems with pairwise buffers

For the job-shop problem with pairwise buffers the situation is very similar to the situation for flow-shop problems with intermediate buffers. In this buffer model a buffer B_{kl} is associated with each pair (M_k, M_l) of machines. Each job which changes from M_k to M_l and needs storage has to use B_{kl}. The input sequence π_{in}^{kl} of buffer B_{kl} contains all jobs in π^k which move to M_l ordered in the same way as in π^k, i.e. π_{in}^{kl} is a partial sequence of π^k. Similarly, π_{out}^{kl} is the partial sequence of π^l consisting of the same jobs but ordered as in π^l. Using the subsequences π_{in}^{kl} and π_{out}^{kl} for each buffer we get a simplified graph $\overline{G}_{kl}$ (see Figure 4.26). The solution graph for given machine sequences $\pi^1, \ldots, \pi^m$ is a composition of all simplified graphs $\overline{G}_{kl}$.

However, for the job-shop problem with pairwise buffers conditions similar to (4.61) are not sufficient to guarantee that the solution graph is acyclic. Note that this property is not caused by the buffers since even in the case of the classical job-shop problem the solution graph may contain cycles (involving incompatible machine sequences $\pi^1, \ldots, \pi^m$). Furthermore, the solution graph may contain blocking cycles over several machines. Therefore, testing feasibility and calculating a schedule for sequences $\pi^1, \ldots, \pi^m$ is more time consuming (for example the Floyd-Warshall algorithm can be used which has running time $O(n^3)$ where n is the total number of operations).

4.7.5 Job-shop problems with output buffers

A further special type of buffers is that of output buffers. In this case, jobs leaving machine M_k are stored in a buffer B_k $(k = 1, \ldots, m)$ if the next machine is occupied and B_k is not full. As before let $\pi^1, \ldots, \pi^m$ be machine sequences, $\pi_{in}^1, \ldots, \pi_{in}^m$ be input sequences and $\pi_{out}^1, \ldots, \pi_{out}^m$ be output sequences which correspond to a solution of the problem.

Clearly, the buffer input sequence π_{in}^k of buffer B_k must be identical with the sequence π^k of the jobs on machine M_k $(k = 1, \ldots, m)$. Thus, for the buffers only the buffer output sequences $\pi_{out}^1, \ldots, \pi_{out}^m$ have to be specified. In the following we show that it is also not necessary to fix buffer output sequences. For given sequences $\pi^1, \ldots, \pi^m$, a polynomial procedure is developed, which calculates optimal buffer output sequences and a corresponding schedule at the same time.

The idea of this procedure is to proceed in time and schedule operations as soon as possible. At the earliest time t where at least one operation is finishing the following moves are performed if applicable:

- move a job out of the system if its last operation has finished,
- move a job finishing at time t to the next machine and start to process it there,
- move a job finishing at time t on a machine M_k into buffer B_k,

- move a job from a buffer to the next machine and start to process it there,
- identify a sequence of operations $i_0, \ldots, i_{r-1}$ with $r \geq 2$ such that
  - at time t each operation stays either finished on a machine or in a buffer,
  - job $J(i_\nu)$ can move to the place occupied by job $J(i_{(\nu+1) \bmod r})$ for $\nu = 0, \ldots, r-1$, (which implies that not both jobs $J(i_\nu)$ and $J(i_{(\nu+1) \bmod r})$ stay in a buffer),

  and perform a cyclic move, i.e. replace $J(i_{(\nu+1) \bmod r})$ by $J(i_\nu)$ on its machine or in the corresponding buffer for $\nu = 0, \ldots, r-1$.

```
Algorithm Output Buffers
 1.  t := 0; C := ∅;
 2.  FOR all operations i DO t_i := ∞;
 3.  Mark all machines and buffers as available;
 4.  FOR all first operations i of the jobs which are
         sequenced first on a machine DO
 5.      Schedule i on μ(i) starting at time t = 0;
 6.      t_i := p_i;   C = C ∪ {i};
 7.      Mark μ(i) as nonavailable;
 8.  ENDFOR
 9.  WHILE C ≠ ∅ DO
10.      FOR each machine M_k which is available DO
11.          IF the current first element i of π^k is
                the first operation of job J(i) THEN
12.              Schedule i on M_k starting at time t;
13.              t_i := t + p_i; C = C ∪ {i};
14.              Mark M_k as nonavailable;
15.          ENDIF
16.      IF an operation i ∈ C with t_i ≤ t exists and a move
             of an operation in C is possible at time t THEN
17.          Update(C, t);
18.      ELSE
19.          IF t_i ≤ t for all i ∈ C THEN
20.              STOP; /* solution is infeasible */
21.          t := min {t_i | i ∈ C; t_i > t};
22.      ENDIF
23.  ENDWHILE
24.  IF there is an operation i with t_i = ∞ THEN
25.      STOP; /* solution is infeasible */
```

Figure 4.27: Algorithm Output Buffers

This dynamic process is controlled by the algorithm shown in Figure 4.27. We keep a set C containing all operations which at the current time t are either staying on a machine or are stored in a buffer. Furthermore, the machine and the buffer are marked as available or nonavailable. A machine is nonavailable if it is occupied by an operation. Otherwise, it is available. A buffer is available if and only if it is not fully occupied. For each operation i starting at time t on machine $\mu(i)$ we store the corresponding finishing time $t_i := t + p_i$. At the beginning we set $t_i := \infty$ for all operations i. A job enters the system if its first operation i can be processed on machine $\mu(i) = M_k$, i.e. if the predecessor of i in the machine sequence π^k has left M_k. At the beginning each job whose first operation is the first operation in a corresponding machine sequence enters the system.

If at the current time t the set C is not empty and no move is possible, then we replace t by $\min\{t_i \mid i \in C; t_i > t\}$ if there is an operation $i \in C$ with $t_i > t$. Otherwise, we have a deadlock situation. In this case, the machine sequences are infeasible (cf. Step 20). An infeasible situation may also occur when C becomes empty and there are still unprocessed jobs (cf. Step 25).

```
Procedure Update(C, t)
1.  IF there is an operation i ∈ C with t_i ≤ t where i is the
       last operation of job J(i) THEN
2.        Move_out_of_system(C, t, i);
3.  ELSE IF an operation i ∈ C with t_i ≤ t on machine μ(i) exists
       AND σ(i) is the current first element of π^{μ(σ(i))}
       AND μ(σ(i)) is available THEN
4.        Move_to_machine(C, t, i);
5.  ELSE IF an operation i ∈ C with t_i ≤ t on machine μ(i) exists
       AND buffer β(i) is available THEN
6.        Move_into_buffer(C, t, i);
7.  ELSE IF there is an operation i ∈ C with t_i ≤ t in a buffer
       AND σ(i) is the current first element of π^{μ(σ(i))}
       AND μ(σ(i)) is available THEN
8.        Move_out_of_buffer(C, t, i);
9.  ELSE IF there is a sequence Z = (i_0, ..., i_{r-1}) of operations
       i_ν ∈ C with t_{i_ν} ≤ t such that for ν = 0, ..., r − 1 at time t
       operation i_ν is on machine μ(i_ν) and J(i_{(ν+1) mod r}) is in
       the buffer β(i_ν) OR operation i_{(ν+1) mod r} is on machine
       μ(σ(i_ν)) and σ(i_ν) is at the second position
       in π^{μ(i_{(ν+1) mod r})} THEN
10.       Swap(C, t, Z);
```

Figure 4.28: Procedure Update

```
Procedure Move_out_of_system(C, t, i)
1.  Eliminate i from the machine sequence π^{μ(i)};
2.  Mark machine μ(i) as available;
3.  C := C \ {i};

Procedure Move_to_machine(C, t, i)
1.  Eliminate i from the machine sequence π^{μ(i)};
2.  Mark μ(i) as available;
3.  Schedule σ(i) on μ(σ(i)) starting at time t;
4.  Mark μ(σ(i)) as nonavailable;
5.  t_{σ(i)} := t + p_{σ(i)};
6.  C := (C \ {i}) ∪ {σ(i)};

Procedure Move_into_buffer(C, t, i)
1.  Move J(i) from the machine μ(i) into the buffer β(i);
2.  Eliminate i from the machine sequence π^{μ(i)};
3.  Mark μ(i) as available;
4.  IF buffer β(i) is now fully occupied THEN
5.     Mark β(i) as nonavailable;

Procedure Move_out_of_buffer(C, t, i)
1.  Eliminate J(i) from the buffer β(i);
2.  Mark buffer β(i) as available;
3.  Schedule σ(i) on μ(σ(i)) starting at time t;
4.  Mark μ(σ(i)) as nonavailable;
5.  t_{σ(i)} := t + p_{σ(i)};
6.  C := (C \ {i}) ∪ {σ(i)};

Procedure Swap(C, t, Z)
 1.  FOR ν := 0 TO r − 1 DO
 2.      IF J(i_{(ν+1) mod r}) is in the buffer β(i_ν) THEN
 3.         Eliminate i_ν from the machine sequence π^{μ(i_ν)};
 4.         Move J(i_ν) into the buffer β(i_ν);
 5.      ELSE
 6.         Eliminate i_ν from the machine sequence π^{μ(i_ν)} or
            from its buffer;
 7.         Schedule σ(i_ν) on μ(σ(i_ν)) starting at time t;
 8.         t_{σ(i_ν)} := t + p_{σ(i_ν)};
 9.         C := (C \ {i_ν}) ∪ {σ(i_ν)}
10.      ENDIF
```

Figure 4.29: Five different moves in procedure `Update`

The algorithm calls the procedure `Update`(C, t) shown in Figure 4.28 which performs one possible move of the five different types of moves shown in Figure 4.29:

- in `Move_out_of_system`(C, t, i) job $J(i)$ leaves the last machine (i.e. the complete system),
- in `Move_to_machine`(C, t, i) job $J(i)$ goes directly from its current machine to the next,
- in `Move_into_buffer`(C, t, i) job $J(i)$ goes from its current machine into a buffer,
- in `Move_out_of_buffer`(C, t, i) job $J(i)$ goes from a buffer to the next machine, and
- in `Swap`(C, t, Z) a cyclic move of jobs in $Z = (i_\nu)_{\nu=0}^{r-1}$ is performed:
  - if job $J(i_{(\nu+1) \bmod r}$ is in the buffer $\beta(i_\nu)$, then job $J(i_\nu)$ goes from its current machine into this buffer and job $J(i_{(\nu+1) \bmod r}$ leaves the buffer,
  - if job $J(i_{(\nu+1) \bmod r}$ is on a machine, then job $J(i_\nu)$ goes from its current machine (or a buffer) to this machine and job $J(i_{(\nu+1) \bmod r}$ leaves the machine.

Example 4.9: Consider an instance with $m = 3$ machines, $N = 5$ jobs and output buffers B_1, B_2 and B_3 of capacities $b_1 = 0$, $b_2 = 1$ and $b_3 = 0$. Jobs $j = 1, 2$ consist of $n_j = 3$ operations and jobs $j = 3, 4, 5$ consist of $n_j = 2$ operations. Furthermore, the operations O_{ij} have the following processing times p_{ij} and have to be processed on the following machines μ_{ij}:

p_{ij}	j				
i	1	2	3	4	5
1	3	1	1	5	2
2	2	4	3	1	2
3	1	2	–	–	–

μ_{ij}	j				
i	1	2	3	4	5
1	M_1	M_2	M_2	M_3	M_1
2	M_2	M_1	M_3	M_1	M_2
3	M_3	M_2	–	–	–

Figure 4.30 shows a schedule for the given instance where the operations on the machines M_1, M_2 and M_3 are sequenced in the orders $\pi^1 = (O_{11}, O_{22}, O_{24}, O_{15})$, $\pi^2 = (O_{12}, O_{13}, O_{21}, O_{32}, O_{25})$ and $\pi^3 = (O_{14}, O_{31}, O_{23})$, respectively.

The Algorithm Output Buffers constructs this schedule as follows: We initialize at $t = 0$ by adding the first operations of jobs 1, 2 and 4 to C and set $t_{11} = 3$, $t_{12} = 1$ and $t_{14} = 5$. Since no move is possible at $t = 0$, we increase t to $t_{12} = 1$. At this time, a move of job 2 into the buffer B_2 is performed. Operation O_{12} of job 2 is eliminated from the first position of π^2, machine M_2 is marked as available and B_2 is marked as nonavailable. Next, the first operation of job 3 is scheduled on M_2. We add O_{13} to C and set $t_{13} = 2$.

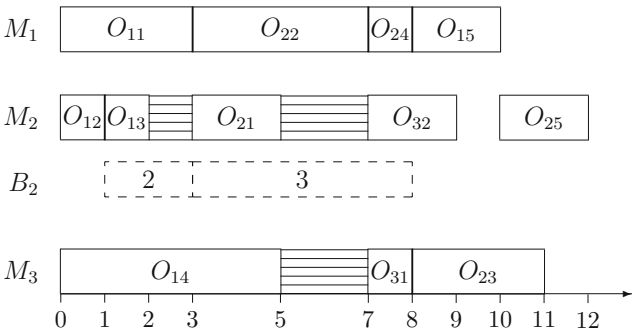


Figure 4.30: Schedule for a job-shop problem with output buffers

t	action	C	t_i	M_1	M_2	B_2	M_3
0		$\emptyset$		a	a	a	a
	schedule first operations of jobs 1, 2 and 4	O_{11}, O_{12}, O_{14}	$t_{11} = 3$, $t_{12} = 1$, $t_{14} = 5$	–	–	a	–
1	move job 2 in B_2;			–	a	–	–
	schedule first operation of job 3	$O_{11}, O_{12}, O_{13}, O_{14}$	$t_{13} = 2$	–	–	–	–
2	no move is possible			–	–	–	–
3	swap of jobs 1, 3 and 2	$O_{21}, O_{22}, O_{13}, O_{14}$	$t_{21} = 5$, $t_{22} = 7$	–	–	–	–
5	no move is possible			–	–	–	–
7	swap of jobs 1, 4 and 2	$O_{31}, O_{32}, O_{13}, O_{24}$	$t_{31} = 8$, $t_{32} = 9$, $t_{24} = 8$	–	–	–	–
8	eliminate last operations of jobs 1 and 4;	O_{32}, O_{13}		a	–	–	a
	schedule first operation of job 5;	O_{32}, O_{13}, O_{15}	$t_{15} = 10$	–	–	–	a
	move job 3 out of B_2 on M_3	O_{32}, O_{23}, O_{15}	$t_{23} = 11$	–	–	a	–
9	eliminate last operation of job 2	O_{23}, O_{15}		–	a	a	–
10	move job 5 from M_1 to M_2	O_{23}, O_{25}	$t_{25} = 12$	a	–	a	–
11	eliminate last operation of job 3	O_{25}		a	–	a	a
12	eliminate last operation of job 5	$\emptyset$		a	a	a	a

Table 4.1: Algorithm Output Buffers applied to Example 4.9

Since no further move is possible at time $t = 1$ and no move is possible at $t = t_{13} = 2$, the next relevant time is $t = t_{11} = 3$. At this time, the sequence

$Z = (O_{11}, O_{13}, O_{12})$ satisfying the conditions in Step 9 of procedure `Update` exists, where operation O_{11} is on machine M_1, operation O_{13} is on machine M_2 and the job $J(O_{12}) = 2$ is in the buffer B_2. Therefore, a simultaneous swap of the jobs 1, 3 and 2 can be performed: Job 1 is moved from M_1 to M_2, job 3 is moved from M_2 into the buffer B_2 and job 2 is moved from B_2 to M_1. Then, we eliminate the first operations O_{11}, O_{12} of the jobs 1 and 2 from C and add their second operations O_{21}, O_{22} to C. The first operation O_{13} of job 3 is still contained in C since job 3 only changes from machine M_2 into the buffer B_2.

We set $t_{22} = 7$ and $t_{21} = 5$ and eliminate operation O_{22} from the currently first position in π^1 and operation O_{21} from the currently first position in π^2. Note, that still M_1, M_2 and B_2 are marked as nonavailable. The further steps of Algorithm Output Buffers are shown in Table 4.1. In the columns M_1, M_2, B_2 and M_3, we set the mark "a" if the corresponding machine or buffer is available. □

For given sequences $\pi^1, \ldots, \pi^m$ of the operations on the machines Algorithm Output Buffers provides an optimal solution since each operation is scheduled as early as possible. Postponing the start of an operation i on machine M_k is not advantageous when the sequence π^k of machine M_k is fixed and i is the operation to be processed next on M_k. Note, that the machine sequences are compatible if and only if Algorithm Output Buffers schedules all operations.

Furthermore, the schedule constructed by the algorithm also induces buffer output sequences. In contrast to the previous buffer cases, these buffer output sequences are dependent on the processing times of the given instance. This means, for two instances of a job-shop problem with output buffers which only differentiate in the processing times of the jobs, the optimal assignment of operations to buffer slots may be different though the given machine sequences are equal. Thus, also the corresponding solution graphs of such instances for given machine sequences may be different. Consequently, in the output buffer case, the constructed solution graph is not only dependent on the sequences of the jobs on the machines as in the preceding types of buffers but it is also based on the processing times of the given instance.

The time complexity of the Algorithm Output Buffers is $O(n \min\{N, m\})$. Since only completion times of operations are considered, we have at most $O(n)$ different time values t. For a fixed value t all possible moves can be performed in $O(\min\{N, m\})$ time because for fixed t at most one operation of each job is involved in a move and all these operations have to be processed on different machines.

4.7.6 Job-shop problems with input buffers

Similarly to an output buffer, an **input buffer** B_k is a buffer which is directly related to machine M_k ($k = 1, \ldots, m$). An input buffer stores all jobs that have already finished processing on the previous machines but cannot be loaded

directly on machine M_k. In the case of a job-shop problem with input buffers the output sequence π^k_{out} of buffer B_k is equal to the sequence π^k.

The job-shop problem with input buffers can be seen as a symmetric counterpart to the problem with output buffers in the following sense: A given instance of a job-shop problem with input buffers can be reversed to an instance of a job-shop problem with output buffers by inverting any precedence $i \to \sigma(i)$ into $\sigma(i) \to i$ and by changing the input buffer B_k related to M_k into an output buffer $(k = 1, \ldots, m)$. Both problems have the same optimal makespan $C_{\max}$, i.e. it is sufficient to solve the output buffer problem. The earliest starting times S_i of operations i in an optimal solution of the output buffer problem provide latest finishing times $C_{\max} - S_i$ of operations i in a solution of the input buffer problem minimizing the makespan.

4.7.7 Job-shop problems with job-dependent buffers

Another buffer model is that of job-dependent buffers. It is easy to handle because it leads directly to a job-shop problem with blocking operations. In this case N buffers B_j $(j = 1, \ldots, N)$ are given where B_j may store only operations belonging to job j. Since operations of the same job never require the buffer at the same time, we may restrict the buffer capacity b_j to the values 0 and 1. If an operation belongs to a job with buffer capacity 1, it is never blocking because it always can go into the buffer when finishing. However, a machine successor k of an operation i can be blocked by i if $J(i)$ has a buffer with capacity 0 and $\sigma(i)$ is not started. Therefore, we have to introduce the blocking arc $(\sigma(i), k)$ with weight 0.

Example 4.10: Consider the example with 3 jobs and 3 machines shown in

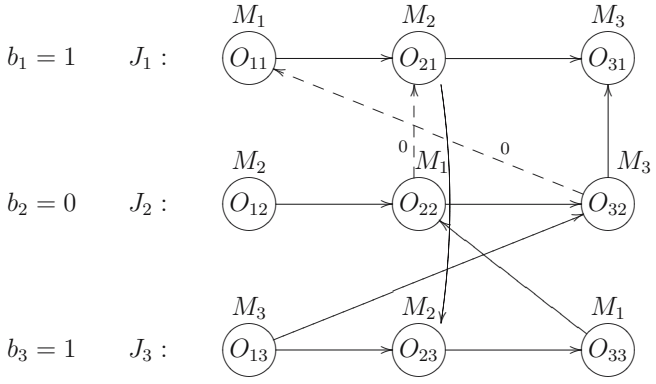


Figure 4.31: Blocking arcs for job-dependent buffers

Figure 4.31. For the machine sequences $\pi^1 = (O_{33}, O_{22}, O_{11})$ on M_1, $\pi^2 = (O_{12}, O_{21}, O_{23})$ on M_2 and $\pi^3 = (O_{13}, O_{32}, O_{31})$ on M_3 we get the blocking arcs (O_{22}, O_{21}) and (O_{32}, O_{11}). □

4.8 Reference Notes

The job-shop problem has been studied since 1960. In the book of Muth and Thompson [109] a benchmark instance with 10 jobs and 10 machines was introduced which could be solved only 25 years later. Since then a lot of work has been done, cf. the surveys of Pinson [124], Błażewicz et al. [12], Jain and Meeran [77]. The disjunctive graph model was proposed by Roy and Sussmann [127] in 1964, the alternative graph model by Mascis and Pacciarelli [102] in 2000. Job-shop problems with blocking and no-wait constraints are discussed in Mascis and Pacciarelli [103].

A computational study of different local search algorithms for job-shop problems can be found in Aarts et al. [1]. The neighborhood $\mathcal{N}_{ca}$ based on critical arcs was first used in a simulated annealing algorithm by van Laarhoven et al. [139]. A block approach was first proposed for a single-machine problem with release dates and the maximum lateness objective function by Grabowski et al. [63]. Later it was successfully adapted to some other scheduling problems (like the job-shop and flow-shop problem, cf. Nowicki and Smutnicki [112], [113], Dell' Amico and Trubian [38]). In Nowicki and Smutnicki [114] an advanced tabu search algorithm is presented which evaluatates neighbor solutions in an efficient way (cf. the descriptions at the end of Section 4.2).

Branch-and-bound algorithms for the job-shop problem were proposed by several authors. Carlier and Pinson [30], [31], [32] developed algorithms which are based on branching on disjunctions and efficient constraint propagation techniques. Brucker et al. [21] used a block approach and Martin and Shmoys [101] described different time-oriented branching schemes (cf. also Martin [100]).

Tabu search algorithms for the job-shop problem with flexible machines have been proposed by Hurink et al. [74] and Mastrolilli and Gambardella [104]. While the first paper uses a block approach, the second introduces the reduced neighborhood $\mathcal{N}^{red}$.

Scheduling problems with transportation aspects have received much attention in recent years (for a survey cf. Crama et al. [36]). Different aspects of job-shop problems with an unlimited or a limited number of transport robots were studied in Knust [84]. Constraint propagation techniques and lower bounds for job-shop problems with transport robots can be found in Brucker and Knust [25]. The dynamic programming algorithm for the robot problem taking into account time-lags and setup times is based on an algorithm for the asymmetric traveling salesman problem with time windows by Dumas et al. [52]. The chain-packing lower bound was developed by Knust [84] extending the job-packing bound used by Martin [100] in connection with the classical job-shop problem. Details for calculating the q-chain-packing bound can be found in [84] and [25]. One-stage and two-stage tabu search algorithms for the job-shop problem with a single transport robot were developed by Hurink and Knust [76]. An efficient tabu search algorithm for the subproblem of scheduling a robot for a fixed machine selection in the second stage can be found in Hurink and Knust [75].

Several authors investigated flow-shop problems with buffers which have been shown to be NP-hard even for two machines by Papadimitriou and Kanellakis [117]. Leisten [97] presented some priority-based heuristics for the permutation flow-shop problem as well as for the general flow-shop problem with buffers. Smutnicki [131] and Nowicki [111] developed tabu search approaches for the permutation flow-shop problem with two and arbitrary many machines, respectively. Brucker et al. [18] generalized the approach of Nowicki [111] to the case where different job-sequences on the machines are allowed.

The different buffer-models for job-shop problems are considered in Brucker et al. [19]. They proved that pairwise buffers, job-dependent buffers and input- or output buffers have the property that if schedules are represented by machine sequences then corresponding optimal schedules can be calculated in polynomial time. It is unlikely that this property also holds for the general buffer model because the problem of finding an optimal schedule for given machine sequences is strongly NP-hard in that case (cf. [19]).

Bibliography

[1] **Aarts, E.H.L., van Laarhoven, P.J.M., Lenstra, J.K., Ulder, N.L.J. (1994)**: A computational study of local search algorithms for job shop scheduling, ORSA Journal on Computing 6, 118–125.

[2] **Aarts, E.H.L., Lenstra, J.K. (1997)**: Local Search in Combinatorial Optimization, John Wiley.

[3] **Ahuja, R.K., Magnanti, T.L., Orlin, J.B. (1993)**: Network Flows, Prentice Hall, Englewood Cliffs.

[4] **Alvarez-Valdes, R., Tamarit, J.M. (1989)**: Heuristic algorithms for resource-constrained project scheduling: a review and an empirical analysis, in: R. Słowiński, J. Węglarz, (eds.): Advances in Project Scheduling, Elsevier Science, Amsterdam, 114–134.

[5] **Baar, T., Brucker, P., Knust, S. (1998)**: Tabu-search algorithms and lower bounds for the resource-constrained project scheduling problem, in: S. Voss, S. Martello, I. Osman, C. Roucairol (eds.): Meta-heuristics: Advances and Trends in Local Search Paradigms for Optimization, Kluwer, 1–18.

[6] **Baptiste, P., Le Pape, C. (1996)**: Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling, in: Proceedings of the 15th Workshop of the UK Planning Special Interest Group, Liverpool.

[7] **Baptiste, P., Le Pape, C., Nuijten, W. (2001)**: Constraint-Based Scheduling – Applying Constraint Programming to Scheduling Problems, International Series in Operations Research and Management Science 39, Kluwer, Dordrecht.

[8] **Baptiste, P., Demassey, S. (2004)**: Tight LP bounds for resource constrained project scheduling, OR Spectrum 26, 251–262.

[9] **Bartusch, M., Möhring, R.H., Radermacher, F.J. (1988)**: Scheduling project networks with resource constraints and time windows, Annals of Operations Research 16, 201–240.

[10] **Bellman, R.E. (1958)**: Dynamic Programming, Princeton University Press.

[11] **Bland, R.G. (1977)**: New finite pivoting rules for the simplex method, Mathematics of Operations Research 2, 103–107.

[12] **Błażewicz, J., Domschke, W., Pesch, E. (1996)**: The job shop scheduling problem: conventional and new solution techniques, European Journal of Operational Research 93, 1–33.

[13] **Błażewicz, J., Ecker, K., Pesch, E., Schmidt, G., Węglarz, J. (2001)**: Scheduling Computer and Manufacturing Processes, 2nd ed., Springer, Berlin.

[14] **Błażewicz, J., Lenstra, J.K., Rinnooy Kan, A.H.G. (1983)**: Scheduling subject to resource constraints: Classification and complexity, Discrete Applied Mathematics 5, 11–24.

[15] **Böttcher, J., Drexl, A., Kolisch, R., Salewski, F. (1999)**: Project scheduling under partially renewable resource constraints, Management Science 45, 543–559.

[16] **Brucker, P. (2004)**: Scheduling Algorithms, 4th ed., Springer, Berlin.

[17] **Brucker, P., Drexl, A., Möhring, R.H., Neumann, K., Pesch, E. (1999)**: Resource-constrained project scheduling: notation, classification, models and methods, European Journal of Operational Research 112, 3–14.

[18] **Brucker, P., Heitmann, S., Hurink, J. (2003)**: Flow-shop problems with intermediate buffers, OR Spectrum 25, 549–574.

[19] **Brucker, P., Heitmann, S., Hurink, J., Nieberg, T. (2004)**: Job-Shop scheduling with limited capacity buffers, Osnabrücker Schriften zur Mathematik, Reihe P, No. 253, to appear in OR Spectrum.

[20] **Brucker, P., Jurisch, B., Krämer, A. (1996)**: The job-shop problem and immediate selection, Annals of Operations Research 50, 73–114.

[21] **Brucker, P., Jurisch, B., Sievers, B. (1994)**: A fast branch and bound algorithm for the job shop scheduling problem, Discrete Applied Mathematics 49, 107–127.

[22] **Brucker, P., Knust, S., Schoo, A., Thiele, O. (1998)**: A branch & bound algorithm for the resource-constrained project scheduling problem, European Journal of Operational Research 107, 272–288.

[23] **Brucker, P., Knust, S. (2000)**: A linear programming and constraint propagation-based lower bound for the RCPSP, European Journal of Operational Research 127, 355–362.

[24] **Brucker, P., Knust, S. (2001)**: Resource-constrained project scheduling and timetabling, in: E. Burke, W. Erben (ed.): The Practice and Theory of Automated Timetabling III, Springer Lecture Notes in Computer Science 2079, 277–293.

[25] **Brucker, P., Knust, S. (2002)**: Lower bounds for scheduling a single robot in a job-shop environment, Annals of Operations Research 115, 147-172.

[26] **Brucker, P., Knust, S. (2003)**: Lower bounds for resource-constrained project scheduling problems, European Journal of Operational Research 149, 302–313.

[27] **Brucker, P., Knust, S. (2005)**: An $O(n^2)$-algorithm for the input-or-output test, Osnabrücker Schriften zur Mathematik, No. 259.

[28] **Brucker, P., Knust, S.**: Complexity classification of scheduling problems,
`http://www.mathematik.uni-osnabrueck.de/research/OR/class/`.

[29] **Brucker, P., Schumacher, D. (1999)**: A new tabu search procedure for an audit-scheduling problem, Journal of Scheduling 2, 157–173.

[30] **Carlier, J., Pinson, E. (1989)**: An algorithm for solving the job-shop problem, Management Science 35, 164–176.

[31] **Carlier, J., Pinson, E. (1990)**: A practical use of Jackson's preemptive schedule for solving the job-shop problem, Annals of Operations Research 26, 269–287.

[32] **Carlier, J., Pinson, E. (1994)**: Adjustments of heads and tails for the job-shop problem, European Journal of Operational Research 78, 146–161.

[33] **Christofides, N., Alvarez-Valdés, R., Tamarit, J.M. (1987)**: Project scheduling with resource constraints: a branch and bound approach, European Journal of Operational Research 29, 262–273.

[34] **Chvátal, V. (1983)**: Linear Programming, W.H. Freeman and Company, New York – San Francisco.

[35] **Cook, S.A. (1971)**: The complexity of theorem-proving procedures, Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, 151–158.

[36] **Crama, Y., Kats, V., Klundert, J. van de, Levner, E. (2000)**: Cyclic scheduling in robotic flowshops, Annals of Operations Research 96, 97–124.

[37] **Davies, E.M. (1973)**: An experimental investigation of resource allocation in multiactivity projects, Operational Research Quarterly 24, 587–591.

[38] **Dell' Amico, M., Trubian, M. (1993)**: Applying tabu search to the job-shop scheduling problem, Annals of Operations Research 41, 231–252.

[39] **Demeulemeester, E. (1992)**: Optimal algorithms for various classes of multiple resource-constrained project scheduling problem, Ph.D. thesis, Katholieke Universiteit Leuven, Belgium.

[40] **Demeulemeester, E., Herroelen, W. (1992)**: A branch-and-bound procedure for the multiple resource-constrained project scheduling problem, Management Science 38, 1803–1818.

[41] **Demeulemeester, E., Herroelen, W. (1997)**: New benchmark results for the resource-constrained project scheduling problem, Management Science 43, 1485–1492.

[42] **Demeulemeester, E., Herroelen, W. (2002)**: Project Scheduling, A Research Handbook, Kluwer, Dordrecht.

[43] **De Reyck, B., Herroelen, W. (1998)**: A branch-and-bound procedure for the resource-constrained project scheduling problem with generalized precedence relations, European Journal of Operational Research 111, 152–174.

[44] **De Reyck, B., Herroelen, W. (1999)**: The multi-mode resource-constrained project scheduling problem with generalized precedence relations, European Journal of Operational Research 119, 538–556.

[45] **Dijkstra, E. (1959)**: A note on two problems in connexion with graphs, Numerische Mathematik 1, 269–271.

[46] **Dodin, B., Elimam, A.A., Rolland, E. (1998)**: Tabu search in audit scheduling, European Journal of Operational Research 106, 373–392.

[47] **Dorndorf, U. (2002)**: Project Scheduling with Time Windows – From Theory to Applications, Contributions to Management Science, Physica-Verlag, Heidelberg.

[48] **Dorndorf, U., Pesch, E., Phan-Huy, T. (1998)**: A survey of interval consistency tests for time- and resource-constrained scheduling, in: [141], 213–238.

[49] **Dorndorf, U., Pesch, E., Phan-Huy, T. (2000)**: Constraint propagation techniques for the disjunctive scheduling problem, Artificial Intelligence 122, 189–240.

[50] **Dorndorf, U., Pesch, E., Phan-Huy, T. (2000)**: A time-oriented branch-and-bound algorithm for project scheduling with generalised precedence constraints, Management Science 46, 1365–1384.

[51] **Drexl, A., Knust, S. (2004)**: Sports league scheduling: graph- and resource-based models, Osnabrücker Schriften zur Mathematik, No. 255, to appear in OMEGA – The International Journal of Management Science.

[52] **Dumas, Y., Desrosiers, J., Gelinas, E. (1995)**: An optimal algorithm for the traveling salesman problem with time windows, Operations Research 43, 367–371.

[53] **Fest, A., Möhring, R.H., Stork, F., Uetz, M. (1998)**: Resource-constrained project scheduling with time windows: A branching scheme based on dynamic release dates, Technical Report 596, Technische Universität Berlin, 1998.

[54] **Fisher, M.L. (1973)**: Optimal solution of scheduling problems using Lagrange multipliers: Part I, Operations Research 21, 1114–1127.

[55] **Floyd, R.W. (1962)**: Algorithm 97: Shortest path, Communications of the Association for Computing Machinery 5, 345.

[56] **Ford, L.R., Fulkerson, D.R. (1956)**: Maximal flow through a network, Canadian Journal of Mathematics 8, 399–404.

[57] **Garey, M., Johnson, D.S. (1979)**: Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, New York.

[58] **Gilmore, P.C., Gomory, R.E. (1961)**: A linear programming approach to the cutting-stock problem, Operations Research 9, 849–859.

[59] **Glover, F. (1989)**: Tabu Search, Part I, ORSA Journal of Computing 1, 190-206.

[60] **Glover, F. (1990)**: Tabu Search, Part II, ORSA Journal of Computing 2, 4-32.

[61] **Glover, F., Laguna, M. (1997)**: Tabu Search, Kluwer, Dordrecht.

[62] **Golumbic, M.C. (1980)**: Algorithmic graph theory and perfect graphs, Academic Press, New York.

[63] **Grabowski, J., Nowicki, E. Zdrzalka, S. (1986)**: A block approach for single machine scheduling with release dates and due dates, European Journal of Operational Research 26, 278–285.

[64] **Hartmann, S. (1998)**: A competitive genetic algorithm for resource-constrained project scheduling, Naval Research Logistics 45, 733–750.

[65] **Hartmann, S. (2000)**: Packing problems and project scheduling models: an integrating perspective, Journal of the Operational Research Society 51, 1083–1092.

[66] **Hartmann, S. (2001)**: Project scheduling with multiple modes: a genetic algorithm, Annals of Operations Research 102, 111–135.

[67] **Hartmann, S. (2002)**: A self-adapting genetic algorithm for project scheduling under resource constraints, Naval Research Logistics 49, 433–488.

[68] **Hartmann, S., Drexl, A. (1998)**: Project scheduling with multiple modes: a comparison of exact algorithms, Networks 32, 283–297.

[69] **Hartmann, S., Kolisch, R. (2000)**: Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem, European Journal of Operational Research 127, 394–407.

[70] **Heilmann, R. (1999)**: Das ressourcenbeschränkte Projektdauerminimierungsproblem im Mehr-Modus-Fall, Ph.D. thesis, University of Karlsruhe, Germany.

[71] **Hentenryck, P. van (1992)**: Constraint Satisfaction in Logic Programming, MIT Press, Cambridge.

[72] **Herroelen, W., Demeulemeester, E., De Reyck, B. (1998)**: Resource-constrained project scheduling – A survey of recent developments, Computers & Operations Research 25, 279–302.

[73] **Holland, S. (1975)**: Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, MI.

[74] **Hurink, J., Jurisch, B., Thole, M. (1994)**: Tabu search for the job shop scheduling problem with multi-purpose machines, OR Spektrum 15, 205–215.

[75] **Hurink, J., Knust, S. (2002)**: A tabu search algorithm for scheduling a single robot in a job-shop environment, Discrete Applied Mathematics 112, 181–203.

[76] **Hurink, J., Knust, S. (2005)**: Tabu search algorithms for job-shop problems with a single transport robot, European Journal of Operational Research 162, 99–111.

[77] **Jain, A.S., Meeran, S. (1999)**: Deterministic job-shop scheduling: past, present and future, European Journal of Operational Research 113, 390–434.

[78] **Kelley, J.E. (1963)**: The critical path method: Resource planning and scheduling, in: J.F. Muth, G.L. Thompson (eds.): Industrial Scheduling, Prentice Hall, New Jersey, 347–365.

[79] **Khachian, L.G. (1979)**: A polynomial algorithm in linear programming, Soviet Math. Dokl. 20, 191–194.

[80] **Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P. (1983)**: Optimization by simulated annealing, Science 220, 671–680.

[81] **Klee, V., Minty, G.J. (1972)**: How good is the simplex algorithm? Inequalities III, Proc. 3rd Symp., Los Angeles 1969, 159–175.

[82] **Klein, M. (1967)**: A primal method for minimal cost flows with application to the assignment and transportation problems, Management Science 14, 205–220.

[83] **Klein, R., Scholl, A. (1999)**: Computing lower bounds by destructive improvement: an application to resource-constrained project scheduling, European Journal of Operational Research 112, 322–346.

[84] **Knust, S. (1999)**: Shop-scheduling problems with transportation, Ph.D. thesis, University of Osnabrück, Germany.

[85] **Kolisch, R. (1996)**: Serial and parallel resource-constrained project scheduling methods revisited: theory and computation, European Journal of Operational Research 90, 320–333.

[86] **Kolisch, R., Hartmann, S. (1998)**: Heuristic algorithms for solving the resource-constrained project scheduling problem: classification and computational analysis, in: [141], 147–178.

[87] **Kolisch, R., Hartmann, S. (2004)**: Experimental investigation of heuristics for resource-constrained project scheduling: an update, submitted to European Journal of Operational Research.

[88] **Kolisch, R., Padman, R. (2001)**: An integrated survey of deterministic project scheduling, OMEGA – International Journal of Management Science 29, 249–272.

[89] **Kolisch, R., Schwindt, C., Sprecher, A. (1998)**: Benchmark instances for project scheduling problems, in: [141], 197–212.

[90] **Kolisch, R., Sprecher, A., Drexl, A. (1995)**: Characterization and generation of a general class of resource-constrained project scheduling problems, Management Science 41, 1693–1703.

[91] **Kolisch, R., Sprecher, A. (1997)**: PSPLIB – A project scheduling library, European Journal of Operational Research 96, 205–216.

[92] **Korte, B., Möhring, R.H. (1985)**: Transitive orientation of graphs with side constraints, in: H. Noltemeier (ed.): Proceedings 11th International Workshop on Graph Theoretic Concepts in Computer Science, Trauner Verlag, Linz, 143–160.

[93] **Korte, B., Vygen, J. (2002)**: Combinatorial optimization – Theory and algorithms, 2nd ed., Springer, Berlin.

[94] **Krämer, B. (1997)**: Branch-and-bound methods for scheduling problems with multiprocessor tasks on dedicated processors, OR Spektrum 19, 219–227.

[95] **Kumar, V. (1992)**: Algorithms for constraint satisfaction problems, Artificial Intelligence Magazine 13, 32–44.

[96] **Lawler, E.L. (1976)**: Combinatorial Optimization: Networks and Matroids, Holt, Rinehart and Winston, New York.

[97] **Leisten, R. (1990)**: Flowshop sequencing problems with limited buffer storage, International Journal of Production Research 28, 2085–2100.

[98] **Leung, J.T. (2004)**: Handbook of Scheduling, CRC Press.

[99] **Malcolm, D.G., Roseboom, J.H., Clark, C.E., Fazar, W. (1959)**: Application of a technique for research and development program evaluation, Operations Research 7, 646–669.

[100] **Martin, P.B. (1996)**: A time-oriented approach to computing optimal schedules for the job shop scheduling problem, Ph.D. thesis, Graduate School of Cornell University, USA.

[101] **Martin, P., Shmoys, D. (1996)**: A new approach to computing optimal schedules for the job-shop scheduling problem, in: Proceedings of the 5th International IPCO Conference.

[102] **Mascis, A., Pacciarelli, D. (2000)**: Machine scheduling via alternative graphs, Report DIA-46-2000, University of Rome, Italy.

[103] **Mascis, A., Pacciarelli, D. (2002)**: Job-shop scheduling with blocking and no-wait constraints, European Journal of Operational Research 143, 498–517.

[104] **Mastrolilli, M., Gambardella, L.M. (2000)**: Effective neighbourhood functions for the flexible job shop problem, Journal of Scheduling 3, 3–20.

[105] **Mingozzi, A., Maniezzo, V., Ricciardelli, S., Bianco, L. (1998)**: An exact algorithm for project scheduling with resource constraints based on a new mathematical formulation, Management Science 44, 714–729.

[106] **Möhring, R.H. (1985)**: Algorithmic aspects of comparability graphs and interval graphs, in: I. Rival (ed.): Graphs and order: the role of graphs in the theory of ordered sets and its applications, NATO Advanced Science Institute Series C, Mathematical and Physical Sciences, 41–101.

[107] **Möhring, R.H., Schulz, A.S., Stork, F., Uetz, M. (2001)**: On project scheduling with irregular starting time costs, Operations Research Letters 28, 149–154.

[108] **Möhring, R.H., Schulz, A.S., Stork, F., Uetz, M. (2003)**: Solving project scheduling problems by minimum cut computations, Management Science 49, 330–350.

[109] **Muth, J.F., Thompson, G.L. (1963)**: Industrial Scheduling, Englewood Cliffs, N.J., Prentice Hall.

[110] **Neumann, K., Schwindt, C., Zimmermann, J. (2003)**: Project Scheduling with Time Windows and Scarce Resources – Temporal and Resource-Constrained Project Scheduling with Regular and Nonregular Objective Functions, Springer, Berlin.

[111] **Nowicki, E. (1999)**: The permutation flow shop with buffers: A tabu search approach, European Journal of Operational Research 116, 205–219.

[112] **Nowicki, E., Smutnicki, C. (1996)**: A fast taboo search algorithm for the job shop problem, Management Science 42, 797–813.

[113] **Nowicki, E., Smutnicki, C. (1996)**: A fast tabu search algorithm for the permutation flow-shop problem, European Journal of Operational Research 91, 160–175.

[114] **Nowicki, E., Smutnicki, C. (2005)**: An advanced tabu search algorithm for the job-shop problem, Journal of Scheduling 8, 145–159.

[115] **Nuijten, W. (1994)**: Time and resource constrained scheduling: a constraint satisfaction approach, Ph.D. thesis, Technical University Eindhoven, The Netherlands.

[116] **Özdamar, L., Ulusoy, G. (1995)**: A survey on the resource-constrained project scheduling problem, IIE Transactions 27, 574–586.

[117] **Papadimitriou, C.H., Kanellakis, P.C. (1980)**: Flow-shop scheduling with limited temporary storage, Journal of the Association for Computing Machinery 27, 533–549.

[118] **Papadimitriou, C.H., Steiglitz, K. (1982)**: Combinatorial Optimization, Prentice Hall, Englewood Cliffs, N.J.

[119] **Patterson, J.H., Slowinski, R., Talbot, F.B., Węglarz, J. (1989)**: An algorithm for a general class of precedence and resource constrained scheduling problems, in: R. Słowiński, J. Węglarz, (eds.): Advances in Project Scheduling, Elsevier Science, Amsterdam, 3–28.

[120] **Patterson, J.H., Slowinski, R., Talbot, F.B., Węglarz, J. (1990)**: Computational experience with a backtracking algorithm for solving a general class of resource constrained scheduling problems, European Journal of Operational Research 90, 68–79.

[121] **Phan Huy, T. (1999)**: Constraint propagation in flexible manufacturing, Ph.D. thesis, University of Bonn, Germany.

[122] **Pinedo, M. (2001)**: Scheduling: Theory, Algorithms, and Systems, 2nd ed., Prentice Hall.

[123] **Pinedo, M. (2005)**: Planning and Scheduling in Manufacturing and Services, Springer, Berlin.

[124] **Pinson, E. (1995)**: The job-shop scheduling problem: a concise survey and some recent developments, in: P. Chretienne, E.G. Coffman, J.K. Lenstra, Z. Liu (eds.): Scheduling Theory and its Applications, John Wiley, 277–293.

[125] **Pritsker, A., Watters, L., Wolfe, P. (1969)**: Multiproject scheduling with limited resources: A zero-one programming approach, Management Science 16, 93–107.

[126] PSPLIB – Project Scheduling Problem Library Kiel
`http://www.bwl.uni-kiel.de/Prod/psplib/`

[127] **Roy, B., Sussmann, B. (1964)**: Les problemes d'ordonnancement avec constraintes disjonctives, Note DS no. 9 bis, SEMA, Paris.

[128] **Schaerf, A. (1995)**: A survey of automated timetabling, CWI-Report CS-R 9567, Amsterdam.

[129] **Schirmer, A., Drexl, A. (2001)**: Allocation of partially renewable resources: Concept, capabilities, and applications, Networks 37, 21–34.

[130] **Schrijver, A. (2003)**: Combinatorial Optimization: Polyhedra and Efficiency, Springer, Berlin.

[131] **Smutnicki, C. (1998)**: A two-machine permutation flow-shop problem with buffers, OR Spektrum 20, 229–235.

[132] **Sprecher, A. (1994)**: Resource-constrained project scheduling – exact methods for the multi-mode case, Lecture Notes in Economics and Mathematical Systems 409, Springer, Berlin.

[133] **Sprecher, A., Drexl, A. (1998)**: Multi-mode resource-constrained project scheduling by a simple, general and powerful sequencing algorithm, European Journal of Operational Research 107, 431–450.

[134] **Sprecher, A., Drexl, A. (1999)**: Note: On semi-active timetabling in resource-constrained project scheduling, Management Science 45, 452–454.

[135] **Sprecher, A., Hartmann, S., Drexl, A. (1997)**: An exact algorithm for project scheduling with multiple modes, OR Spektrum 19, 195–203.

[136] **Sprecher, A., Kolisch, R., Drexl, A. (1995)**: Semi-active, active and non-delay schedules for the resource-constrained project scheduling problem, European Journal of Operational Research 80, 94–102.

[137] **Stinson, J.P., Davis, E.W., Khumawala, B.M. (1978)**: Multiple resource-constrained scheduling using branch and bound, AIIE Transactions 10, 252–259.

[138] **Tsang, E. (1993)**: Foundations of Constraint Satisfaction, Academic Press, Essex.

[139] **van Laarhoven, P.J.M., Aarts, E.H.L., Lenstra, J.K. (1992)**: Job shop scheduling by simulated annealing, Operations Research 40, 113–125.

[140] **Vilim, P. (2004)**: $O(n \log n)$ filtering algorithms for unary resource constraint in: Proceedings of CPAIOR 2004, Nice, France.

[141] **Węglarz, J. (1998)**: Project Scheduling: Recent Models, Algorithms and Applications, Kluwer, Dordrecht.

[142] **Wennink, M. (1995)**: Algorithmic support for automated planning boards, Ph.D. thesis, Technical University Eindhoven, The Netherlands.

Index