SWISS FEDERAL INSTITUTE OF TECHNOLOGY LAUSANNE
EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE LAUSANNE
POLITECNICO FEDERALE DI LOSANNA

LSL/EPFL Laboratoire de systèmes logiques
INN-Ecublens CH-1015 LAUSANNE, Switzerland

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

PhD Thesis No 2541

Cellular Automata and other Cellular Systems: Design & Evolution.

March 1^{st}, 2002

Candidate: Mathieu Capcarrère
PhD Supervisor: Prof. Moshe Sipper
President of the Jury: Prof. Paolo Ienne
Members of the Jury: Prof. Bastien Chopard
Prof. Dario Floreano
Prof. Marco Tomassini

PhD Abstract

Nature abounds in examples of cellular systems. From ant colonies to cellular tissues, from molecular systems to the human brain, cellularity seems to be the way Nature operates. The brain, surely one of the most complex objects to be found on earth, is the quintessence of a cellular system: a huge number of simple elements with an extremely high local connectivity and deprived of any sort of central control, giving rise to a rich global behavior. Cellular interactions thus seem to be the basis for complex phenomena, exhibiting qualities often missing in human artifacts : robustness, self-repair and, more generally, adaptability.

The goal of this thesis is to answer the following question: "What may be computed in cellular systems ?". This question is far from obvious and implies many interrogations such as how to obtain the aforementioned qualities, how to program such systems, and, more fundamentally, what does computation mean in a cellular system.

This thesis is mainly centered around the abstract and formal model of *Cellular Automata*. Through the study and the resolution of different tasks by means of evolution or mathematical demonstrations, I will show that it is not unreasonable to expect artificial systems to exhibit some of the qualities of natural systems, and that (guided) artificial evolution is surely the best approach to define the local behavior of elements which, when grouped as a cellular system, give rise to a desired global behavior. Above all, I will argue that truly emergent behavior in such designed systems is only a matter of perspective.

Résumé de la thèse:

Les exemples de systèmes cellulaires dans la nature sont innombrables. Ils sont présents à tous les niveaux: des colonies de fourmis aux tissus cellulaires, des systèmes moléculaires au cerveau. Le cerveau, justement, l'une des structures connues les plus complexes, est la quintessence du système cellulaire: des éléments simples, excessivement interconnectés, et dépourvus de contrôle centralisé, donnant naissance à un comportement global très riche. Les systèmes cellulaires semblent donc être à la source de nombreux phénomènes complexes. Or ils présentent souvent des qualités inexistantes dans les productions humaines: la tolérance aux fautes, l'auto-réparation, et, plus généralement, la capacité d'adaptation.

Cette thèse a pour but de répondre à la question du traitement de l'information dans les systèmes cellulaires. Cette problématique est loin d'être évidente et contient en elle différentes interrogations liées à l'obtention des qualités sus-mentionnés dans des systèmes construits, ou encore à la programmation de tels systèmes. Au-delà de ces problèmes d'ordre «pratique», la question fondamentale porte sur la signification même de calcul, lorsque celui-ci est effectué par de tels systèmes.

Nous nous attacherons surtout au modèle abstrait et formel des *automates cellulaires*. À travers l'étude et la résolution de différents problèmes, tant par évolution artificielle que par démonstration mathématique, nous démontrerons que les qualités présentes dans les systèmes naturels peuvent se retrouver dans les systèmes artificiels ou encore que l'évolution artificielle (guidée) est sûrement le meilleur moyen de retrouver le comportement local d'éléments, qui, mis ensemble, produisent le comportement global désiré. Surtout, nous argumenterons qu'un traitement de l'information réellement émergent, dans le cadre des automates cellulaires, n'est qu'affaire de regard.

Beuveurs
tresillustres
&
vous
Verolés
tresprecieux
-
car
à
vous
non
à
autres
sont
dediés
mes
escriptz [1]

[1]Most noble boozers, and you my very esteemed and poxy friends - for to you and you alone are my writings dedicated - (Translated by J.M Cohen, Penguin 1955).
Rabelais, *Gargantua*, Prologue de l'autheur, in [157].

Il n'y a pas un point où l'on puisse fixer ses propres limites,
de manière à dire: jusqu'ici c'est moi.
Plotin, Ennéades[2].

Remerciements et autres futilités

Les remerciements d'une thèse s'adresse à deux publics différents, mais pas forcément distincts. Le premier, professionnel, aguerri aux choses de l'informatique et intéressé par l'objet que vous tenez en main, a lu cette thèse, et même, pour certains, l'a décortiquée. Ce public-là mérite ces salutations publiques pour avoir constamment, ou ponctuellement, amélioré, guidé, corrigé, compris, suggéré, en un mot, enrichi ce travail. Le deuxième public me touche plus personnellement. Les 453 grammes de papier que vous tenez entre les mains argumente de l'importance des interactions. Cette thèse est le fruit de mille interactions sociales. Même si elles ne sont pas toutes citées ici, et je m'en excuse déjà, les personnes qui le sont ont largement participé à celles-ci.

D'abord et avant tout, il convient donc de rendre hommage à ceux qui ont lu ce travail, et au premier d'entre eux, mon directeur de thèse, le Professeur Moshe Sipper. C'est grâce à lui que j'ai découvert l'évolution artificielle et les automates cellulaires au cours d'un stage il y a un peu plus de six années, et c'est sous sa houlette que j'ai accompli les travaux ici présentés. Sa relecture attentive de toutes et chacune des pages qui suivent les rendent aujourd'hui un peu plus lisibles et un peu moins fautives. Qu'il en soit remercié. Je tiens ici aussi à rendre hommage à mes rapporteurs: – le professeur Marco Tomassini co-responsable de mon goût pour les automates cellulaires et les algorithmes évolutionnistes, et relecteur inflexible; – le professeur Bastien Chopard, physicien et commentateur avisé de ce travail; – le professeur Dario Floreano, dont les travaux sur la robotique évolutive sont à la base de mon intérêt pour la Vie Artificielle en général, et dont le point de vue externe au domaine fut l'un des plus «dérangeant» – et enfin, le professeur Paolo Ienne, remplaçant méritoire de notre ex-directeur de département, le professeur Hersch.

Le lieu où ce travail fut accompli, le Laboratoire de Systèmes Logiques, ou, plus précisément, les gens qui y travaillent ou y ont travaillé ont apporté beaucoup à ce travail et à son auteur. Le professeur Daniel Mange, directeur émérite de ce lieu, chercheur passionné et figure paternelle incontournable, nous fournit à tous salaire et conditions de travail exceptionnelles. Quiconque a vu la fameuse feuille de répartition des salaires lui en témoignera gratitude. Sa passion des trains (à l'heure) et des montres (Suisses) s'est parfois heurtée à un certain flegme matutinal – adaptation trop parfaite au quart d'heure vaudois ? – mais qu'il soit ici remercié de son extrême humanité. Pour quelques raisons historiques obscures le LSL ne compte pas un professeur mais trois. Aux côtés de Daniel, se trouvent, le professeur Zahnd, JK, à sa droite et le professeur Eduardo Sanchez, à sa gauche. JK, combien de fois

[2]Edition de Plotin, [154], cité in [17], p.29.

ai-je silencieusement «béni» ton aptitude à la modération des séances volubiles ou apprécié tes remarques acides, l'air de ne pas y toucher. Eduardo, représentant underground du M19 en Suisse, tes apparitions au laboratoire à des heures indues en compagnie de ta placidité et bonne humeur habituelle – sauf peut-être lorsqu'une balle de squash heurte la ligne ? – ont adouci les moments d'écriture nocturne de ce mémoire. Soyez tous deux remerciés de votre influence sur l'ambiance du lieu. Au LSL, comme ailleurs, les trois mousquetaires sont quatre. Je tiens donc aussi à remercier André Stauffer, le talent graphique du LSL et compagnon fort agréable des nuits praguoises.

Après les hautes sphères académiques, il convient de parler des petits, des obscurs, des sans-grades, eux qui travaillent fourbus, blessés, affamés, malades, avec espoir de doctorats et de dotations. Eux qui travaillent toujours et jamais n'avancent... et vice-versa. Pour ces remerciements, l'ordre sera chronologique et topique, comprenne qui pourra.

En premier lieu, il convient de rendre hommage au plus ancien d'entre tous. Gianluca, l'homme aux mille bières, cruciverbiste et angliciste très impressionnant, défenseur intraitable de la pause syndicale et soutien musical appréciable (ah Leonard C.), ton amitié fut la première en territoire helvétique. Ensuite vint Emeka, expert-ès-distillation de houblon belge, entrepreneur impénitent (la sonde reste dans les mémoires). C'est grâce à toi, malgré ta jovialité usuelle, que j'ai appris que l'on pouvait être quinteux sans tousser et que j'ai pu enfin mettre un terme sur ma procrastination. Enfin, Dom, le quatrième du 235, complice politique, ami indéfectible, partenaire de Blast toujours regretté, esthète pur malt et fine bouche, sache que l'Hôtel de ville nous attend. Soyez tous trois remerciés pour avoir créé une atmosphère unique faute d'être studieuse. Mais les doctorants doctorent et les têtes changent. Ainsi sont arrivés Enrico et Yann. Récupérant un joueur de volley-ball et un basketteur semi-professionnels, le laboratoire multiplia ainsi par trois son contingent de sportif. Enrico, l'enfer pour un francophone doit se trouver quelque part entre deux italophones. Yann, Frank Zappa est mort... Désolé. Merci à tous deux d'avoir (avantageusement ?) remplacé Moshe sur la liste Jokes. En nous déplaçant de quelques mètres, nous trouvons les valaisans, espèce étrange se nourrissant uniquement d'abricotine et de fromage fondu. Jean-Luc, merci de m'avoir fait découvrir le ski de rando et la montagne octodurienne (même si je t'assure que la montagne ailleurs est jolie aussi). Jaco, nous n'avons pas gagné contre Métrociné, mais je compte sur toi pour continuer le combat contre Europlex. Dans le bruit indescriptible que peuvent produire les deux sus-mentionnés jouant au Blast, se tenait Fabio coi, discret, serviable et aimable comme à son habitude. Son statut de père peut seul expliquer cette capacité surhumaine à s'abstraire des contingences sonores. Puis vint Christof, travailleur infatigable, chercheur obstiné, l'homme qui publie plus vite que son ombre, dont la seule vue suffit à culpabiliser le plus consciencieux d'entre nous. J'ai apprécié, et j'apprécie encore les conversations que nous avons. Enfin, pour prendre un peu de hauteur, le LSL s'est judicieusement muni avec Fabien d'un aérostier béarnais, pays délicieux du Jurançon et du pâté maison. Merci Fabien d'avoir apporté un peu de sel aux verrées du négoce. Mais l'ambiance du Laboratoire ne serait pas sans l'armada colombienne. Nous avons déjà évoqué leur chef suprême Eduardo, et l'un de ses hommes de main, Fabio, mais le compte n'y serait pas sans Andrés et Carlos-Andrés. Philosophes à leurs heures, l'un est amateur de robotique et d'intelligence artificielle au sens premier, l'autre est féru de logique floue, que celle ci soit informatique ou politique et sociale. Merci à tous deux des longs échanges passionnants que nous avons eus. Finalement, Ralph, étant le plus jeune des plus éloignés, se retrouve à la fin de cette liste, et pourtant tu fus un partenaire de Blast et de Squash très apprécié (et pas seulement lorsque tu perdais).

Finalement le LSL ne serait pas sans Marlyse ni Chico. Organisatrice impeccable et d'un gentillesse toujours égale, jamais Marlyse ne fut à cours d'une solution. Merci pour ton aide et ta patience. Jongleur au fer-à-souder paraît-il, mais que je connais plus pour ses jeux de mots laids, son acharnement à recréer la petite boutique des horreurs dans notre salle de réunion et son maniement de la caisse occulte, Chico, je dois te dire que malgré tous tes efforts, tu n'arrives pas à être insupportable.

Ce séjour en Suisse m'a permis de découvrir une image assez lointaine de la carte postale mais fort sympathique, faite de bars, de théâtre amateur, de militantisme politique et de mille autres choses encore. M'a accompagnée dans les premiers beaucoup de monde mais plus particulièrement Sandrine, Bossetteuse à nulle autre pareille, et compagnonne d'infortune. Merci de ton amitié. Je tiens aussi à remercier les Polyssons, troupe enthousiaste qu'aucun défi n'arrête. La joyeuse ambiance des représentations me manque déjà. Parmi tous ces histrions herbeux, je tiens à remercier Caro pour nos longs papotages en tous lieux et à toutes heures. Mention politique, mais aussi amicale, je tiens à remercier les membres de Regards Critiques, et tout spécialement, George et Carmen, Carola et par extension Igor, Antonin, Marc, et biens d'autres encore, démonstrations que l'extrémisme est toujours nécessaire. Enfin, une mention spéciale pour Joce, toujours présente quand il a fallu.

Just a little bit of English to thank the few people from IC that still matter. Thanks to Jon, Simon, Karen and Torbjørn. Torbjørn, the discussion with you influenced most of the research in this thesis and trust me: representation is useless.

Encore une note pour remercier la clique restée en France, et en particulier Eric et Christophe, amis indispensables.

Last but not least, je tiens à remercier ici ma famille, soutien indéfectible et irremplaçable. Ma mère, d'abord, à qui je dois, tout simplement, tout. Mon père ensuite, partenaire de discussions intenses, mais père avant toutes autres choses. Mon frère, enfin, questionneur attentif de mes travaux et modèle d'humanité. Je tiens aussi à remercier mon parrain, figure exemplaire pour cette carrière académique que je commence et à qui je dois, très directement, d'être arrivé ici. Je voudrais encore rendre hommage, in memoriam, à mes deux grands-mères pour m'avoir permis de faire les études que je désirais. Que ma tante, Marie-Germaine, soit aussi remerciée pour m'avoir montré le chemin de la Suisse, mais aussi, pour m'avoir, enfant, distrait avec la science. Enfin, je voudrais aussi remercier Betty et Michel, Marie-Françoise, Jean-Louis et Chris d'être ce qu'ils sont.

Je remercie Anne pour avoir toujours cru en moi, m'avoir encouragé et relancé dans les moments difficiles et aussi pour m'avoir supporté durant l'écriture de ce mémoire. Merci d'être là et d'accompagner ma vie.

Contents

List of Figures

Objection contre la science: ce monde ne *mérite* pas d'être connu[1].
Cioran, *Syllogismes de l'amertume*, ch. L'escroc du Gouffre. [30].
Les philosophes n'ont fait qu'*interpréter* le monde de diverses manières;
ce qui importe, c'est de le *transformer*[2].
Karl Marx, Thèses sur Feuerbach, XI. [126].

Chapter 1

Introduction

What are cellular systems? A precise answer, would require this thesis, and many more to come. But not to lure the occasional reader further than what is necessary, I shall try here to define the scope of our interrogation. A cellular system is considered here to be, simply, many locally interacting elements. The three important notions are *locality, interactions and the number of elements. Locality* implies the absence of global control, which excludes our usual way of thinking, we who consider the brain as a necessary supervisor. *Interactions*: mutual or reciprocal actions or influences, as the Merriam-Webster defines it. More precisely, mutual influence is what we are looking for. Cellular systems are not made of independent elements, but, on the contrary, of deeply interconnected elements. It is through interactions that a global behavior may emerge from the system as a whole. *The number* is a rather vague notion. What is enough? What is too few? The importance of the number is not so much a question of quantity but rather a question of richness, of potentialities of the system. It is often in its redundancy, multiplicity, that a cellular system find its qualities which go beyond the qualities of each individual elements. This may seem like much constraints and one may wonder why should one use cellular systems.

Why use cellular systems? One of the prime motivations in this thesis is that Nature abounds in examples of cellular systems, from ant colonies to cellular tissues, from molecular systems to the human brain. Actually, though it is natural for us to design globally controlled systems, as a reproduction of a certain model of the human body that we feel as controlled by a unique instance, our brain, this is not the way natural systems proceed. This is quite remarkable as the brain, as far as we know, is not a monolithic controller but rather the quintessence of a cellular system: a huge number of simple elements exhibiting extremely rich local connectivity and deprived of any sort of global control. Cellular interactions seem thus to be at the foundation of the complex phenomena. Actually, aggregation and collective working is more and more often purported as one of the basic, fundamental structures of "progress" in the universe. Elementary particles, quarks, interact and aggregate

[1]Objection against science: this world does not *deserve* to be known.
[2]Philosophers did only interpret the world diversely; what matters is to transform it.

to 'become' hadrons; hadrons aggregate following the *strong force* to become atoms; atoms, molecules; molecules, cell; cells, body; bodies, society, etc. It seems that collective behavior resulting from local interactions eventually brings out a new level of complexity that supersedes previous structures. Though speculative on physics grounds, this point of view is strongly backed when one considers biology. Multi-cellular organisms present a complexity, a robustness, and a wealth of potentialities unknown to unicellulars. To take an objective metrics, as shown in Fittkau and Klinge [50], about one-third of the entire animal biomass of the Amazonian rain forest is composed of ants and termites. Moreover, along with bees and wasps, these insects comprise more than 75% of the total insect biomass. Thus, social insects, that is, ants, termites, bees and wasps, constitute an amazing ecological success. What makes Nature's examples so interesting is that they demonstrate surprising qualities usually nonexistent in human production: robustness, self-repair and more generally adaptability. And there are good reasons to believe that these qualities come from cellularity, at least that cellularity is a necessary condition. First, all elements work in interactions but the whole system does not rely on any one of them as there is no global controller; thus none are a priori essential. Second, the number of elements allows simplicity of any one of them without impairing the global potentialities of the system, and simple elements make for an easier replacement. Finally, cellular systems do not rely on a minute number of elements, the arrangements are not brittle and thus adapt "easily" to a dynamic noisy environment as is the case in Nature. So cellular systems have nice qualities, but, though an intermediate goal may be to reproduce these, what we seek is to get these qualities in a system that solves any desired problem. Then the question is how to get the cellular system to solve a given problem. Though this question is a difficult one and is implicitly or explicitly evoked throughout this thesis, we have first to answer the question of what does it mean for a cellular system to solve a problem. This may not seem at first sight a particularly tough question, but it turns out to be central to the "how to" interrogations.

What is computation by means of a cellular system? This question is what this thesis tries to answer, or at least give pieces of an answer. Here, I will only try, in a few lines, to give a feeling for the problem. Considering the computation of the system as a whole, we concentrate on the global behavior of the system. This brings us to a first distinction of importance. The workings of a cellular system, when this system is designed, is described in terms of the local interaction function(s). On the contrary, the desired goal, the problem-solving capability, is expressed as a global behavior. This difference is an important one when the vocabularies, the levels of description of the two behaviors, are in different categories. For instance: one may talk of H_2O or of *water*. In one case, we talk at the cellular element level, the molecule, in the other, at the system level, the macroscopic level. There is no clear link between one and the other. It does not mean anything to say that one molecule is liquid. The fact that there is the necessity of distinct levels of language is actually the definition of an *emergent* global behavior according to some (Luc Steels for instance,[197]). The question of emergence is a vast one that I will not tackle now, but this difference of levels of language, this gain between the single elements of the system and the system as a whole is what defines an interesting cellular system. To come back to the question of what is computation by means of cellular systems, we have to wonder what we mean by problem

solving, especially in the case of such "interesting" cellular systems. In living systems, the global behavior is the "output" of the system, the "input" being the external constraints, and the problem this "computation" solves is survival. To take a specific example, let's take a wasp colony whose "output" may be considered to be a nest. The only objective behaviors of the system are that of single wasps' behavior. The fact that a nest is constructed is only the emergent, global behavior. As outside observers, we see the nest which is protecting the colony, and deduce that to solve the problem of protection, wasps build nests. However, this is not obvious at all, and from each wasp's point of view, it is its own behavior that insure reproduction, the final effect of the global emergent property that is a nest. This may seem like a fallacious argument either because the result of wasp building strikes us as a nest or because this is not specific to cellular systems. The way one envisages a problem always affects the possibility of finding a solution or not, but the question here is far more pregnant. The global, emergent behavior, especially in designed systems, is only a figment of our imagination, even if the realization is tangible. The only thing one knows for sure is the local behavior. Whereas in classical systems we design a system to fit the problem, in cellular systems we design at the local level; we design the immediately accessible behaviors of the system, but we wish to obtain the computation result at a different level, at a global level. Hence we come back to the original problem of linking the local and the global behaviors, but adding the supplementary constraint that this global behavior is not given as such. In the abstract computing model of cellular systems such as cellular automata that will be studied throughout this thesis, this question is particularly important. It is our view as a human of the system that determines the quality of its global behavior. In effect, it is the system in itself, by its own global state that solves the problem. Here, the question of what is computation is thus really how should we look at the system. The system may very well solve the problem, in a certain way, but we may be drawn into the local details, or look "wrongly" at the behavior, from the wrong perspective. The fourth chapter of this thesis actually argues that computation in cellular systems is really a question of visual efficiency. "[I] have to act like painters and pull away, but not too far. How far then? Guess!"[3].

How do we design cellular systems? What we are looking for are locally interacting systems producing global behavior surpassing the capacity of each individual, namely, *emergent cellular systems*. As we saw, this emergence in itself may not be obvious, and requires the correct look. So, is it at all possible, given a problem, to design a cellular system that will solve it? Is it possible to derive the local interactions of a cellular system so as to get the desired global behavior? This is a very difficult task, and even in very formal systems, such as cellular automata, studied in this thesis, mathematics leaves us powerless, missing some sort of a universal solution. However, there are two ways studied in this thesis to make that missing link between local and global behavior. The first one, still following Nature's model, is evolution. That is to say, devise through artificial evolution, the local interaction function(s) by selecting the individuals according to the global behavior of the group, of the whole cellular system. The usual mechanism of (artificial) evolution is to work on a genotype but to select a phenotype, thus it maps naturally here to our problem: we work at the local

[3] «Il faut que je fasse comme les peintres et que je m'en éloigne, mais non pas trop. De combien donc ? Devinez!» Blaise Pascal, pensées 479 in [149].

level but evaluate at the global level. The second way to map the local to the global is more mathematical. It consists in stripping down the idea behind the task, to avoid looking at it from too narrow a view, with the wrong perspective, and then derive the maximum of constraints to be able to observe physically or artificially[4] the restrained system so as to evaluate their efficiency on the task.

In summary, this thesis is going to concentrate on the question of computation in cellular systems. I thus try to answer what is computation and how to do computation with cellular systems, how to link local and global behavior, both mathematically and by evolution. I will look also into the question of how to fully exploit cellularity so as to gain robustness.

In chapter 2, I present formally and in detail, the basic, and less basic, principles of this work, namely: – *Cellular automata*, which will be used in all but the third chapter and which are the quintessential model of a cellular system; – *Ontogenetic systems*, which will be used in the third chapter to exploit the underlying robustness quality of cellularity and; – *The Evolutionary paradigm*, which will be used as a tool in chapters 3 and 6 and studied as a process in chapter 5.

Chapter 3 presents an original cellular system developed for this thesis, named *Phuon*. The motivation behind this project was to go beyond classical cellular systems, such as cellular automata. As we will see in the following chapters, these are powerful while remaining conceptually "simple". However they usually lack adaptability, and are very brittle to synchronization and general failure. The idea here was to add ontogeny to cellularity, growth and development being means of adaptation and thus robustness.

In chapter 4, I will tackle the question of what is a problem-solving cellular automata. Through the study of mostly mathematical properties of both uniform and non-uniform CAs, I am going to provide some answers, at the price of limiting the scope of this question. More precisely, by concentrating mainly on the density classification task, I will ponder what computation by means of CAs really is, and argue that it is *'visual'* computation. Moreover, I will propose a new definition of emergent global behavior in the limited scope of non-uniform CAs. This chapter constitutes the core of this thesis, in the sense that it tackles directly the problem underlying the other chapters.

In chapter 5, I propose to study statistically a peculiar kind of evolutionary algorithm. The motivation of this work is quite evident when we think about the lack of understanding of the inner workings of evolutionary algorithms in general. I will concentrate on a special kind of algorithm: Structured, fine-grained parallel evolutionary algorithms. To do this, *the cellular programming algorithm*, presented by Sipper [180], will be studied.

Finally, in chapter 6, I will come back to the question of robustness, but in the restrained scope of Cellular Automata. Life was an awe inspiring model for the first Cellular Automata designers. I believe that robustness, an ever present quality in living systems, was omitted more for practical reasons. obustness to faults and robustness to asynchronous dynamics will be studied in this chapter. This study will also be the occasion to deepen our understanding of the information contained in a cellular automata.

I conclude this thesis in chapter 7 where I come back on the question of what is computation in emergent cellular automata.

[4] This is not done in this thesis though.

Like all people who try to exhaust a subject,
he exhausted his listeners.
Oscar Wilde, The picture of Dorian Gray, ch.III. [221]

Chapter 2

Cellularity, Ontogeny, and Evolution

2.1 Introduction

The readership of this thesis may come from very diverse backgrounds, ranging from people with a broad interest in A-Life methodology to physicist studying density conserving cellular automata as traffic-flow models. This chapter presents the basic, and less basic, principles of this work, namely, *Cellular automata*, *Ontogenetic systems* and the *Evolutionary paradigm*.

In section 2.2, we detail what Cellular Automata are. Then, through a little bit of history, we will give a glimpse at both their importance and diversity. Finally, the question of behavior classification will be the occasion to hint at one of the issues of this thesis: going from global to local and vice versa. Cellular systems in general will not be overviewed in this section for two reasons. The first reason is that most systems of interests to us are ontogenetic, and will thus be treated in the following section. The second reason is that the field as such is far too large to be covered in any meaningful way here. It will be better served by a restricted development at the occasion of the presentation of our own cellular system, *Phuon* , in chapter 3.

Ontogenetic systems, or systems that develop, are not so common in computer science. In section 2.3, after a brief presentation of our own definition of such systems, I propose to present some examples of such studies. Though most of the work in the literature concentrated on the morphogenesis aspect, I will try there to give an insight on works that go beyond. The systems presented are both abstract and realistic. As we will see, their purpose is as well understanding biology as problem solving.

Finally, in section 2.4, we will return to more familiar grounds for computer scientists, at least for the A-life community, and describe some of the most common evolutionary techniques. To avoid babbling over too large a subject, I will specifically concentrate on the line of this thesis, Genetic Algorithms and Genetic Programming. The question of the efficiency of such algorithms will be briefly addressed through the presentation of the schema theorem, a question related to the study of the evolutionary cellular algorithm presented in chapter 5.

2.2 Cellular Automata

There are many (perhaps too many) cellular systems developed by the Artificial-Life community. Moreover systems of interest from our view point encompass also many of those originating from the Artificial Intelligence or from the Neural Network communities. In effect, neural networks as such, but also some works on agents could be considered as cellular systems. So what are cellular systems ?

To adopt an intuitive definition, we could say that any system composed of a multitude of simple quasi-identical elements interacting locally with each other and lacking global, central control is a cellular system. This characterization, however simple and however large the domain it covers, reflects the two quintessential properties of such a system: massive parallelism and no global control. Nevertheless, it would be pointless to start here a review ranging from cellular phones to massively parallel computers via quantum computing, resulting in too long a list. More reasonably, I will present here only the cellular automata model, the quintessence of any cellular system, leaving the works that were the most significant when we defined the concepts underlying *Phuon* to section 2.3 and chapter 3.

In the following section, I will thus concentrate on cellular automata, a model which in its infinite variations nourished most of this thesis, and percolated through every chapter. However, I should warn the reader that this will not constitute in any way an exhaustive review of cellular automata and that many results will be presented in the succeeding chapters for the sake of simplicity and understandability.

2.2.1 Cellular automata

Cellular automata (CA) is surely the best-known – and most thoroughly studied – abstract cellular system model. Its main qualities are simplicity, a rigorous mathematical definition, and, paradoxically, a potential wealth of complex behaviors. The attractiveness of this model was such that since its first formal characterization by von Neumann [139], it became an interdisciplinary object of study, from computer science to biology, from mathematics to physics. Its importance in discrete mathematics, the mathematics of computing, was deemed by Toffoli and Margolus [210] to be as high as differential equations for the mathematics of the continuum. Wolfram argued numerous times of CA's importance in modeling physics and, if we are to believe the rumors about his latest book [228], it could be *the* complete model for the universe. Whatever these declarations, in this thesis I shall, more modestly, concentrate on Cellular Automata as a computing model: a model of massive parallelism, a model suited for hardware implementation, and a model suited for the study of Artificial Life problems.

So what are Cellular Automata ? I shall first answer this legitimate question in the forthcoming subsection. I will then look at the past to understand the variety CAs offer, and how this model lies naturally in the A-Life field. Finally, before concluding with some variations of the cellular automata model, I will evoke the attempts at classification of the global behavior of CAs.

What are cellular automata ?

Cellular automata are dynamical systems discrete both in space and time. *In space*, a CA is a collection of finite state automata, distributed over the nodes of a regular, geometric structure, *a lattice*. There is an automata at each and every node of the lattice. This topological structure induces the connections in-between automata. Usually, each automaton is connected to every other automaton at a pre-determined distance, usually called radius, along each dimension of the lattice. *In time*, each automaton updates its state synchronously with all other automata. This update is done according to a fixed mapping from the present states of the automaton itself and of the neighboring automata to its future state. Each automaton applies the *same* mapping function. The neighborhood is usually composed of the automata to which it is connected. In two dimensions, for instance, we say that a CA uses the von Neumann neighborhood when each automaton is connected to its immediate two neighbors in each dimension, left and right, up and down. Each automaton can be in one of any **finite** number of states.

To make things clearer, consider the simple example of a two-state, one-dimensional, linear CA with radius 1 (Figure 2.1). In this case, the topological structure is an array of dimension one, and thus the neighborhood at distance one is composed of the immediate right and left neighbors. Each automaton can have only two states, 0 or 1. Hence the mapping function is $f : \{0,1\}^3 \rightarrow \{0,1\}$. In such a simple case, it can also be expressed as a rule table, and even more simply by the output bit string of this rule table in the order of Figure 2.1. This is called the rule of the CA. Wolfram, in [226], who concentrated on such $1d$, radius 1, 2-state CAs, so-called *elementary* CAs, proposed this scheme to characterize all 256 such CAs by a decimal number corresponding to the binary one formed by the output bit string, i.e., CA 232 refers to transition rule 11101000.

.

Rule table:

neighborhood:	111	110	101	100	011	010	001	000
output bit:	1	1	1	0	1	0	0	0

Grid:

$t = 0$	...	1	1	0	1	0	1	1	0	1	1	0	0	1	...
$t = 1$	...	?	1	1	0	1	1	1	1	1	1	0	0	?	...

Figure 2.1 Illustration of a one-dimensional, 2-state CA. The connectivity radius is $r = 1$, meaning that each automaton has two neighbors, one to its immediate left and one to its immediate right. The rule table for updating the grid is shown on top. The grid configuration over one time step is shown at the bottom. This figure is based on Mitchell [131].

In theory, the lattice is infinite, implying for instance that the set $\mathbb{N}$ of integers is isomorphic to 1d CAs. However, in practice, a CA is always bounded. There are two

standard ways to deal with borders: *fixed* or *periodic* boundaries. **Fixed** boundary conditions can be of two sorts: 1) The missing input automaton at the boundaries, i.e. the left (resp. right) neighbor state for the left-most (resp. right-most) automaton for the 1d case, is replaced by a fixed input to the transition function. For instance in a 1d, 2-state CA, a fixed 1 at the right mimic an infinite string of 1. 2) The fixed boundary problem may be solved by introducing $2 * d$ special rules for d-dimensional CAs, one at each border taking only itself and its available neighbors as inputs. **Periodic** boundary conditions are used most often. This is due to their simplicity, natural mimicry of infinite lattice and conservation of uniformity. This boundary condition simply consists in suppressing the boundaries. By this we mean folding each dimension, so that one bound in dimension d is connected to its counterpart. For instance in dimension 1, we fold a segment so as to get a loop. In dimension 2, we fold a matrix so as to get a toroid (a doughnut). One may see the example of the latter "folding" in Figure 2.2. These two forms of boundary condition always result in very different behaviors of the CA, though as one may see in Figure 2.3, using the same general rule, the computational power may be preserved.

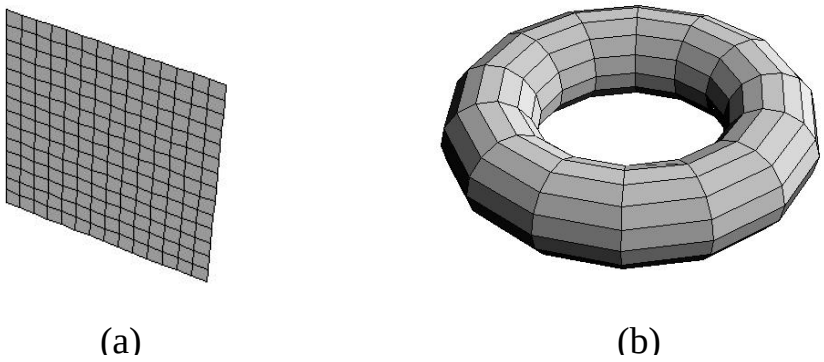


Figure 2.2 Example of a fixed border, 15x15 CA (a), and the same CA with periodic boundaries (b) .

Hence, CAs exhibit three notable features, namely, massive parallelism, locality of cellular interactions, and simplicity of basic components (cells). As such they are naturally suited for Artificial Life studies, living systems exhibiting the same features, and well suited for hardware implementation, with the potential of exhibiting extremely fast and reliable computation that could be as robust to noisy input data and component failure as living systems are. A major impediment preventing ubiquitous computing with CAs stems from the difficulty of using their complex behavior to perform useful computations. Designing CAs to exhibit a specific behavior or to perform a particular task is highly complicated, thus severely limiting their applications. This results from the local dynamics of the system, which renders the design of local rules to perform global computational tasks extremely arduous. In this thesis, we will both try to automate this design, thereby following the footsteps of Mitchell *et al* [132] and Sipper [180], and prove some mathematical results useful to design these complex machines.

A question of vocabulary

CA or cellular automata will be used interchangeably to refer to all automata, with their current state, the topology of the geometrical structure on which they are distributed, the neighborhood and the local transition rule. An automaton is usually called *a cell* in this thesis. Hence, the plural will be cells and the term automata will be reserved as an abbreviation of cellular automata. The *global configuration* is simply the state of all the cells at a given time. One should keep in mind that this information is geometrically structured. For instance, the global configuration of a two-dimensional CA is a matrix. However, if we talk about a toroidal CA, any configuration that may be attained by translation, or a composition of translations (in dimension greater than 1), is identical. The configuration at time 0 is the *input configuration*. When the neighborhood is regular, namely identical along each dimension, the *radius* is defined as the number of neighbors along each dimension, on one side. So if there are d dimensions, the number of neighbors of a CA with radius r is $2dr + 1$.

A short historical tour

It is hard to pinpoint the exact origins of Cellular Automata. As with many mathematical notions, it is quite sure that they were reinvented several times under different appellations before being commonly accepted. Nevertheless, it is the habit to attribute the founding ideas to Stanislav Ulam [214] and John von Neumann [139] in the fifties, even though, according to Toffoli and Margolus [210], the mathematical structures developed by Zuse [236] are cellular automata that are not named as such. These official "parents" are of particular significance for the development of the field. If we have earlier underlined the mathematical importance of the field, the point of view of Stanilav Ulam one could say, von Neumann paternity on the other hand lead naturally to the Artificial Life development of Cellular Automata.

The original study of von Neumann could be summed up as: Can machine self-reproduce like living organisms ? Hence, I could say *now* that Cellular Automata from their first formalizations were developed from an A-Life approach. In fact, the idea of von Neumann was to bring the axiomatic and deductive treatment to the study of biological systems. The central question was self-reproduction, but not in a naive way as he points out in the fifth lecture given at the University of Illinois in 1949: "One of the difficulties in defining what one means by self-reproduction is that certain organizations, such as growing crystals, are self-reproductive by any naive definition of self-reproduction, yet nobody is willing to award them the distinction of being self-reproductive"[1]. The original model, theoretically developed by von Neumann in the fifties, using a cellular structure at Ulam's suggestion, was a 2d, 29-state, infinite CA. This CA structure is basically made of one main part, a universal constructor. This constructor can take an input string describing any structure in the CA and replicate it elsewhere on the CA grid. It is universal in the sense that any structure can be described as an input string, including the constructor itself. This simple and clever idea allows this system to go beyond what seemed to be an intuitive limit. In fact, it answered one of the first of von Neumann's questions: is it possible for a machine to

[1] In the works edited by Burks [139], p.86

construct a machine as complex as itself? There he had created one able to create machine of any complexity, even *more* complex than itself.

This original milestone remained only an abstract work on paper (at least until 1995, see Pesavento [151]), but the ideas attracted a lot of interest, not only for CAs but also for the question of self-replication/self-reproduction, as one may judge from the Sipper's review of self-replication [183]. One should note that these works did not restrain itself to the cellular automata. For instance, Laing [105] proposed a string system, far less structured than CAs, which stemmed from his earlier work [104] which was directly inspired by DNA molecular interactions. This was some twenty years before the seminal paper of Adleman [3] on DNA computing. Closer to our interest, the idea of self-replication in CAs motivated many works, mainly centered around the question von Neumann exposed from the start: what constitutes non-naive self-replication? The idea of Burks was that the self-replicating automata should have universal computation power (i.e., the power of a universal Turing machine). As early as 1973, Hermann [82], presented a very simple self-reproducing CA structure capable also of universal computation. This is due to the fact that a universal Turing machine is actually very simple and thus a single automaton with few states (less than von Neumann's 29) is capable of universal computation. Replication is then just cell duplication, and thus trivial. I will leave here the question of self-replication (see chapter 6 for further discussion of this theme). However, this quest of what constitutes a complex, interesting, some would say emergent, process in cellular automata has permeated through all the past and current research, and through this thesis.

The work of von Neumann opened more than the self-replication problem, and developed into the vaster field of Cellular Automata. These studies started with their application to optimization problems by John Holland [83] and, two years later, the first complete formalization of CAs by Codd [31]. Soon after, Conway's game of life, popularized by Gardner [67], opened the field to a large audience. Strangely, this is through this large non-academical attention, but also through the collected essays by Burks [23] which included works by Moore, Holland, and Myhill, that the field gained its spurs. Although this allowed the development of Cellular Automata research, this newly acquired independence also led the enthusiasts for novelty to forget the mathematical roots of CAs as Transitional Shift Dynamical Systems. Thus some duplication of results occurred. For instance, as noted by Toffoli and Margolus [209], Patt [7] re-proved in 1971 and Richardson [163] in 1972 results presented to the mathematics community by Hedlund [80, 81] in 1963 and 1969. Unfortunately, the probability of duplicate results is no lower today than at the start of CA studies... CA very rapidly attracted a large variety of persons from different fields, but they never really joined forces. The activities of people with an interest for CAs could be divided into 4 main categories: Mathematical theory, Physics modeling, Biological/Social systems modeling, and Computational Systems. This division is often very well marked and communication between the scientists is rather poor. I will briefly cover these four fields.

For the mathematical part, the history of CAs by Sarkar [168], though concerned with the theory of computation, is a good overview of mathematical problems linked to CAs. This review duly acknowledge early theoretical thinkers, such as von Neumann, Arbib, Amoroso and Cooper or Smith ([6, 7, 12, 190]) or current mathematicians working on CAs,

such as Mazoyer or Sutner ([127, 200]). It should be noted that it fails to acknowledge some important theoretical physics thinkers such as Vichniac [218], or even pure mathematician like Fisch [49]. This review is exemplary of the partition of the field, in that Artificial Life works are almost completely forgotten.

The physics modeling use of CAs is surely the most successful sub-domain as a research subject. The use of discrete models (CAs or close variations) in physics appeared early. Greenberg, Gerola and Seiden, Harvey, and Langer presented between 1978 and 1980 results using non-continuous models ([69, 72, 110]). However, Stephen Wolfram's 1983 article [223] is often considered as a major milestone. Its goal was rather modest: "CA are used as simple mathematical models to investigate self-organization in statistical dynamics", but its extensive study of one-dimensional, two-state CA would bring a re-birth of the field and stir great interest from computer scientists. In 1984, Vichniac [217] argued that CAs are more than just a surrogate for partial differential equations and could be used as an exact model for simulation of complex phenomena. More fundamentally, he remarked that CAs exhibit "unmistakably" physical properties and interrogations, such as ergodicity, relaxation to chaos, and time reversibility. They are physical models in themselves. As Toffoli [208] put it in the same issue, the idea was to consider "Cellular automata as an alternative to (rather than an approximation of) differential equations in modeling physics". Still in this special issue of Physica D, Wolfram [225] presented what is considered a landmark paper in the field of CA. The paper concentrates on one one-dimensional, two-state CAs, so called elementary CAs. Firstly, he introduced a classification of the global behavior of CAs based on 4 classes, through the systematic observation of their dynamics (see next section for details). Second, and this derives from the first point, Wolfram introduced the idea that simple CAs are powerful in the sense that they can generate complex self-organization that is neither chaotic nor periodic. This idea of self-organization, and walking on the border line of chaos, exerted a great attraction at the time, and still does. Nowadays, even though the excitement has subsided, the research on CAs as physical models is still active, for instance for traffic flow simulation [47, 61, 137, 138] or crystal and other surface growth modeling [1, 16, 167]. The main results for CAs are still to be usually found in the physics literature. For a contemporary discussion on the use of CAs as physical models, one may see the recent book by Chopard and Droz [28].

Using CAs to mimic biology, as we saw, was the starting point of the field. Was not the idea of von Neumann to formalize living phenomena ? However, we have to distinguish, on one hand, the close modeling of biological phenomena with CAs, and on the other hand, using biological inspiration to study interesting phenomena in CAs. In some sense, these are the two viewpoints of Artificial Life, if we are to consider CA as computational tools. Many people, like Sieburg *et al* [175, 176], have done much work in the field. But we will not discuss the close modeling of biology here as it goes beyond our competence and the scope of this work. For a good discussion on this use of CA in biology, the reader may refer to the paper [46] by Ermentrout. On the other hand, the work inspired by biology is of prime concern for our approach. The distinction we are now making between this paragraph and the next, treating computational aspects of CAs, is rather artificial, and some of the articles presented here could lie just as well below in the third category. After the seminal reflection

of von Neumann on self-reproduction, as we saw, Burks pointed out that the universality-of-computation property of CAs was essential to avoid naive replication. However, in 1984, Langton [112] presented a self-replicating loop incapable of universal computation, stating that this property was anything but essential. On the contrary, he argued that natural (self-)reproduction relied on a *genotype* (a DNA string) that was decoded into a *phenotype* (the acting organism), and that these specificities were the ones to be looked for. His loop exhibiting these two characteristics, it really answered the original question of von Neumann. What is sure is that the article attracted much attention, and I would guess that its visual similarity to a living "thing", the undeniable feeling that something is replicating in front of your eyes, constituted its best argument[2]. Langton's work on self-replication led to several developments in the same spirit: Tempesti [204], who added programmability, Perrier [150] *et al* who added a universal Turing machine, and Chou and Reggia [29], who used duplication to create a fully parallel solver for the NP-Complete Problem, SAT-3. To stay with Artificial Life flavored work, Reggia *et al* [162] also obtained emergent self-replicating structures in CAs through artificial evolution. This is a very interesting example of ALife systems plowing into biology but concerned with results proper to computing. Still in this category, Hiroaki Sayama [170] added an evolvable genome to the self-replicating loops in CAs, thereby allowing more complex structures to emerge.

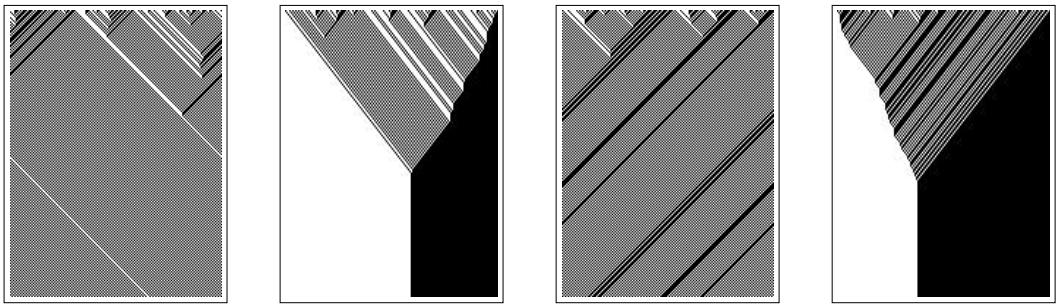


Figure 2.3 We can see in this figure an example of emergent computation in CA demonstrated on the density classification task. The rule in all four figures is 184 in Wolfram's notation, but the boundaries are toroidal in (a) and (c) and fixed in (b) and (d). The left border is fixed to 0 while the right border is fixed to 1. In (a) and (b) there are more 0s than 1s while it is the contrary in (c) and (d).

The computational category of works, one would expect, should be the main part in a computer science thesis, but the presentation here will be just a quick overview, and most of the material will appear in the subsequent chapters. Ultimately, CAs are but a computational tool, and if we except the mathematical theory, all the other domains are just using CAs as a simulation tool, and so as a computational tool (even if physics tried to go further). There are four ways to view the computational aspects of Cellular Automata.

[2]Barry McMullin [128] presented an excellent paper on von Neumann quest for complexity, in which he argued that Langton's loop, however interesting, was no answer to the original question.

The first and most natural one is to consider them as abstract computing systems such as Turing machines or finite state automata. These kind of studies consider what languages CA may accept, time and space complexity, undecidable problems, etc. There has been a wealth of work in this field [36, 37, 41, 91, 224], most of which is based on or completes the mathematical studies of CA. *The second way* is to develop structures inside the CAs (a specific initial configuration) which are able to perform universal computation. A prime example of this is the *Game of Life*, [67], which was proved to be a universal computer using structures like gliders and glider guns to implement logic gates [18]. This was also applied to one-dimensional CAs where structures were found to be a universal Turing machine [120]. A *third way* is to view CAs as "computing black boxes". The initial configuration is then the input data and output is given in the form of some spatial configuration after a certain number of time steps. This includes so-called soliton or particle or collision based computing. In these systems, computing occurs on collision of particles carrying information [2, 198]. In these works, usually, computation really occurs in some specific cells, i.e., most cells are only particle transmitters and some, often the same, do the computation, this even if factually all cells are identical. Hence, parallelism is not really exploited as such. However, there are CAs belonging to this "black box" type which are quite different. In these, like the density task solvers that we will present and study in chapter 4 and which are demonstrated in Figure 2.3, computation is intrinsically parallel. Their solving property relies on each and every cell. All are involved in the computation and there is no quiescent state as such. The result is then to be considered globally. The frontier between these two sub-categories is not always clear, and in fact, the core of this distinction depends on the ill-defined notion of emergence. The latter type being so, the former not. We will come back to this discussion in chapter 4. Finally, there is a *fourth* way, which is to consider CA computational mechanics. This kind of study concentrates on regularities, particles and exceptions arising in the spatial configuration of the CAs considered through time. This is really a study of the dynamical global behavior of the system. This research does not concentrate on the particular computation of CA. Though it often took as object of study CAs of the emergent type of the third category, it was also applied to CAs with no problem solving aim, dissecting their behavior with no further consideration. There has been much work in this domain, accomplished mainly by Hanson, Crutchfield and Mitchell [35, 77] and Hordijk [88].

In this thesis, we will mainly concentrate on the third kind of study. More particularly on the second type, the research of problem solving, emergent CAs. Nevertheless, our interests and some of our results will lie also in the first and the last category. Though not our prime concern, the interrogation on what is emergent, global behavior will percolate through our work.

Classifying cellular automata behavior

The classification of cellular automata behavior, first initiated by Wolfram, is neither mathematically tight nor complete. The reason we would like to expose it briefly here, aside from its historical importance, is that, as it characterizes more objectively global behavior, it is a prerequisite to any attempt to link local behavior to global behavior, a question that is hovering over this thesis.

Wolfram classified uni-dimensional, two-state CAs into four classes, according to their global behavior through time, starting from a random configuration. **Class I** CAs develop toward an homogeneous state, all 1 or all 0. **Class II** CAs develop toward simple stable or periodic structures (with a short period). **Class III** CAs develop into totally chaotic patterns. **Class IV** CAs develop toward complex localized structures, sometimes, long-lived. Firstly, we can remark that these classes may be applied to any sort of CA. Secondly, it strikes us that though class I and II are objectively defined, class III and IV rely more on a subjective observer. In fact, the popularity of this classification comes from the analogy made by Wolfram between the dynamics of CAs, discrete systems, and dynamical systems described by differential equations. In this analogy, class I CAs correspond to fixed point in phase space of continuous dynamical systems, class II to limit cycles, and class III to chaotic strange attractors. Class IV has no counterpart and is specific to CAs. Computer scientists argued along with Wolfram that only class-IV CAs were capable of universal computation. This last result, however, is only a conjecture. As we will see later in chapter 4, rules from class II may turn out to be interesting from a computational standpoint. Actually, we will conjecture that any rule creating regularity, which can be classified into distinguishable classes, may prove to be computationally effective. Thus compared to the general view, I will not only consider class-IV CAs, but also class II, as "interesting". Many other classifications have been proposed, for instance, a refinement of the Wolfram scheme into 6 classes by Li and Packard [117, 118] or a mathematically tight version (but intuitively more obscure) by Culik and Yu [38].

To try to formalize this class definition and, more importantly, to predict to which class the CA behavior will belong, Langton developed the λ term. The lambda term is defined as follows: $\lambda = (q^n - t_q)/q^n$, where q is the set of all possible state of the automaton, n is the size of the neighborhood (thus q^n is the number of entries in the transition table) and t_q is the number of entries of the transition table which map to the quiescent state. The quiescent state is arbitrarily defined. This measure was first introduced in 1986 [113], but it is in 1990 [111] that Langton argued for the "clear" correlation between Wolfram's classes and the lambda parameter. As we can see this measure does not apply only to elementary CAs and, actually, Langton argued for its prediction efficiency, through the study of $q = 4$, $n = 5$, one-dimensional CAs. In fact, he acknowledges that for low values of q and n, the capacity of prediction of the lambda parameter is weak, while it gets better as q and n get larger. However, as this conclusion relies on an exhaustive study of CA behavior, it very soon becomes impossible to check its validity as q and n grow. His main conclusion was that classes I and II match low lambda values and class III high lambda values, while the "interesting class", class IV, lies at critical values between 0.35 and 0.45, that is, in-between periodic and chaotic behavior. Following his argument, he claimed that the real order for the classes should be I, II, IV and III and that computation happens only at the edge of chaos

The attempts to classify "objectively" CA behavior is valuable, in that they are an essential preliminary to developing tools to predict the global behavior from the local rules. Langton's work remains one of the best attempt to solve the question. However, some researchers have criticized the usefulness of the scheme given its average validity and the fact

that it is based on visual control [88, 133]. We can also note that from an adaptation of the formalized classification of Culik and Yu, Sutner [199] deduced that it is undecidable whether all spatially periodic configurations (i.e. on a periodic-boundaries CA) evolve to a fixed point. There are actually many results of undecidability, like this one, precluding any general map from locality to globality. In this thesis, I will argue, most notably concerning the density classification task, that visual observation is at the heart of computation in cellular automata and more generally of any valuable classification of global CA behavior. For this very reason, our critique is that Langton's work does not answer an essential question: what is visual efficiency. Wuensche [231] recently presented a measure, the input-entropy, in an attempt to classify automatically CAs, a measure which may turn out to be a possible answer to this essential question. We will discuss in more depth this point in chapter 4.

Variations: quasi cellular automata

There are many variations around the concept of Cellular Automata, and in fact, we could even argue that the Moore neighborhood is already an heterodox understanding of the original model. The real question is when are the differences such that we can no longer talk of cellular automata. To answer, we will present in this paragraph some of the most common variations still lying in the CA scope, and more particularly those we used and studied in this thesis.

As a first variation we could alter the structure of the lattice. In two dimensions, the original model, if we want the cell to cover the whole space, there are only two different structures possible: a grid with square cells (the usual model) or hexagonal cells on an hexagonal mesh. Each cell then has six neighbors. This has been used to model "hexagonal" phenomena such as snowflake growth [121] or the fly retina [57]. On the other hand, if we are to abandon any visual representation, then obviously it is possible to imagine any kind of structure. However, this kind of theoretical model has never been researched to my knowledge. After the structure, the second characteristic of a CA is its connectivity, i.e., the neighborhood. In a certain sense, this is also a structural variation if we are to consider the connectivity graph as the backbone of the cellular automata. Natural CAs in one and two dimensions could then be classified as the ones with a planar, undirected, connectivity graph. This would include in 2D both von Neumann and Moore neighborhoods, and 6-neighbor hexagonal CAs and in one dimension any CA with any radius. If we keep on with the idea of not distinguishing the topology of our CA and its connectivity graph, then any undirected graph will define a "normal" CA in any dimension, in the sense of a normal extension to the original model. If we introduce the idea of directed graph, then we have one-way CAs[3]. To go further, we distinguish between the connectivity graph and the topology. Then we can imagine various lengths of connections, for instance at distance 2 and 5 but not 3 and 4, with or without non-symmetrical connections or any other kind of neighborhood. In this latter case, the appellation of CA is arguable and would surely rely on the relative locality and regularity of the connections. A third variation, and one of

[3]These CAs were introduced by Dyer [44] and have been the subject of countless studies about their theoretical computational power.

peculiar interest to this thesis, is to alter the uniformity of the automata. Very intuitively, in these non-uniform CAs, each automaton can have its own transition rule. These have been the subject of many studies by Sipper *et al* [180, 182], and is at the center of chapter 5 of this thesis. Finally, the fourth variation may be on the functioning of the CA. It may be a relaxation of the synchronization of the updates, or the introduction of non-deterministic transition rules. These two aspects will be discussed and treated in chapter 6.

2.3 Ontogenetic Systems

Ontogeny: the development or course of development especially of an individual organism. Such is the definition of the Merriam-Webster dictionary. But what is *development* and how does it apply to computer systems ? Morphogenesis which is certainly the most visible aspect of the phenomenon, is often given as an answer to the question. However, though this notion covers the vast majority of the works concerned with development, ontogenesis is not limited to this façade. The works that state morphogenesis as their explicit goal are really concerned with an exterior appearance, namely reaching a certain form. Whereas real developmental systems, ontogenetic systems, enclose all the aspects of development and growth including cell differentiation, adaptation to the conditions, cell lineage, etc. Hence, morphogenesis is an observer point of view while ontogeny is an insider's standpoint. Unlike for most works in computing science, the meaning of morphogenesis in biology includes all the development of structures beneath, but even then, it is just a subpart of ontogeny which comprises development not toward form.

Computer scientists went to look at biology for inspiration on how to get flexibility, adaptation, to gain qualities such as robustness, self-repair and other properties common to all living things but rarely found in engineered systems. In the studies concerned with development however, the understanding of biological reality is often the underlying motive, at least primarily. This is the other side of Artificial Life: trying to understand the quintessential principles of the living systems.

The developmental models presented here may be divided into two classes: the abstract and the realistic models. The former are concerned with extracting the principles of what is development at a very high level. Their goal is often an application to problems rather than the study of the system for itself. The latter try to be as close as possible to reality. The very nature and limitations of computing systems necessitate a certain level of abstraction, even in these latter realistic models, but the prime motivation for these studies is really biological modeling in itself. Nevertheless as we will see, applications sometimes also derive from these.

Abstract models

Surely the most famous work among the abstract systems of development are the L-Systems, conceived by Lindenmayer in the late 70's, [119]. This grammar-based model goes beyond morphogenesis even though it is mainly concerned with form. In fact, it is a good abstraction of genetic control of cell-division and is convenient to describe cell lineage. The system

uses rewriting rules to sequentially modify a string which represents the organism. This system was particularly successful at modeling plant growth (see Figure 2.4 and [156]) and other systems, such as neural networks, [114], or body organs, [43]. The model was later augmented to incorporate some environmental influences and cell-cell interactions by using a context-sensitive language. An even later model included some elements of differential equations to mimic physical forces such as osmotic pressure [144]. This latter model is in-between abstract and realistic conceptions, and could as well have been in the next section. Nevertheless, these systems are explicitly morphogen, and the end result is not due to the internal metabolism of cells but rather to a "constructing program".

Figure 2.4 Example of a L-system to model plant growth. The L-grammar is given at the top left corner. X is the starting case, represented here by a vertical segment. -X (+X) is X inclined $-25°$ ($+25°$ resp.) relatively to its parent. This figure is adapted from figure 1.24a in [156]. To the right, we can see the results of a more complex grammar, and a much more complex graphical mapping.

Another interesting work of abstract ontogeny is Gruau *et al*'s [74] developmental encoding of neural networks. This model is interesting for two reasons. The first one is that it is an example of development that disregards shape — it is not morphogenesis. The second one is that it is rather far from the biological reality and clearly aimed at problem solving. By these specificities, it is close to our work presented in chapter 3. It is, as the L-systems are, based on a rewriting rule grammar that defines the topology of a neural network. The grammar is represented as a tree that encodes the cell lineage. The system proved to be efficient at solving a variety of simple problems, but it is not biologically defensible. For instance this is not a cellular model[4] in the sense that the resulting topology is encoded in the grammar, not in the cell metabolism. Surprisingly, this work, not concerned with form, is close to the very morphogenetic work of Sims [177]. In that model, the development of a body was encoded as a recursive graph (a folded tree), each node being a body part, each arc a joint.

[4]In this ontogenetic section, I use the term cellular model in a restricted sense compared to the definition of p.6.

There are many more grammar-based works and other abstract models but to finish this paragraph, I would like to briefly talk about "*Tile Automaton*", a system developed by Yamamoto and Kaneko [232]. This work is worth mentioning on several grounds. Firstly, it is one of the very few model of basic ontogeny, at the limit between elementary organic growth and chemical self-organization. Secondly, it is a good example of what is covered by ontogeny at large, going beyond morphogenesis. Finally, it is close, in its spirit, to many aspects of our own work. The Tile automaton is an abstract model of chemical reaction of molecules scattered over a 2D Lattice. Each molecule reacts to strictly deterministic rules, in discrete time-steps. It is thus very close to a CA structure. What is remarkable is that given certain initial environmental conditions, structures develop with specialization of certain parts to produce the energy needed by the rest of the structure to maintain itself. There are many other interesting works of cellular ontogeny, such as the development model of Dellaert [40] and the Cell Programming Language by Agarwal [4].

Realistic models

One of the first examples of realistic models kind of studies dates back to 1952, when Turing, in his paper *The chemical basis of Morphogenesis*, [213], presented a mathematical model of cell interactions based on reaction-diffusion systems that could exhibit stable properties. This model was quite simple, focusing on one mechanism of development, cell interactions. This early work, however, gave rise to many successful models of pattern formation in shells [130], zebra coats [15], and butterfly wings [136].

More recently, in 1994, Kitano [97, 98] proposed a system that seems rather abstract, topologically based on a 2D lattice, but that is quite realistic in its modeling of the chemical cell-cell interactions, the morphogen diffusion, and the active transportation of chemicals. Evolution was applied to the metabolism of the cell. This is the internal process that gives rise to the development of a structure. A successful process of morphogenesis (ontogenesis really in our understanding) was obtained, but the results remained preliminary. It is interesting to note that Kitano has an on-going project to develop a realistic simulation of the complete cell metabolism (yeast) and then the complete growth of an organism: *C-Elegans*. The first, modest step in this direction was to reconstruct the cell lineage of the worm from 2 to 7 cells[233].

The most complete realistic model to date was done by Kurt Fleischer [51, 52]. It is mostly based on differential equations, where a set of these constitutes the genome of a cell. The space is either 2D or 3D, and continuous. Basically, this equation set governs how the chemical concentration varies and how the external forces are applied to the cell and so modify its shape (its radius to be precise) and its movement. Internally to the cell, a simple conditional system models loosely the gene expression regulation. Besides change in the environment outside the cell, interactions between two cells and friction mechanical forces are simulated via an external set of partial equations. Thus, there is an internal metabolism and an outside world with fixed rules. However realistic this modeling may be, many processes remain discrete. For instance, when a specific behavior function crosses the zero threshold, a cell division occurs and a new cell is created, instantly. Most of the encoding and tuning of the parameters were done by hand and they gave rise to some interesting

phenomena of development: from simple behavior, such as cells climbing a gradient to more complex behaviors, such as cell differentiation and neurite growth. As the preceding work, all development results only from the confrontation of the internal metabolism of the cell with an external environment — there is no constructing program. The advantages of this work are quite evident: it is the most biologically realistic model to date, it is a general model, and it presented results close to reality. Nevertheless, the disadvantages are also numerous: it is computationally very intensive, hard to tune (the attempts to do the tuning by means of evolutionary computation failed), and many biological aspects, like the gene expression regulation network, are still poorly approximated. Finally, to conclude this section, it is nice to see that Fleisher's work which was mainly interested in basic research problems found an application as a texture mapper for complex 3D shape. The general idea for this task was that each cell could grow only on the surface, and then developed according to the cell around it and some physical law (like gravity). The result was an automatic covering of the abstract bodies, for instance, mimicking rather well animal fur.

There are many other works of interest in theoretical biology, among which we can cite the work of Savchenko *et al* [169] or Unemi [215] where developmental cellular system are studied and more particularly in the last one, the evolution of cellular system.

2.4 The Evolutionary Paradigm

If development is adaptation on the scale of life, evolution is adaptation on the scale of history. As we saw, in the 1950's people like Turing and von Neumann got interested in modeling and understanding biological phenomena in terms of information processes. The beginning of the computer era pushed forward that simulation trend and it was then natural that models of artificial evolution were developed. The first work going in this sense date back to the 1950's and was done by Fraser [59]. Fraser's work is really the modeling of biological phenomena by means of a digital computer. A year later, Friedberg [60] conceived an evolutionary model in the modern sense. He explicitly wrote that it was for *automatic programming*. These two works are usually considered as the foundation of evolutionary computing. By the middle of the 1960's three of the four main strands of evolutionary computation were defined, *Evolutionary Programming* (EP), by Lawrence Fogel [55], *Genetic Algorithm* (GA), by Holland [84], and *Evolution Strategies* (ES), by Bienert, Rechenberg and Schwefel [159]. The field, however, matured into a unified domain of research of itself only in the late eighties, early nineties, with the acceptation of the term evolutionary computation as covering the whole field, and the first EC conferences.

In the next subsection, I will present the first important form chronologically[5], *Genetic Algorithms* (GAs). A general overview of the principles will be presented along with the most common problems encountered using evolutionary algorithms in general. This subsection will also present most of the vocabulary of this field, which is usually borrowed from biology.

In the subsequent subsection, I will present *Genetic Programming (GP)*. The material evolved in GAs is a string, not a full-blown program. GP proposes to evolve tree-structured

[5]This is arguable, and some would rightly say that evolution strategies were first. However, Genetic Algorithms were the first to be formally presented and intensively explored.

programs, a form much closer to real programming. However, this variation is not problem-free. More specifically, I will present one of the main problems of GP, program bloat.

Finally, after this presentation of GA and GP which are the techniques used in this thesis, we will take a quick look at the two remaining strands of evolutionary computation, Evolutionary Programming (EP) and Evolutionary Strategies (ES).

2.4.1 Genetic algorithms

After having cleared up some of the jargon, I will present the pseudo-algorithm of a Genetic Algorithm. Quite naturally, the best way to understand its workings is to grasp the global architecture of the system. I will then expand upon some fundamental points: population generation, fitness evaluation, parent selection, reproduction, mutation, and replacement. I will conclude by examining the question of GA efficiency by way of a discussion of the schema theorem.

Jargon

Inspired by biology, the specialized vocabulary of EC descends directly from natural evolution. However, sometimes the terms have changed in meaning, and, moreover, they always have a precise sense, specific to computer science. The terms will remain vague here as their exact meaning will become clear throughout this section. First, the strings subjected to the Genetic Algorithm (and more generally any structure subjected to any evolutionary algorithm) is *evolved*. What is evolved (for GAs, strings) is called a *genome* or a *genotype* or sometimes a *chromosome*. This is quite understandable if we are to think of these structures as the genetic material. Each genome is an *individual* in *a population*, a pool of genomes. This notion of individual can be taken as the string itself, or the behavior implied by the execution of the string. Hence, it is both the genotype and the phenotype. The *fitness* of an individual is the equivalent in natural evolution of its capacity to survive. A higher fitness improves the probability of being selected for reproduction. *Reproduction* of individuals is the altered copy of their genome, either randomly, and this is *mutation*, or altered as a function of another selected individual, and this is a *crossover*. The *children* or *offspring* of an individual are the resulting genomes from these alterations. If all the population is replaced by the offspring, then the new population is called the next *generation*.

The algorithm

The best way to explain a Genetic Algorithm is to have a look at an archetypal pseudo-algorithm of such programs.

As one may see in Figure 2.5, the general principles are quite simple. The idea is to first probe randomly the search space, `GENERATE_POPULATION`, and then apply recombination, `REPRODUCE`, to good individuals, `SELECT_PARENTS`, to lead to better individuals. However the algorithm is more subtle than simple hill-climbing because a Genetic Algorithm is supposed to poll new points in the search space and so avoid local maxima. This is done most notably through mutation, `MUTATE`, but also through recombination, `REPRODUCE`. Actually, the aim of most of the variations around this archetypal structure is to find for the

```
P = GENERATE_POPULATION();
current_generation = 0;
WHILE (current_generation < MAX_NUM_GENERATION) DO
   FOR_EACH Individual i in P DO
      fitness_i = EVAL_FITNESS(i);
      IF (fitness_i >= ACCEPTABLE_FITNESS)
         result = i;
         HALT;
      ENDIF
   ENDFOR
   fitnesses = {fitness_1,...,fitness_n};
   PP = SELECT_PARENTS(P,fitnesses);
   C = REPRODUCE(PP);
   MUTATE(C);
   P = REPLACE(P,C);
   current_generation = current_generation +1;
ENDWHILE
```

Figure 2.5 Pseudo-algorithm of a generational Genetic Algorithm.

given problem a good balance between exploration and improvement. This question is often called the exploration vs exploitation problem. We will now explicate all the functions of the pseudo-algorithm, thereby showing the most common types of GAs.

`GENERATE_POPULATION`

The first thing to do in a GA is to generate a population of string candidates of solutions. This is usually done completely randomly according to the problem to solve. For instance, in chapter 4, we seek a one-dimensional, nearest neighbor CA rule which can be characterized as an 8 bit string. Hence, the population is just generated by randomly picking numbers between 0 and 255. However, in some cases it may be interesting to do this not quite randomly, in order to bias the initial generation of the population. This can be done either by strictly restricting the search space, e.g., in the example above only pick odd numbers, or by introducing a preference for a part of the search space, e.g., still in the example above, double the probability of picking an even number, or by a combination of both techniques. This question has been seldom explored in the past, but it brings out the representation issue. This is central to any evolutionary algorithm and could be summed up as how to encode the problem to solve into the structure evolved, in the case of genetic algorithms, into a string. There are two sides to this question, one is how to encode the problem so that the crossover operator(s) will be meaningful and the other is how to map the string to a solution — how to map genotype to phenotype. We will explore the first part of the question in the REPRODUCE paragraph, while treating the second part in the next paragraph.

`EVAL_FITNESS`

Fitness evaluation is *the* central part of Genetic Algorithms. It aims at favoring, i.e., giving a high score to, individuals with a good genome, that is with promising genes, that could, when crossed with other good individuals, give better individuals. But, fitness is very specific to the problem considered. Genetic Algorithm (more often Genetic Programming that we will see in section 2.4.2) have been qualified as an automatic process of discovery or as an automatic programming system. We would tend to moderate these claims by qualifying GA's as *semi*-automatic. The success of GA techniques often depends on its design and more specifically on the fitness function design. In effect, the fitness function judges the validity and the potential of the candidates. In some sense, this is where the discernment, the acumen of such algorithms lies. Though the solutions are dependent on the problem sought, a certain number of recurring issues occur when designing a fitness function. For instance, starting from a random population, the fitness should be able to distinguish between hopeless individuals and poor individuals with a potential. This problem is usually addressed in two ways, **through selection**, by amplifying the fitness differences as we will see in the `SELECT_PARENTS` paragraph, or **by using several/evolving fitness functions**, where the fitness evaluation criteria gets harder as the average fitness gets better. A second problem encountered is avoiding "false solutions". It is often hard to include all possible solutions to the problem in the good-fitness set and to preclude from this set all bad solution. And Genetic Algorithms are very good at optimizing the fitness with candidates not solving the problem we are tackling. This is perhaps Murphy's Law of evolutionary computation in general. Hence, one should take care that high fitness candidates should **only** be good candidates. It would be impossible here to review all the problems linked to the design of a fitness evaluation function, as this is where the intelligence of the system lies. However, in the coming chapters concerning the evolution of cellular machines (3 and 5), we will address the question.

`SELECT_PARENTS`

Talking about fitness without saying anything about the parent selection procedure would be meaningless. It would be a bit like giving a measure without the scale. Contrary to the fitness function, the selection procedures are well-defined. Of course, it is not impossible to define new selection operators, but the four detailed below are, by far, the most commonly used.

- **Fitness proportionate selection** This selection procedure was introduced by Holland (see [85], ch.5 and 10.2), as a way to optimize the trade-off between exploration and exploitation. Its workings are quite simple. First, normalize all fitnesses to the range [0,1], and then use the normalized fitnesses as the probability to select individuals. As pointed out by Tettamanzi and Tomassini [206], this selection process, though

backed by decision theory[6], is not without drawbacks. For instance, it does not deal well with low fitness differences, or high-fitness unique individual. In the latter case, it leads to quick convergence, which of course, may result in premature convergence.

- **Rank selection** Fitness proportionate selection is directly based on the absolute value of fitness. However, as we have seen earlier, fitness definition can be quite arduous and its absolute value less meaningful than its relative value. Rank (or linear-ranked) selection establishes a probability of selecting an individual as a function of its rank. The formula to calculate this probability is usually:

$$p_i = \frac{1}{||P||}[\beta - 2(\beta - 1)\frac{i-1}{||P|| - 1}],\ 1 \geq \beta \geq 2. \tag{2.1}$$

  This selection procedure, first developed by Grefenstette [73] and Whitley [219], is modulated by the *selection bias* term of the equation, β. One may see that if $\beta = 1$, then there is no selection pressure at all, while if $\beta = 2$, the worst individual has simply a null probability of being selected.

- **Tournament selection** As put by Sean Luke in [123], "tournament selection is popular because it is simple, fast and has well-understood statistical properties". The selection is done by successive tournaments among k individuals of the population each time. These individuals are chosen with uniform probability. Some have argued for no re-insertion while others have put forward the necessity of totally independent selection. The winner is either the fittest of the k (this is the *deterministic case*) or is chosen via fitness proportionate selection among the k individuals. The selection bias for this algorithm is, evidently, the number k. The higher the tournament size the more selective it is.

- **Truncation selection**

  Truncation selection is what has been used for most experiments with *Phuon* , the system presented in chapter 3. Its principle is very simple: select the μ best individuals and reproduce each parents $\frac{n}{k}$ times. It is sometimes called the (λ, μ) selection, where $\lambda = k\mu$, and λ is the size of the population. A variation is the $(\lambda + \mu)$ selection, where one parent is selected from the μ set of individual while the other is taken from the remaining of the population.

`REPRODUCE`

How does one combine the genetic material of the selected parents ? The answer lies in the reproduce operator, almost always called *crossover*. In the present case of Genetic Algorithms, as we are dealing with fixed-length strings, crossover is quite straightforward. One point is chosen in the strings, and the material beyond that point is exchanged (see Figure 2.6.a). A popular variant is to chose two points and exchange the material in-between the two points (see Figure 2.6.b). These are by far the most current crossover techniques

[6]It is built on Dan Frantz's work [58].

for GAs. However, this operator is often defined as a function of the representation of the problem to be meaningful. For a complete overview of recombination operators see [22].

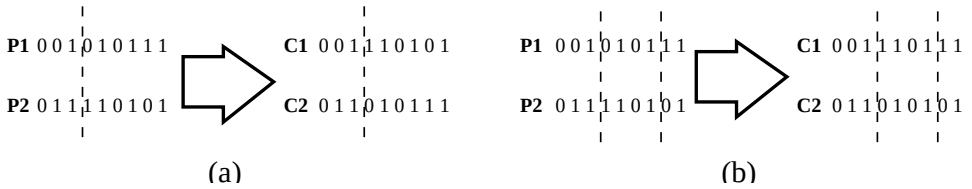


Figure 2.6 The two parents P1 and P2 give rise to the two children C1 and C2 via one-point crossover (a) or via two-point crossover (b).

`MUTATE`

Mutation is the other genetic operator. This would be the only operator for evolution in asexual reproduction. Actually in evolutionary programming, EP, (see Porto [222] for a complete overview) mutation is the *only* operator, with selection. In GAs, the aim of mutation is to introduce more or less random individuals into the population and thus explore more of the search space. The principle is straightforward. It usually consists in exchanging a symbol of the string chosen randomly by any other symbol from the available alphabet. Of course according to the representation of the problem this may vary. This is more than an exploration tool, however, as proven by EP, as it polls the search space nearby a good candidate.

`REPLACE`

The `REPLACE` function delineates how to place the children in the existing population. In **Generational GAs**, the whole population is usually replaced by the children of the selected parents. Hence the "good" parents from the previous generation are lost. This has the advantage of keeping diversity rather high and so avoiding premature convergence. However, if crossover is potentially destructive or very destructive, the best individual from the next generation may be very much worse than the best from the preceding one. To avoid this, we can copy directly to the next generation the best individual, or individuals. This **elitist** approach is often limited to the top 5%, to maintain the balance between exploration and exploitation.

However, there exists a totally different replacement algorithm. Actually, it implies a modification in the structure of the algorithm itself. The **steady-state** approach suppresses the idea of generation. At each loop, only new individuals are evaluated, then over the whole population the parents are selected to generate a number of children, c, where $c \ll n$, the size of the population. Usually c is about 1 or 2. To maintain the size of the population fixed, c individuals are selected for removal. In this algorithm parents thus compete with children, thereby removing the question of using elitism or not. A *generation*, in such an algorithm, is defined as n/c times the main loop, i.e, when n individuals have been replaced.

This approach, known as steady-state for GAs, is also known as $(\mu+1)$ for evolution strategies. Under that latter name, it was first presented by Rechenberg [160], and expanded by Holland in his landmark book [85]. The reader may refer to Fogel and Fogel [53] for a study of the virtues (or lack thereof) of this strand of GAs for optimization problems.

Some words on the schema theorem:

The schema theorem was presented by Holland in 1975, (in [85], theorem 6.2.3). It has often been presented as the reason genetic algorithms work. In fact, this crucial question remains unanswered for the most part. In this section, we will present the schema theorem and briefly recapitulate what we can say about the efficiency of GAs. The schema theorem is as follows:

Theorem 1 *In a genetic algorithm using fitness proportionate selection and single point crossover occurring with probability r, then for each schema $\mathcal{H}$:*

$$p(\mathcal{H})' \geq p(\mathcal{H})\frac{\overline{\omega}(\mathcal{H})}{\overline{\omega}}\left(1 - r\frac{L(\mathcal{H})}{L-1}\right)$$

where: $\mathcal{H}$ is a schema, L is the length of the chromosome, $L(\mathcal{H})$ is the length of the schema and is strictly less than L, $p(\mathcal{H}) = \Sigma_{x\in\mathcal{H}}p(x)$ is the frequency of the schema $\mathcal{H}$ in the population, $p(\mathcal{H})'$ is the frequency of the schema in the next generation population, and $\overline{\omega}(\mathcal{H}) = \Sigma_{x\in\mathcal{H}}\omega(x)p(x)/p(\mathcal{H})$ is the marginal fitness of the schema[7].

It basically says that the frequency of a schema in the next generation is directly proportional to its frequency in the previous generation, its relative fitness, and the non-destructiveness of the recombination operator. Some, like Radcliffe [158] even said that the schema theorem was a tautology. This is not quite true, as it can be deduced from it that if a schema is rare and has high relative fitness, it will multiply through the population at an exponential rate. Actually, it is interesting to note that Holland in his original publication does not promise anything more. So this theorem does not teach us much on why GAs are good search algorithms. Altenberg argued in his 1994 article (in [96], pp.47-74) that a way to measure the efficiency was to compare GA efficiency to random search efficiency. More precisely, for GAs to be better than random search there has to be a correlation between the fitness of parents and the upper tail of the distribution of their offspring, i.e., the offspring should not be normally distributed. Theorem 5 from Altenberg's paper [5], called by the author the missing schema theorem, basically proves that GAs are better than random search, according to the criteria above, depending on: the positive co-variance of the fittest schemata and the fittest offspring, and on the probability of the recombination operator to create more positive genotypes than it disrupts (positive in the sense that it alters positively the fitness distribution). This theorem is an answer to the question proposed by Altenberg, nevertheless, it is not immune to the tautology critique. Without delving into this discussion here, I can note, with Altenberg himself, that this correlation itself does not answer the

[7] The notation adopted here is that of Altenberg in [5]. It differs quite a lot from the original one by Holland. A *schema* is a regular string setting only some loci thereby matching many chromosomes, e.g., *101*1 matches 010101, 010111, 110101, and 110111.

question of GA efficiency in its general form. If we consider a system where the offspring always get the mean fitness value of its parents, Theorem 5 entails that the upper tail of the fitness distribution fares better than random search — there is a perfect correlation between parents and offspring, *but* there is no evolution. The fitness never gets better than the best fitness at generation one.

So the question of GA efficiency remains unanswered, and the numerous studies on the subject do not explain why GAs work, much beyond the intuitive arguments. It should be remarked that the No Free Lunch theorem[8] by Wolpert and Macready, [229, 230], tells us that it is just hopeless to try to prove that any search/optimization algorithm could outperform all the others on every class of problems. Nevertheless, as pointed out by many, this theorem is no longer true if one considers only a special class of problems, and that real, interesting problems were such a special class. For an interesting discussion on this matter, one may refer to Oliver Sharpe's work, [174].

The situation for Genetic Programming, that we are going to see in the coming section, is now the same. There was almost no theory at all for some time, but the recent attempts by Poli *et al* were successful. There is now an exact schema theorem for GP, [155],... which obviously does not say more than the GA schema theorem.

2.4.2 Genetic programming

Genetic Programming is essentially Genetic Algorithm on different structures: trees[9] instead of strings. In fact, it was introduced as such by Koza in 1989 [99]. This paternity of the field attributed to Koza is mainly due to the large body of work presented in his 1992 book [100]. As always in research, GP is actually the result of many preceding works such as [34, 56], but we owe to Koza a wealth of experiments that gave to GP its status as a separate strand from GAs. And though the global structure of the algorithm is identical to the one presented earlier in Figure 2.5, GP presents specific problems. Moreover its own crossover operator and its specific mapping from genotype to phenotype really "defines a conceptually different approach to [GAs]" as Kenneth Kinnear, [96], put it.

In this section we will outline the main differences between GP and GA. Technically, these lie principally in the `REPRODUCE` and `MUTATE` functions, and conceptually, in the fitness evaluation. We will also evoke some of the problems specific to Genetic Programming, and more particularly code-bloat.

Genetic Programming is an evolutionary technique where the genetic material is not a string, like in GAs, but an *executable program*. This change in structure involves major modifications in the fitness evaluation procedure. With GAs, one of the main questions was to define a proper encoding/decoding of the problem as a fixed-length string. Here, decoding is just executing the program. Hence, ideally, the fitness is just the ability of the program to solve a given problem, i.e., given the inputs, to produce the desired outputs. With the exception of a few, Nordin for instance [141], the program is not executed, and usually not

[8] To put it briefly the NFL theorem says that all algorithms that search for an extremum of a cost function perform exactly the same, when averaged over all possible cost functions.

[9] Though tree is the historical structure and the one mainly used, GP actually concerns any variable-size structures.

executable, directly on the system. The original idea remained conceptual and the structure evolved is just the representation of a program. There are many reasons to this choice but the three principal ones are: 1) An evolved program is something issued from a "random" process and may be either dangerous for the machine, or not executable at all; 2) The space of all possible programs, in any language, is vastly too large for the problem tackled, and so would imply a loss of both time and computational resources; 3) Finally, the crossover of real-program would be very complex compared to the usual crossover that we will see below (Figure 2.7).

So Genetic Programming evolves program representations, more precisely *trees*[9]. This representation presents many advantages, but the original reason was that it fitted well with LISP s-expressions. If we take lisp-atom or no-argument functions as leaves, and functions as nodes with arguments modeled as subtrees then the tree-representation is immediate. Usually, the leaves are known as *terminals* and the nodes as *non-terminals*. So there is a coding and decoding part in GP, just like for GAs, but it is limited to determining what the necessary terminals (inputs) are (both variable such as sensors, and constant such as integer numbers) and what the ideal non-terminals (functions) are for the problem to be solved.

Before jumping to the crossover question, there are some specific problems concerning the population generation for GP to be mentioned. The **tree generation** is just as random as the string generation is for GAs. However, as the structure is more complex, we usually distinguish three different ways to do this: *grow, full* and *ramped half and half*. The *grow* method is the "natural" way to generate trees: 1) pick at random, in the bags of terminals and non-terminals, a function; 2) for each argument of the function (if any), start again to generate the sub-tree argument. To limit the size of the trees generated, it is usual to fix a maximum depth, which when reached only allows picking terminals. The *full* method is exactly the same except that before reaching maximum depth only non-terminals may be chosen. This implies that the tree generated will be full. Finally, the *ramped half and half* consists simply in choosing at random either the full or the grow method for each tree generated. To go further, the reader may refer to Sean Luke's thesis ([123], ch.8), in which he studied of the consequences of the different tree-generation strategies, these three and others, for certain sets of terminals and non-terminals.

The most structure-dependent functions in evolutionary algorithms are obviously the genetic operators. Thus, **crossover** and **mutation** for genetic programming differs completely from the standard crossover of GAs (Figure 2.6). The *crossover* here proceeds as follows. First, a crossover point must be chosen in both parents. If we consider the tree as a graph, it will be one of the edges. This defines a subtree and an amputated parent tree for both parents. The two subtrees are then swapped. Figure 2.7 illustrates this mechanism. It is usual to predetermine a probability of picking a terminal as the subtree to be swapped. This is due to the fact that if the average arity of the non-terminals is a, then the ratio of non-terminals to terminals in a full tree tends toward $\frac{1}{a-1}$, thereby implying a high probability of picking a terminal, which in terms of evolution is not a good strategy as it assumes a correct global structure of the tree.

This crossover technique implies a fundamental property about the language chosen to be evolved. It has to be *syntactically closed*. If we are to find a numerical approximation

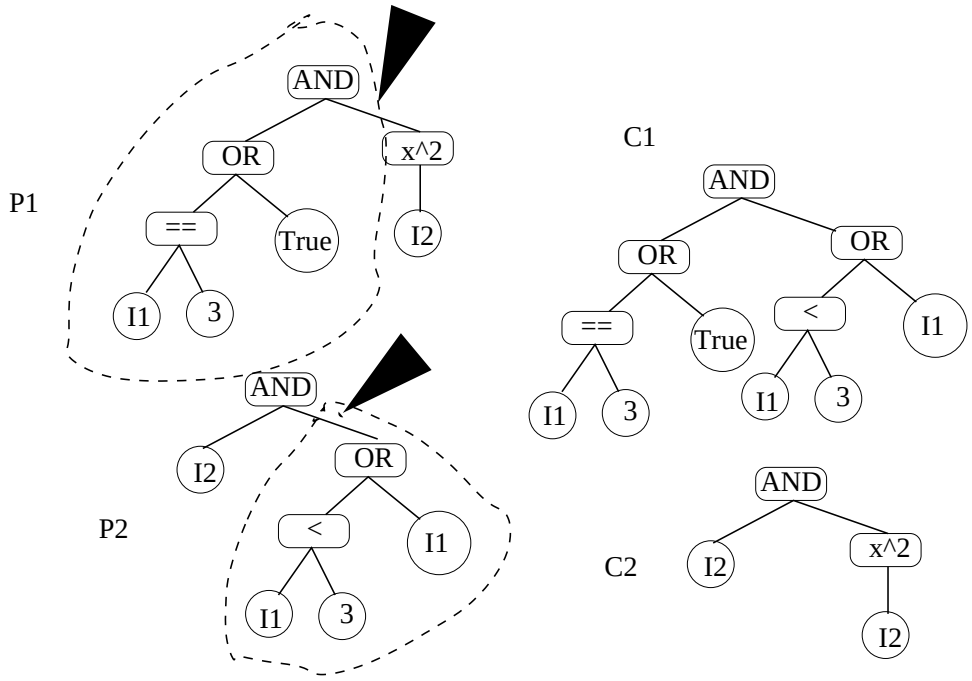


Figure 2.7 The two parents P1 and P2 give the two children C1 and C2. The dashed lines show the subtrees resulting from the crossover points (the black arrows) involved in the creation of C1 and C2. Here, we can see that if we take 0 for False and 1 for True, we have a syntactically closed language. Subtrees such as (OR (True,...)) which could be simplified as True are typical of unused genetic material (in the biology analogy, non-coding DNA).

of a function and we choose $\mathbb{R}$ as the set of terminals and $(+, *)$ as the function set, then it will naturally be closed as we are working in semigroups. Nevertheless, if we include division we will already be confronted with the problem of division by zero. The solution usually adopted in this case is to define a meta-function, DIV, which returns a predefined result when given zero as argument. However, if we extend the language to more complex instructions such as Boolean operators and conditional branching, the problem of syntactic closure appears in its full-blown form. There are two main strategies to solve this. The first is to render the language artificially syntactically closed. For instance all functions may return an integer and take an integer as argument. The advantage of this solution is that it is easy, simple, and leaves complete liberty to the genetic programming algorithm, the drawback being non-nonsensical crossover. The second strategy is to define syntactically closed subgroups of the language. For instance, Boolean or arithmetical subgroups, and then modify the crossover operator so that it picks two subtrees of the same subgroups. The advantage of this method is a more meaningful crossover. The drawback is, of course, the complication of the global algorithm but also, for certain researchers, the interventionist approach. This second method is usually called *strongly typed genetic programming* (See Montana [134]).

Mutation in GP consists in choosing a subtree from the selected individual randomly,

removing it, and replacing it by a new randomly generated subtree. The method of generation is usually the same as for the parent: grow, full, or ramped half and half. Of course, the depth of the subtrees is chosen in accordance with the maximum depth. Actually, it happens often that this maximum depth is chosen at random in a pre-specified range. Hence, even if the method is 'full', a mutated tree may result in a non-full tree if the depth of the new subtree is not in accordance with the depth of the original parent. Crossover is rarely constrained (except for special problems) to maintain the fullness of the offsprings, and thus the full approach only influences the original population. Both these genetic operators tend to create bigger and bigger trees. This problem is called *tree bloat* and will be treated below.

Automatically Defined Functions (ADFs) extension to GP

An interesting extension to GP, and the only one we are going to talk about, is Automatically Defined Functions (ADF). The goal of this extension is to create building blocks, which once evolved, are left untouched. It is somehow in the same spirit as strongly typed GP. They both intend to create more meaningful genetic material recombination, either by recombining blocks of the same type or by evolving functional blocks. ADF were first introduced by Koza [101]. The idea is to add procedure call to the language set of the problem, and evolve simultaneously but separately both the caller and the callee. To do so, two types of functions, ADFs and ARGs, are added to the language set of the problem. ADF_0,...,ADF_n are the names of the procedures, where n is decided at the conception of the language set. The ARGs are just here to specify how many arguments the ADFs take. Hence, when we apply crossover to two trees, the ADFs appears as monolithic, unbreakable blocks in the main tree, while at the same time it is possible to breed the two trees defining two ADFs. There is a double crossover taking place.

This method gave excellent results on a number of problems[10]. It was further developed in 1996 by Koza and Andre [102] to permit dynamic adaptation of the ADFs during evolution and thus to limit further the user pre-specification of the system. It is interesting to note that, though this is the most common, this is not the only method to automatically define procedure, automatically define constructing blocks. Rosca and Ballard [165] proposed in 1996 a set of heuristics that evaluates the usefulness of all the subtrees in the population and creates automatically procedures from the most useful ones. Lee Spector in 1996 [193] introduced a variation of ADF, called ADM, Automatically Defined Macro, which reevaluates the arguments each time; this is of particular import when this evaluation either changes with time or modifies the external environment. Finally, to conclude this paragraph on ADFs, we may evoke the claim of Angeline and Pollack [11], that GP automatically discovers effective modularization of the evolved programs and thus renders ADFs superfluous.

The bloat problem

There are many issues concerning evolutionary techniques in general, the main one being, as we saw when we exposed the schema theorem, why it works. In this paragraph, I would

[10] Actually, the Genetic Programming II book [101] by Koza is a catalog of problems where ADFs were used efficiently.

like to concentrate on an important and well-studied problem: **bloat**. This problem is not GP specific and affects any evolutionary algorithm working on variable-length structure. However, GP being, by far, the most renowned and popular algorithm of this type, and the one used in this thesis (see chapter 3), I will treat it in this context.

Bloat is the uncontrolled growth of an individual's genetic material. More exactly, it is uncontrolled growth with no gain in fitness — a prejudicial growth. It was first noted as a serious problem in 1980 by Smith [189] in the context of classifier systems. Concerning GP, Koza, in his 1992 book already mentioned bloat, but as the field developed, the issue became more pregnant and thus the subject of many studies [10, 20, 129], especially by William Langdon [107, 108]. These latter theoretical studies have shown that when the size of the trees reaches a critical size, growth tends to be quadratic, which very quickly stalls the evolutionary process.

Even if intuitively the occurrence of bloat, given the crossover techniques used, is not completely surprising, there is no clear theory about its causes. Sean Luke has quite rightly identified 4 main theories(see [123], ch.9), that I will expose now. The first one, chronologically, presented by Tackett [201], argues that the numerous useless subtrees, called introns in reference to non-coding DNA, that are stuck to useful subtrees will travel with them through the generations and propagate through the population, thereby accumulating in time. Tackett argued that the more selective pressure you have, the more code growth you get. Introns are also involved in a second theory. This relies on the fact that crossover is as constructive as destructive. Hence, the more ineffective code a good individual possesses, the less destructive crossover may be, and thus the less brittle the individual will be. It is interesting to remark that Tackett disagrees with this "risk-aversion" theory, usually termed as defense against crossover, which was independently defended by Blickle [20] and Nordin [142]. Removal bias, a third theory introduced by Soule and Foster [192] is basically based on the same argument as the second one, namely crossover defense. It focuses on special kind of introns, inviable code. Inviable code is code that cannot change an individual's fitness or function through crossovers. Finally, the fourth theory, diffusion, developed by Langdon [109], who also supports the three others, is *not* based on introns. He argues that there are more large fit trees than small ones in the whole space of possible trees. But, as we always start with very small trees (if we are to consider the whole space of possible trees), the size inevitably tends to augment as the search goes on, simply to reach equilibrium. None of these theories is improbable, but none neither reached a consensual acceptance.

Though the causes are unclear, solutions exist. There are several methods to limit bloat. We can divide these techniques into prevention and cure. As to the first, we have depth-limited tree generation, and its corollary, depth limited mutation. A more radical prevention is to change the structure of GP. Here, the introduction of the child or the parent depends on which is the best, the shorter being chosen in case of equal fitness. In-between prevention and cure, we have fitness pressure on size, where bigger trees are penalized. This last method attempts to withdraw the relative advantages of resisting destructive crossover. Finally, on the cure side, we have a radical method: forbid any trees with a size bigger than a predetermined depth-limit, which, unfortunately, tends to limit the space searched to the structure of the bigger trees existing when the limit is reached. Thereby, the effectiveness

of GP tends to nil. However, another dire method, code editing, has proved effective. The idea is to remove automatically the introns, and simplify the existing code. This method combats bloat efficiently, but should be used with precautions to maintain the balance between exploration and exploitation. Over-simplifying the tree inevitably leads to very poor search results.

As we see, there is no definite theory for the causes of code bloat and thus no cure. The heart of the problem may be that code bloat is an inherent part of artificial evolution and that limiting it, however useful from a practical standpoint, is just harmful to the global efficiency of the algorithm. Tentatively, one could pursue the analogy with natural evolution, and consider the great percentage of non-coding DNA as natural bloat. We will come back to the issue of code bloat when confronting the problem in chapter 3.

2.4.3 Other evolutionary techniques

To conclude this section on evolutionary computation, I will complete our tour by having a look at what is considered as the other two main strands of evolutionary computation. We detailed GA and GP that are used in this thesis. They allowed us to understand the fundamental principles of evolutionary computation in general. We now look briefly at the other two models, Evolutionary Programming (EP) and Evolution Strategies (ES).

Evolutionary Programming (EP)

EP was proposed by Fogel in 1962 [54]. He proposed that intelligence was the ability to make predictions and translate these into suitable responses to attain a certain goal. He deduced from this proposition that the classical heuristic approach to Artificial Intelligence was flawed, and remarked that evolution in nature is what produced this sort of adaptability, this sort of 'intelligence'. Fogel's first experiments, described in [55], were on finite state machines. He generated at random a population of finite state machines (FSM), and then repeatedly mutated the current population, evaluated the mutants, and then selected among both originals and mutants for the next 'generation'. This evaluation, the fitness measure, was the ability of the candidate FSMs to predict the desired output. This process was in perfect concordance with his first proposition. EP has been expanded to any sort of genome, its particularities being no sexual reproduction and selection *after* mutation. EP is used nowadays with success on numerous problems, notably neural-network training. Being based only on mutation, EP somewhat contradicts the usefulness of building blocks evoked in the ADFs section. Nevertheless, the current trend is to attribute to EP good capabilities on problems different than those on which GA or GP work well.

Evolution Strategies: (ES)

Evolution Strategies were first conceived in 1964 by Bienert, Rechenberg and Schwefel [159], but the archetype of ES that was later to be adopted was defined in 1973 [160]. This 1973 version of ES proceeds as follows: an individual a consists of two elements, $\vec{x} \in \mathbb{R}^n$, and $\vec{\sigma} \in \mathbb{R}^n$. $\vec{x}$ is mutated by adding a vector of random independent normally distributed values with mean 0 and standard deviation σ. If the new mutant vector $\vec{x}'$ is better, i.e.,

optimizes a certain pre-defined function $f : \mathbb{R}^n \to \mathbb{R}$, then it replaces the old one $\vec{x}$. As such the system was not a novelty. As Schwefel [172] pointed out, there were already forerunners in the late 50's. However, Rechenberg introduced the idea of mutating $\vec{\sigma}$ by increasing it, if over a certain number of steps, the mutations of $\vec{x}$ were successful more than $1/5$ of the times, decreasing it otherwise. Hence, each individual carries its own standard deviation, its own genetic operator in some genre. Later on, the selection-replace scheme was extended to $(\mu + \lambda)$, where μ parents generate λ offspring, and then μ individuals are selected among the $\mu + \lambda$ candidates. The earlier scheme was thus (1+1) ES. The modern forms of ES include a more subtle set of parameters controlling the mutation, rather than just the deviation vector $\vec{\sigma}$ and usually, beyond mutation, a certain crossover between two parents. That crossover operator, when applied to vectors, is either done *discretely*, as for GAs, or, *intermediately*, in which case the values x_i, σ_i are linearly combined.

To understand things, you have to see them develop.
Aristotle, *Metaphysics, X.*

Chapter 3

Phuon: An Evolving, Ontogenetic System

3.1 Introduction

In this chapter I present an original cellular system developed for this thesis, named *Phuon*. The main motivation behind this project was to go beyond classical cellular systems, such as cellular automata. As we will see in the following chapters, CAs are powerful while remaining conceptually very simple. However they lack adaptability, and are highly prone to synchronization and general failure. The idea here was to add ontogeny to cellularity, growth and development being means of adaptation and thus robustness.

In the first section, I detail the motivations behind the conception of *Phuon*. I also overview the general principles of the model and relate them to some inspiring works about cellular systems exhibiting some similarities.

In section 3.3, I detail the implementation of *Phuon*. This is not a technical description as such, though many technical details will be given. This section is rather an explanation of the inner workings of *Phuon*, where I discuss the questions surrounding such a system, including synchronization and growth.

Section 3.4 finally presents the results obtained with *Phuon*. I will explain the two main tasks on which successful solvers were found, and demonstrate their robustness and adaptability qualities, or lack thereof.

These results are limited and consequently the concluding section, 3.5, will be a discussion of the problems encountered, the limits and the capabilities of *Phuon*.

3.2 Motivations

Cellular interactions are at the foundation of the complex phenomena of life. Kennedy and Eberhart even argued in their recent book, [95], that human intelligence and intelligence in general derives from "social" interactions, i.e., from the group. While not going this far, as I will show in the forthcoming chapters, even the simplest models of interactions, such as Cellular Automata, are capable of generating complex emergent behavior from simplicity.

However the capabilities of such simple models are limited and so is the global behavior to be expected. As we saw in chapter 2, many researchers have in recent years concentrated on more complex models, to study biological, sociological or computing problems, [13, 51, 90, 97, 203], to cite but a few.

The question when one sets to study emergent behavior in cellular systems should be, in most cases, what for? The general aim here is to get adaptiveness and robustness through cellularity. However, these properties do not come *per se* with cellularity. The absence of global control avoids a sensitive point whose failure is inevitably fatal to the system, and local interactions create good conditions for adaptation. On the other hand, as a counterpart, cellular systems require sophisticated interactions to gain self-organization and their very decentralized nature may render any subpart as important and as vital as the global controller of a centralized system. One means to create adaptiveness and provide reconstruction after failure is ontogenesis. One may wonder how growth can be a factor of adaptation? I use growth here in the sense of growth in nature: a process that depends on environmental factor and as such *is* adaptation[1]. As a historical aside, it is interesting to note that growth is somehow a characterization of life. When pre-Socratic philosophers had to chose a term for the concept of nature, as opposed to the $\tau\epsilon\chi\nu\epsilon$ (*Techne*), to what was man-made, they called it $\Phi\upsilon\sigma\epsilon\iota$ (*Phusei*), which derived from $\Phi\upsilon\epsilon\nu$ (*Phuein*), To grow. And however strange, this is still a rather good definition. What are self-replication, reproduction, self-reparation, if not growth, if not self-production. Crystals grow, one may even remark that technically it is a growth that changes according to the environmental conditions, but they miss the essential counterpart to growth, and that is death. In homage to ancient thought, the system presented below has been called $\Phi\upsilon\omega\nu$, (*Phuon*), the growing one.

Choosing between enough complexity to get interesting global emergent behavior and enough simplicity to still be able to talk about emergent behavior ([164] and references therein) is a complex task. The main model for this work was Cellular Automata. The aim here is to go beyond CAs, but they were the starting point and the base for reflection. Like a CA, this system is a spatially extended model discrete both in time and space. However there are two major differences, the first one in the dynamics and the second in the computation unit. Here the dynamics is asynchronous, and each cell is updated one after the other in order or randomly. They do so very briefly so as to mimic a sort of parallelism, but there is no perfect synchronization. For the second point, the cell we consider is more complex than a CA cell. Besides, I divided the system into two layers, one active (the cellular layer) and one passive (the environmental layer). This is only a particular point of view of the system, and it may have been seen as a single layered system (though the mapping is not obvious). This secondary view point has been adopted for CAs many times, for instance by Reggia[161]. Though CAs were the major starting point they were not the only source of inspiration. Most of the works[2] presented in the ontogenetic section of chapter 2 influenced this work. Here I would like to point out the works of Taylor [202, 203] which are quite similar in the topology and the dynamics, keeping the system simple, in the sense of discrete

[1]It is interesting to note that, in a different context, Lee Spector [194] proposed an evolving system capable of adaptation at run time that he called ontogenetic GP.

[2]Basically the work before 1998.

time and space. Nevertheless contrary to our system, this work concentrates on the study of the system in itself as a model of the early chemistry of life and has no other aim than itself. The work of Furusawa and Kaneko [62] on developing cellular systems aiming at cell differentiation was of interest not for the design of my system but rather as a good overview of the possible modeling of cell growth, and the problems linked with it. This model again, as all cellular developmental models to date, is only studied in itself. Finally the work of morphogenesis via developmental cellular systems of Hugo de Garis [68] is very close to *Phuon* in many respects: both time and space are discrete, and this model is based very closely on CAs. However there the cell possesses only a *very* basic internal program. I should point out that this is one of the only works that I know of where the aim was not only the study of the system in itself. In *Phuon*, the aim is not to evolve the system for itself, but rather to evolve cells that through their development will solve problems in an adaptive and robust manner. So actually what we are seeking is a developmental system for problem solving.

The problem with CAs and their likes is the lack of a mathematical framework or any other systematic means to map a desired global, emergent behavior onto the necessary local behavior. In *Phuon*, as the system is much more complex, this task becomes perfectly impossible systematically. One way to avoid this impossible design is to search automatically the whole space mapping every genotype to every phenotype. Unfortunately exhaustive search is out of the question given the size of the search space. This is why I make recourse to evolutionary strategies to find the proper internal cell program that would develop into solving the desired problem. So in this project one has to distinguish two phases of research: the *evolutionary phase* and the *simulation phase*. The former is there to find a good solution to a problem, the latter is to exploit this solution, i.e, to see the emergent global behavior. In the detailed description below I shall first begin by the developmental system, the core and soul of *Phuon*. This phase is the only way to map a genotype to a phenotype, thus it is a necessary preliminary to understanding the evolutionary phase that will then be presented.

3.3 A Detailed Description of the System

In this section, the *Phuon* system will be detailed. I will present every important choice of implementation. This will also be the occasion to discuss the choices made. This presentation will be divided in two subsections: *the developmental system* and *the evolutionary engine*. The developmental system is the core of the project. As stated earlier my aim is to evolve the internal program of the cell so that it develops into a consistent organism, potentially able to solve a problem. Hence, this part is really what determines the "physical rule" of the system, physical in the most general sense, that is, including the biology of the world. It determines what a cell can and cannot do, the environment, what are the interactions, etc. The second subsection is concerned with evolution. It depicts the parameters chosen for the evolution process, and more generally raises the problematics related to the artificial evolution of the system.

3.3.1 The *Phuon* developmental system

Phuon can be viewed as a two-layer cellular system. A passive environmental layer, and an active cellular layer. I will first present the former briefly and then describe the latter at length. In the second part, I will first explain the cell globally, then its language and its growth, to finally conclude with the question of synchronization.

The environmental layer

The environment is a simple 2D square grid of size n^2 (In the example of subsection 3.4.1, n is 100). Each square is denoted by its coordinate (x,y) and a state s, where s a positive integer. On every square of this layer at most one cell of the cellular layer may lie. It is a passive layer but not a static one: the cellular layer can alter the state. Topologically, it is bounded and finite (thus, not toroidal). The environment is thus defined as the mapping $f : \{(x, y)|x, y \in (0, .., n)\} \rightarrow \mathbb{N}$. The original mapping is the data input to the problem.

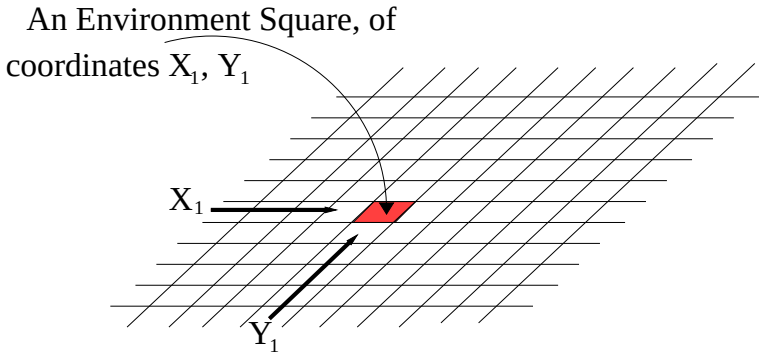


Figure 3.1 A view of the environment layer in *Phuon*.

The Cellular Layer

A population of cells develops over the environmental layer. This population of cells is called the cellular layer. As every cell is exactly over one environmental square and every environmental square is below at most one cell, it is conceptually easier to see the cellular layer as composed of void and cells, thus making it isomorphic topologically to the environmental layer.

The cellular layer is the active layer of the system. One may say the "living layer", that will through its activity solve a given problem. The environment, at least at the start of the experiments, encodes the data that depends on the given instance of the problem. A germ cell (or a few germ cells depending on the problem) is placed on the environment and will through its functioning develop, interact with its daughters and the environment so as to finally solve the given problem. The results presented in section 3.4 take this cellular layer to be the solution to the given instantiation of the problem. However, it is perfectly imaginable that the modified environment at the end of the run will be the result for other kinds of problem.

To understand its workings, let's first describe the cell and the then its dynamics.

What is a cell ?

The cell is the fundamental unit of the system. It is the only active element and it is through its functioning and multiplication that the system will eventually solve the given problem. As exposed earlier our goals were cellularity and ontogeny. The first of these terms implies that there is no global control, that globality may emerge only through local interactions. Hence our cell only interacts with its direct environment. The second term, ontogeny, implies some developmental mechanism. This is done both explicitly and implicitly. Explicitly the cell can duplicate itself. Implicitly, just as biological cells specialize and abandon their totipotent status, here our artificial cell may specialize irreversibly in time.

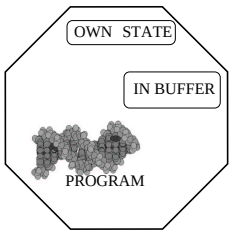


Figure 3.2 An overview of the cell organization

Interactions

The general architecture of the cell is characterized by three objects: a state, an in-buffer and a program (Figure 3.2). The program controls all the activity of the cell and will be detailed below. As for the 'state' and the 'in-buffer' they are the results of our choice for interactions. Many questions arise when considering interactions, but they may be summed up as two main ones: What information may a cell exchange ? And is the exchange passive or active ? In Cellular Automata, the information exchanged is the state of the cell, and the communication is passive, that is, a cell reads another cell's state, but does not exchange information as such. Here, the 'state' is the only information available to the neighbor cells. However, this is different than CAs, as it is set by the cell 'voluntarily'. So it may be a functioning state as for CAs, but it is not necessarily so. The second kind of information that cells may exchange in *Phuon* is through the 'in-buffer'. Any cell may write to the in-buffer of any of its neighbors. Then, the data may be read by the receiving cell. So its a doubly active process. It is noteworthy to say that the in-buffer is only readable by the owner cell. Technically the information exchanged in both cases is a positive integer. To continue the CA comparison, the numbers exchanged here, through the 'state' or the 'in-buffer', are not bounded a priori, though obviously it is de facto bounded. So these exchanges may only occur in a local neighborhood. This neighborhood is defined as its four direct neighbors (a sort of von Neumann neighborhood). Finally the last kind of interaction is indirect and through the environment. Each cell may read or write its own environmental square. As we will see in the next paragraph the cell may move as a result of other cells' replication. So another cell may later find itself on that modified environmental square.

Ontogeny

As hinted above, the ontogenetic part of the cell is double, both implicit and explicit. As specialization may only occur as an emergent property of development, here I will only present the explicit ontogeny of the system, growth and death. The cell can duplicate itself. A growth by scissiparity, one could say, to pursue the biological analogy. This replication is done voluntarily and is the result of the execution of an instruction of the cell's internal program. The child cell is created in a chosen direction above an environmental square next to the one where the parent stands. There are eight possible directions. (see Figure 3.3.a). A problem appears when the desired reproduction site is already occupied. In real cell replication, the new cell makes its way by pushing up the existing cells. I dealt with this problem in the same way, though the result is a bit more extreme. Here we have a two-dimensional discrete space, so when a cell is created in an occupied space, that older cell is pushed forward in the same direction. If that place is also occupied, the process is repeated until the border of the layer is reached. As our environment is limited in size, it made no sense to make it toroidal. So a cell that happens to be pushed over the "edge" of the grid is destroyed (see figure 3.3.b and 3.3.c). If there is a way to grow, there must be a way to die. As pointed out in section 3.2, death is as much a part of the ontogenetic process as growth. This instruction is equivalent to suicide. This instruction is all the more necessary as in a first version of *Phuon* the program was read forever in an infinite loop. Thus voluntary death was the only way for the number of cells to decrease. Secondly, and principally, if we desire ontogeny to be a means of computation it seems necessary that dying should happen voluntarily at any time during the execution of the program, as part of computation and not as a fatality.

These actions have been encoded into a language that is described in the next paragraph.

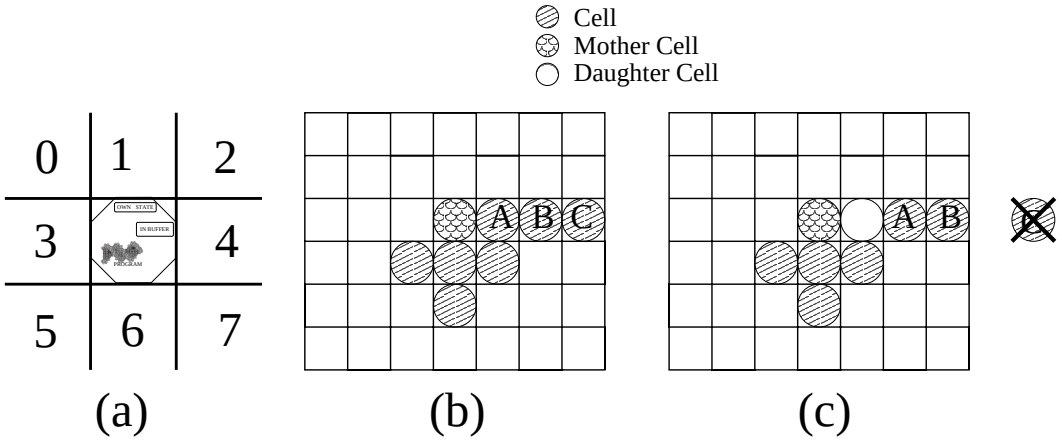


Figure 3.3 The Cell reproduction: In figure (a) we can see the 8 possible reproduction sites. In figure (b) we see the cellular layer before the mother cell (the scaly cell) creates a daughter cell in direction 4, and in figure (c) we see the results with the older cell C being pushed out of the grid and thus sacrificed.

The program

The program of a cell is a list of simple instructions, encoded as a basic register machine assembler. Table 3.1 presents all the instructions, with their arguments and actions. It encodes the behavior of the cell. A certain number of instructions are there for interactions with the environment layer and the neighbor cells, in the form of Reads and Writes. A special instruction `SPLIT` allows for cell duplication. Its counterpart `DEATH` kills the cell. Besides these instructions which have been discussed above, there is one more 'action' instruction and `NOP`. The operation `NOP` is necessary for synchronization between cells, to create some sort of "waiting for results" instruction. Beyond action instructions, there are a number of control instructions. They appear in the assembler language as the branching instructions `BZ, DBNZ`. However to understand what these really mean, I have to say that this assembler language is designed and used only to facilitate asynchrony. Our cellular system is partially or totally asynchronous. I am going to come back to this question in the paragraph on the system dynamics. However, to simulate this asynchrony, we need some sort of multitasking to switch between cells program execution. To do this, *Phuon* uses this pseudo-assembler language. In effect, this language is totally linear. Hence if the program counter and the registers are saved, any computation may be stopped anywhere and restored easily. Conceptually, however I designed the language at a higher level. At that level, it may be represented as a tree and this is the representation I will use quite naturally for the evolutionary engine. This representation is exposed in section 3.3.2. So conceptually, rather than branch, the control in the language comprises conditional expression, of course, If...then and if..then...else, and also a Repeat loop. These control instructions necessitate Boolean operators such as comparison and logical operations, which are thus also part of the language. Addition and multiplication operators were naturally added to allow some sort of simple computation. This language allows for complex and sophisticated behavior, the downside being a rather large search space. Evolved trees, as pointed in the background chapter, may be meaningless. Though I have developed specific genetic operators (see section 3.3.2 for details), it is still possible to have trees of the form "IF (false) Then...". It is useful to keep such genetic material for the sake of diversity, but when compiling into the pseudo-assembler I simplify the trees as much as possible without changing the semantics.

Instruction code	Arity	Arg. type	Comment
DEATH	$\emptyset$		The cell commits suicide.
NOP	1	CST	The cell does nothing for n cycles, where n is specified in the constant arg.
WRITEENV	1	REG	The environment square on which the cell is takes on the value in the register specified.
WRITEO	1	REG	The state of the cell takes on the value in the register specified.
WRITEx	1	REG	where x is one of E,W,S,N. Writes the value in the register specified to its East,West, south,North neighbor respectively.

Continues »

» Continued			
Instruction code	Arity	Arg. type	Comment
SPLIT	2	REG	Create a new cell in the position specified in the first register whose initial state is the value in the second register.
READENV	1	REG	Puts the value of the environment square on which the cell is in the register specified.
READO	1	REG	Puts the value of its own state in the register specified.
READB	1	REG	Puts the value of the in-buffer in the register specified.
READx	1	REG	where x is one of E,W,S,N. Puts the value of the respectively East, West, South and North neighbor in the specified register. Puts 0 if there is no neighbor in that direction.
CONST	2	R/C	Puts the value of the constant in the register specified.
SUM,MULT	2	REG	Sums/Multiplies the values of the two registers, and puts the result in the first register.
GT, LT, EQ	2	REG	Returns True in the first register if the value in the first register is greater than (less than , equal to, respectively) than the value in the second register. Returns False otherwise.
AND,OR,XOR	2	REG	Returns in the first register the results of the logical 'and' (or, xor, resp.) between the value the two registers specified.
NOT	1	REG	Replace the value of the register by its logical negative.
BZ	2	R/C	Branch to the relative address specified by the constant if the register is zero.
DBNZ	2	R/C	Branch to the relative address specified by the constant if the register is NOT zero.
B	1	CST	Branch unconditionally to the relative address specified by the constant.
END	$\emptyset$		Depending on the version, either the program terminates and the cell dies, or it resets the PC and the registers.

Table 3.1 The register machine instructions. For the argument types, REG denotes a register, CST a constant and R/C both types. True is coded as 1 and False as 0.

The dynamics of Phuon

In the real world, true parallelism is the standard at every level. Galaxies work in parallel, planets work in parallel, organisms work in parallel, cells work in parallel, atoms work in parallel. Each entity has an inner movement or follows external forces, but none requires some global external processing power to keep going. Sequentiality is a characteristics of computer simulation. When one sets to simulate parallelism on a conventional computer, it is impossible to create an exact image of parallelism. So the question is what synchrony or asynchrony to apply when evaluating each entity. Traditionally CAs for instance are

totally synchronous. This hardly reflects any real systems though it does provide a background prone to mathematical analysis[3]. As the aim of this system is to gain robustness, an asynchronous model was the obvious choice.

Two models of asynchrony were implemented in *Phuon*, a regular model and fully asynchronous model. I define a regular asynchronous model to be a deterministic model. Technically, in this first model, every cell present at the beginning of a time step is run for a fixed number of instruction cycles in order of their creation. A time step is the update of every cell once. The cells created at the present time step are updated for the first time at the next step. Thus these cells may remain inactive for a long time if there are already many cells, while they are physically present in the cellular layer at creation. Thus this deterministic model allows for repetitive experiment, but does not reflect real parallelism. The second model is fully asynchronous. All cells are updated individually, but the order is totally random and the number of instructions executed at each update is also random within a small range. This model is thus much closer to full parallelism, however it is *not* really parallel as no two cells may be updated at the same time step. As we will see in the results section, this model, while more satisfying for the mind, seems to impair basic cell communication.

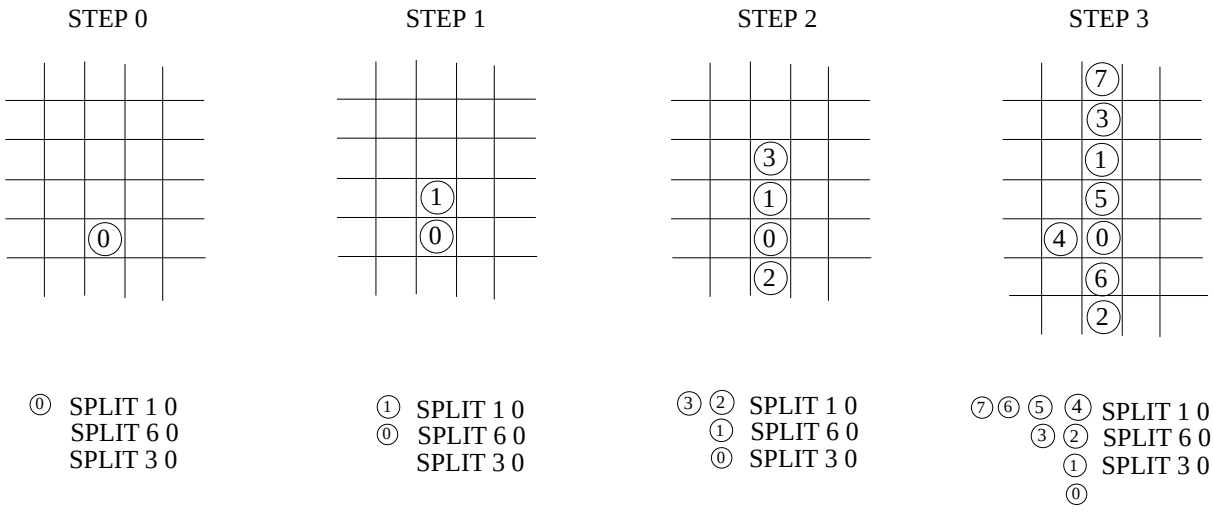


Figure 3.4 An illustration of the deterministic synchronization between the cells. The number on the cells is their id, and represents their order of creation. They are placed next to the program instruction they are about to execute. The number of instruction per update here is 1.

As said earlier the program is expressed as a tree for the evolutionary algorithm (See section 3.3.2), and is compiled into a simple assembler program for the simulation phase. Each update of the cell is thus a precise number of instruction evaluations in the deterministic case or a random value within a range for the non-deterministic case. For both the (n_i/s_t) number of instructions per step (a step being the evaluation of all the cells created before the step begun in that model) is an important parameter. The smaller the n_i/s_t the more

[3]In chapter 6, I study asynchronous CAs.

aware of the environment the cell is, the larger n_i/s_t the more sophisticated the program may be. For instance if we perform 1000 steps per evaluation, then n_i/s_t implies at most 1000 instructions executed which in the compiled assembler remains a rather basic program. So this parameter is a sort of measure of what percentage is supposed to be computed via interactions and what percentage is left to a sophisticated complex and poorly interactive cell. I tested values n_i/s_t of between 1 and 1000.

One can see in figure 3.4 an example of the execution of a simple program with the regular asynchrony mode.

3.3.2 The evolutionary engine

As exposed earlier, there are two motivations in using evolutionary computation techniques. The first one is practical. It is always very complex, and most often mathematically intractable to devise algorithms for *complex systems*. The second one is that if the aim here is to develop problem solvers, the questions related to the study of self-organization are also of interest. Evolutionary techniques often propose unexpected means of self-organization. Given the choice that a cell would be controlled by an internal program, Genetic programming (GP) was the natural evolutionary paradigm to adopt.

But using an evolutionary strategy does not come for free, it introduces many parameters to tune. The search space here being huge and the control sought being complex, the evolutionary tuning is actually central to our work. The basic framework of the algorithm remains unchanged through our experiments and is based on the GP paradigm presented in chapter 2, but three functions which are of crucial importance for a successful search were modified for *Phuon* : Tree_Generation, Tree_mating, and Tree mutations.

I am now going to describe the general framework and then each of these three specific functions. But first let's describe what is the concept of a *world* in *Phuon* and then the language used for the algorithm.

What is a world ?

What do we evolve ? Obviously we evolve the program of the cell, but it is conceptually better to see the population as a population of *worlds*. To pursue the biology analogy, evolution takes place on the genotype, but what is "selected" is the individual behavior in an environment, the phenotype. A world (see Figure 3.5) here is a program, in both its forms (evolvable and compiled), an environment, and a developed cellular layer. It includes all these elements as we may calculate our fitness based on any of these, for instance, we may calculate a penalty based on the length of the compiled program, a bonus for a particular final configuration of the environment and of the cellular layer. This precision is important in that the mapping from genotype to phenotype is rather complex here. If we mate cell programs, we are not mating cells (as in the cellular algorithm presented in chapter 5). Each cell multiplies and behaves in a different world.

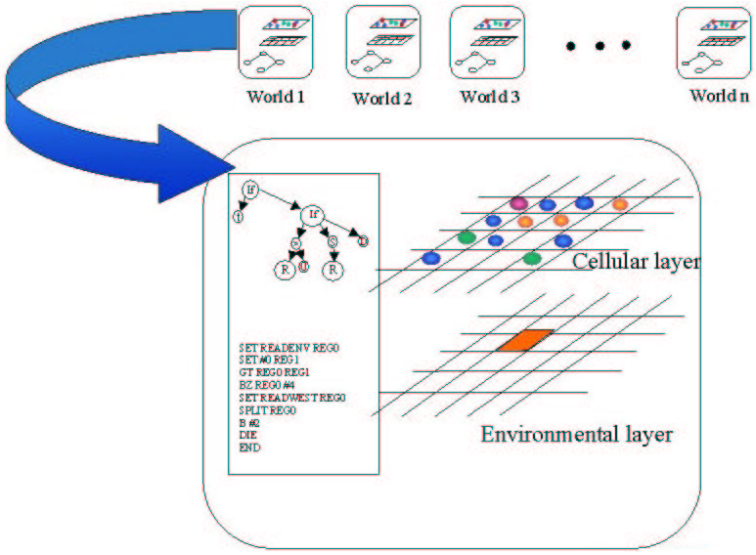


Figure 3.5 A population of worlds

General framework

The evolutionary paradigm used is classic Genetic Programming. A number of programs as tree structures are generated. They are then compiled and run, each of them in their own world, following the simulation phase described in section 3.3.1. The fitness is mostly calculated during this simulation phase. I used for almost all experiments a truncation selection where we select the top half of the trees which are mated to produce 95% of the population. A high elitism rate is also employed as the top 5% are copied from one generation to the next. This selection combined with elitism lead to a rather high convergence so the mutation rate is also high comparatively to what is found in the literature to be established at 5%.

The general settings where established after a core of initial experiments and were seldom changed afterward. Elitism is undeniably essential to *Phuon*. Each and every experiment run without elitism failed to find good individuals. On the selection mode, I also tested ranked selection with a medium and high pressure ($\beta = 1.5$ and $\beta = 1.8$) which gave no significant difference with truncation selection which was finally adopted. These results however should be mitigated by the fact that they are the results of about twenty experiments on the food foraging task.

The cellular language

As said earlier the assembler language is the result of the compilation of the tree structured language used for the evolutionary algorithm. Actually the language was designed in the from presented here. This design has been influenced both by the simulation phase and by the evolutionary phase. The former is of course the most important, influencing the essence of the language, the inner semantic. The latter influenced the syntax. The language

is divided into three types of instruction, which will be useful for the genetic operators as we will see below. These three types are: Statement, Arithmetic and Boolean. The global structure of the language is a specially shaped tree. Each Arithmetic or Boolean function is a standard GP tree, however, the Statement function take its arguments as defined in table 3.2, plus the next Statement. Hence it is more a list of trees rather than a tree (see Figure 3.6). Here is now a description of the high-level language.

Statement type: This is a kind of generic type that groups many different functions which all have two characteristics in common: – they are actions and – they do not return any value. They are described in table 3.2(a) This is the top level type, and the trees to be evolved are a list of 'statements'. A typical tree may be seen in Figure 3.6. The instructions have their exact counterpart in the assembly language, so their meaning is evident from table 3.1.

Statement Type Instr.			Arithmetic Type Instr.			Boolean Type Instr.		
Instr.	Arity	Type	Instr.	Arity	Type	Instr.	Arity	Type
Die	0	-	Sum	2	A1,A2	>	2	A1,A2
Nop	1	A	Mult	2	A1,A2	<	2	A1,A2
IfThen	2	B,S	#n	0	-	==	2	A1,A2
IfThenElse	3	B,S1,S2	ReadEnv	0	-	^^	2	B1,B2
Repeat	2	A, S	ReadO	0	-	||	2	B1,B2
WriteEnv	1	A	ReadB	0	-	&&	2	B1,B2
WriteO	1	A	Read$x$	0	-	NOT	1	B
Write$x$	1	A				True	0	-
Split	2	A1,A2				False	0	-

(a) (b) (c)

Table 3.2 The cellular language instructions. The 'Type' columns denote the type of the arguments, where A is for arithmetic, B for boolean and S for statement. It is importnat to note that the first argument is always evaluated first. The meaning of each function is quite intuitive. The reader will refer to table 3.1 and the text above for an explanation. Write$x$ and Read$x$ are actually for instructions with x being any of E,W,S,N.

Arithmetic type: All operations of this type return an unsigned integer. The terminals are either a constant (#n) or the Read functions which allows the cell to see its environment and the neighboring cells. The constant is chosen at random at tree generation between 0 and a predefined constant[4]. The functions are the two classical arithmetic operations, sum and multiplication. (Modified Division and Subtraction, so as to remain in $\mathbb{N}$, were introduced and then removed when it appeared that their effect was rather detrimental or nil). The instructions are listed in table 3.2(b).

Boolean type: The Boolean type operations return a Boolean, True or False. The mapping to the assembler language is almost immediate, knowing that || means OR and ^^ means XOR. The instructions are listed in table 3.2(c).

[4]In practice this constant is often very low, between 10 and 100.

Tree generation

At the start of a GA there is classically a random generation of possible trees. The positive aspect of such a procedure is that no a priori direction is given toward a possible solution. The negative is obviously the counterpart, we do not partake of the knowledge (or intuition) we may have about the problem to be solved. The idea, thus, in implementing a special Tree_generation procedure is to introduce some a priori knowledge.

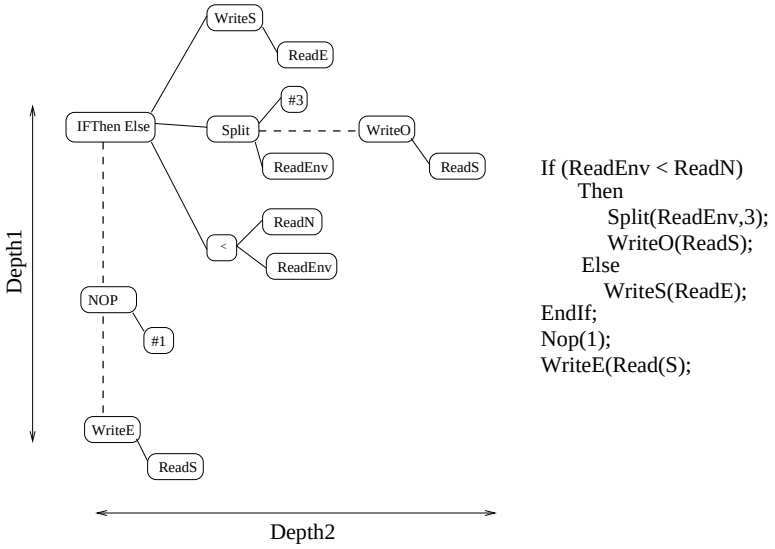


Figure 3.6 The general structure of the language as a list of trees. The arguments are linked to the node with plain lines while the next statement is linked with dotted lines. Depth1 represents the number of top-level statements. Depth2 represents the depth of the subtrees which is calculated taking the next statement as any other argument. For instance here, Depth2 is 3: `IfThenElse-Split-WriteO-ReadS`, while Depth1 is 3: `IfThenElse-Nop-WriteE-ReadS`.

There are two procedures for generating trees in *Phuon*. The first one is the classic random one with the 'grow' method. The trees are then generated with the only constraint to have a size within a predetermined range. Given the structure of our language, there is actually one defining the limits on the size of the list of trees, and another how deep may each subtree be (Depth1 and Depth2 respectively in Figure 3.6). These limiting factors do not prevent bloat as we will see[5]. The second procedure to generate trees follows these constraints. However, instead of picking the nodes at random with uniform probability, a bias is introduced in function of the task. For instance, one may classify the instructions according to some interesting characteristic, e.g., communication (Reads and Writes), size regulation (Split and Death) and control (Ifs and Repeat). The rest of the operation is left aside. Then we could estimate a task as highly demanding in communication but not in growth factor and generate the trees with on average 10% growth instructions, 60%

[5]This confirms Langdon's recent works which state that the initial tree size does not influence bloat,[108].

communication, and 20% control instructions. I should point out that the experiments I ran, though not statistically significant, tend to show that except with extreme percentages (10 and 90% for instance), the general behavior of the evolutionary runs did not show noticeable differences.

Genetic operators: Mutation and Crossover

Classically in GP, Mutation and Crossover are done totally randomly. To do so, it is customary to make the language subject to evolution semantically closed. Here as False is coded as 0 and True as anything not 0, then making any statement return a dummy positive integer would have done the trick. Nevertheless, this would lead to rather "dumb" programs with meaningless expressions like `If (Split) Then {Not True}`. Thus I modified Mutation and Crossover in order to make meaningful operations. By meaningful, I mean both making sensible crossover, but also deriving programs out of the evolutionary runs that are understandable by humans. For this, as we saw in table 3.2, I divided the language into the three types 'Statements', 'Arithmetic' and 'Boolean'. Then, for crossover, a node is chosen totally randomly in the first parent, but is chosen at random in the second parent among the nodes of the same types. For mutation, the subtree starting at the node chosen randomly is replaced by a new subtree generated randomly but of the same type. Therefore an `If` statement, for instance, will always have a Boolean as a first argument, and two statement subtrees as its other two arguments. Of course, one may argue that the division could have been finer, and that replacing a `Split` with a `Write0` does not make much more sense. However, one should remember that this "knowledge" inserted into the genetic operations should be limited so as not to prevent unexpected solutions to emerge. This is even truer when there is no a priori knowledge of what the solution may look like. We always fall back on the eternal question of Exploration Vs Exploitation. Thus, I voluntarily maintained that partition of the language to what seemed the minimal acceptable and desirable level. Leaving still as much liberty as possible to the GP algorithm.

3.4 Results

In this section I present the two main tasks solved with *Phuon*: Food foraging and Controlled growth. As one may see, these results are rather limited compared to the ambitions. However, they provided the opportunity to observe some nice properties and limitations of the system, specifically as it concerns adaptiveness and fault-tolerance. I will discuss, in section 3.5, the reasons for these limited results after having presented very briefly in this section the problem of program bloat in *Phuon*.

3.4.1 Food foraging

This task was the first one on which the system was tested, and the task on which it was the most successful. As we will see, not only several solutions were found but they exhibited the desired properties of working in full asynchrony and being relatively fault tolerant.

The task

The aim of the task is starting from an undifferentiated zone to explore the environment and find a pile of food. This pile is modeled as a limited zone on the environmental layer where the values are different from zero. As we may see in figure 3.7, I modeled it as a large square with different values. At the start, the cell starts at a random place and should only develop largely on the pile of food.

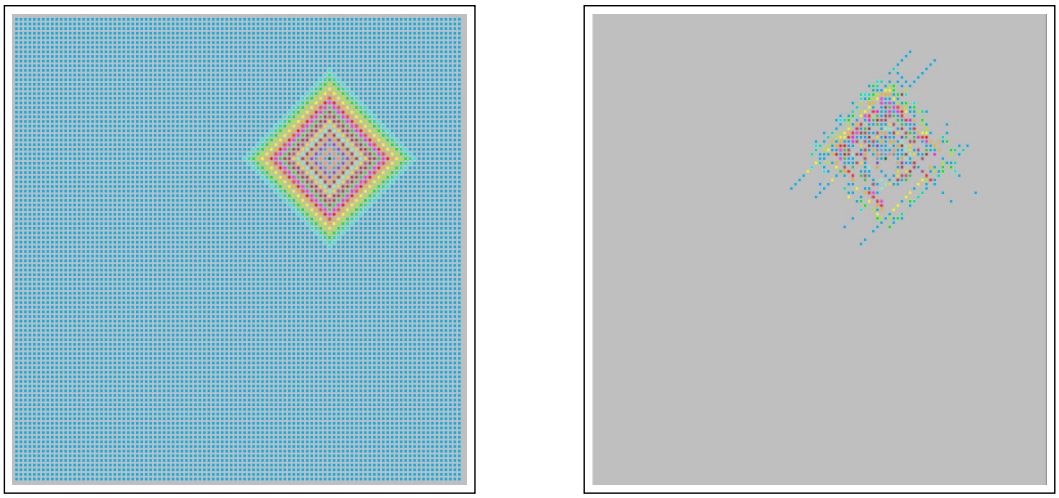


Figure 3.7 In (a), we can see the environmental layer used for the evolution of the Food Foraging solvers and in (b) the cellular layer after a 100 time steps with $n_i/s_t = 1$. Light blue color in (a) reflects the state 0, i.e., no food.

The main result

For the evolutionary runs, we used the environmental layer of figure 3.7.a. The fitness evaluation was done as follows: A germ cell was placed at random on the grid, and then run for a random number of steps between 100 and 150. The pile of food is also placed at random each time. Here 'step' is to be understood in the sense defined in *The dynamics of Phuon* paragraph, p. 40. This is done 10 times. The fitness is summed over the ten times and was +30 for a cell being at the center of the pile and then it was decreasing by the sum of x-distance and y-distance to the center of the pile. Thus it was still +10 for a cell at the border of the pile but rapidly decreasing to large negative values. The world where no cell was created was given a fixed -1000000, and a world where cells had been created but no cells were left was given -500000. The n_i/s_t parameter, the number of instructions executed at each step for each cell was set to 50. The mode of Asynchrony used was the regular one. For the general data, the global size of the layers was 100x100, and the maximum number of cells was 5000. The population size in terms of GP, the number of worlds in other words, was set at 500.

These evolutionary runs were very successful. A good fitness was reached in about 80%

of the experiments[6], and this with either of the two selection procedures: truncation and ranked selection. The result of the development of one of the best individuals found is shown in figure 3.7.b. Its program may be found in the annex A, in section A.1, at the end of this thesis.

The working principles of the program are hard to grasp. By using GP and a natural language, for computer people, one of the aims was to gain understandability in the end results of the evolutionary runs. In this respect, the use of GP has not proven successful. However, I can say that basically it compares the state of its neighbors with the environment square on which it stands to decide to split or not and in which directions. Besides at one point it takes as state the value of its environment square, while never altering that environment. Actually if one looks carefully at the program, one may realize that it is only good at covering completely the pile of food while not extending too far from it. Quite logically it optimizes the fitness, but it is not really foraging. Nevertheless, this should be mitigated, as it does a rather good job at foraging for other piles if it can start on a pile of food, as one may see in Figure 3.8.

Regular Asynchrony Vs Fully Asynchronous mode

The specific program presented in more detail here was obtained through evolutionary runs using the Regular Asynchrony mode. Other results faring as well in fitness and reasonably well visually were also obtained when running the evolution with the Fully Asynchronous mode. The interesting fact to note here is that the program studied here works as well in the fully asynchronous mode. This shows that hoping to gain general synchronization failure robustness using cellularity is not hopeless, if not guaranteed of course. This also underlines that the communication between cells is not sophisticated. However, as appears clearly when looking at the code, it is not nonexistent by far.

On the importance of the n_i/s_t parameter

As underlined in the general description of *Phuon*, the n_i/s_t parameter influences greatly the outcome of run. Using a more complex environment than the original one (Figure 3.8.a), we illustrate this change in behavior. In figure (b), we see the program evaluated for a total of 100 instructions and in (c) after 1000 instructions. One may see in (d) that the same program, still evaluated for 1000 instructions, but 4 instructions at a time, exhibits a different behavior as it discovers another pile of food. This is even clearer with 10 instructions per step, where for still 1000 instructions in total, it discovers 2 more food piles than for $n_i/s_t = 1$. Finally in (e) we can see how the cell behaves with the parameter as during the evolutionary run. This little illustration of the consequences of this parameter shows two interesting characteristics of the program found here. Firstly, it backs the idea of asynchrony robustness. The cells work in a different environment for each variation: the neighboring cells change all the more between each step as n_i/s_t gets larger. But secondly, it reveals the "strategy" for foraging in that case. As n_i/s_t gets larger, a lot more cells get created at each step. Hence a particular cell gets displaced a lot farther before being active, thereby increasing the chance to be on a more distant pile of food. This is particularly clear in figure (f).

[6]The runs being rather long at the time, the statistics are based on relatively few experiments, a little more than 50 in this case.

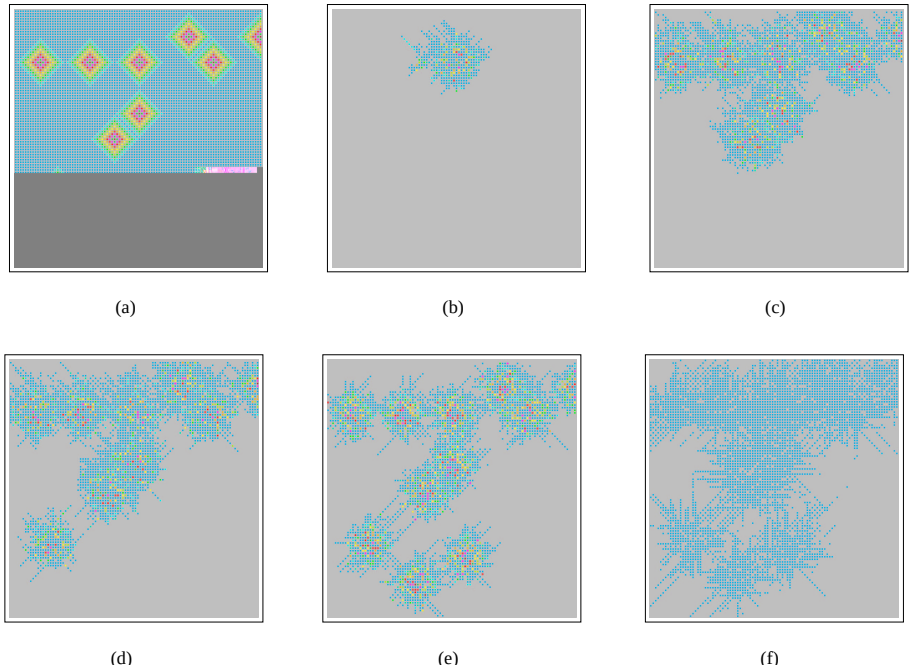


Figure 3.8 An example of the differences implied by the n_i/s_t parameter (in a secure environment). (a) The environment with several piles of food, at varying distance from one another. (b) $n_i/s_t = 1$, $s_t = 100$, (c) $n_i/s_t = 1$, $s_t = 1000$, (d) $n_i/s_t = 4$, $s_t = 250$, (e) $n_i/s_t = 10$, $s_t = 100$, (f)$n_i/s_t = 50$, $s_t = 20$.

Fault-Tolerance of the solution found

One of the motivations for using growth was to gain adaptability and up to a point this was reached. As we saw, though it was evolved on only one environment, the program found fitted many. The other motivation, in conjunction with the use of cellularity, was to gain a certain robustness to failure of the cell. To test this quality, a faulty environment was devised. This environment is defined as follows: For each instruction, a cell executes the `DEATH` instruction instead of the correct one with a probability p_f. Thus the cell is then destroyed and removed from the cellular layer. Hence if the number of instructions executed is n then the probability of failure is $1 - (1 - p_f)^n$.

The program of Annex A.1 was tested in this uncertain environment, and proved successful for reasonable values of p_f[7](below 8%). This confirms that the a priori expectations of the systems are not unfounded. It is interesting to note that the parameter n_i/s_t is not negligible in terms of failure robustness. Basically, it seems that it does not change significantly the value of p_f for which the program fails. For all the n_i/s_t tested: 1, 2, 3, 5, 10, 5, and 100, it always fell apart for $p_f = 8$. However, the same program copes better for higher values. If for $n_i/s_t = 1$, performance starts to degrade for $p_f = 3$, for $n_i/s_t = 50$, the performance is maintained for $p_f = 7$. But both fail for $p_f = 8$ and higher. We can remark that the number of cells when it stabilizes gets smaller and smaller as p_f increases, which

[7] p_f will be expressed as a percentage to make the reading easier.

seems rather intuitive, but this creates a certain instability meaning that structures like in Figure 3.9.d, h and i may at some points in time cover less piles of food. Nevertheless, they do always recover, which does not happen for $p_f = 8$.

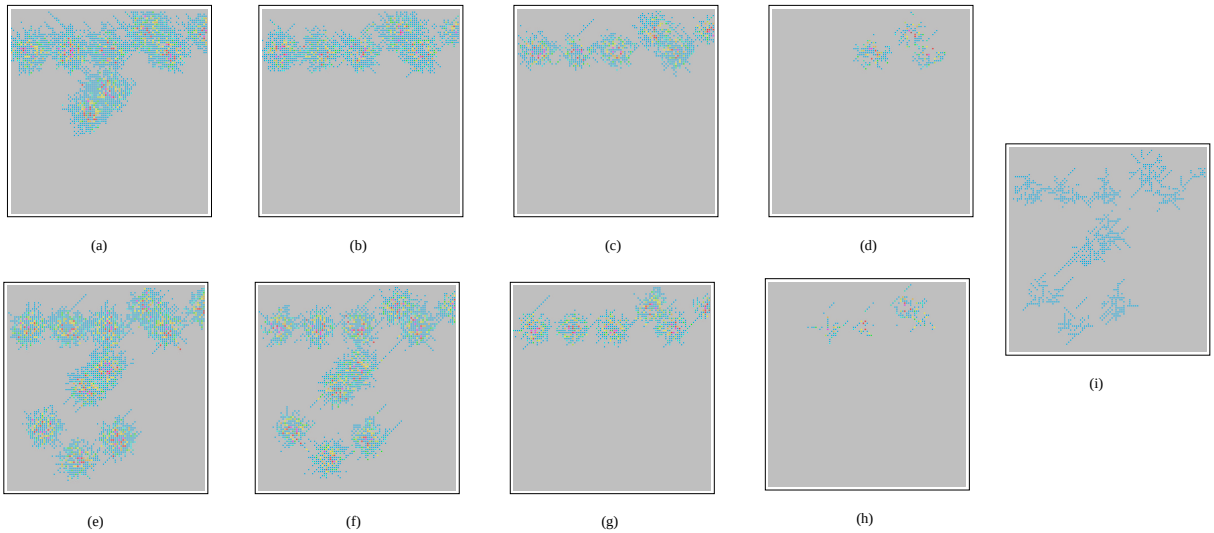


Figure 3.9 An example of the differences implied by n_i/s_t according to the value of p_f in a faulty environment. From (a) to (d) $n_i/s_t = 1$, and resp. $p_f = 2, 3, 5, 7$ and $s_t = 500, 600, 2500, 3500$. As one may see performance is not degraded in (a) but gets worse from there as p_f goes up. From (e) to (h) $n_i/s_t = 10$, and resp. $p_f = 2, 3, 5, 7$, and $s_t = 1000, 2000, 400, 400$. One can see that with a higher n_i/s_t, the same program maintains the same level of performance as in a non-faulty environment, for a higher p_f. In (i), $n_i/s_t = 50$, $p_f = 7$, and $s_t = 700$. The s_t given is the first one for which the structure shown was attained, and no larger structure was ever reached.

3.4.2 Controlled growth

Maintaining its own size at a stable limit is not a simple task in general for any growing system in which there is no energy limit. In the food foraging task, there was a clear limit predefined in the environment. Here, the task is to be able to grow first, but then stop growing through cellular communication. It thus requires much more active interaction than food foraging. Our tests actually showed that, on the one hand, it was a much tougher task for evolution and on the other hand the solution found was much more brittle.

The task

The aim of the task is to grow in a controlled manner. For that a unique seed cell is placed in the middle of a totally empty environment. All the squares are set to zero. After 100 time steps, an organism should have grown to any size that is non-negligible but far less than the maximum size allowed by the total number of cells and should be able to maintain an almost constant size.

The main result

The definition of fitness for such a task is not obvious. We are faced with one of those tasks were it is easy to explain the goal to a human but hard to do so to a computer. To be

precise, it is hard to define it in a way that is not limiting. There are many criteria like the shape or the size of the organism that are not defined at all. Any shape or size would do. The fitness is measured in three stages. First, at step 100, the number of existing cells is counted. If the number of cells is within 20% of the maximum number of cells allowed, the fitness is set to a very low score L. Otherwise, the simulation is run again for 20 steps. The fitness of the program is set to the original number of cells*10. Then it is decreased by 10 times the number of cells that are outside a 10% margin of the original count. If there are no more cells, the fitness is set lower than L. This second stage is performed again at step 200 to confirm. This fitness presents several advantages:— it is very general;— it privileges large structures over small ones and growth over full death; — finally it allows for non-fixed sized structures. This last property is necessary in *Phuon*. The very nature of the system almost forbids any fixed size structures. In effect, cells die when reaching the end of their program, so they must be continuously replaced, a bit like our own body cells, and at any one time it is thus impossible to guarantee an exact number of cells.

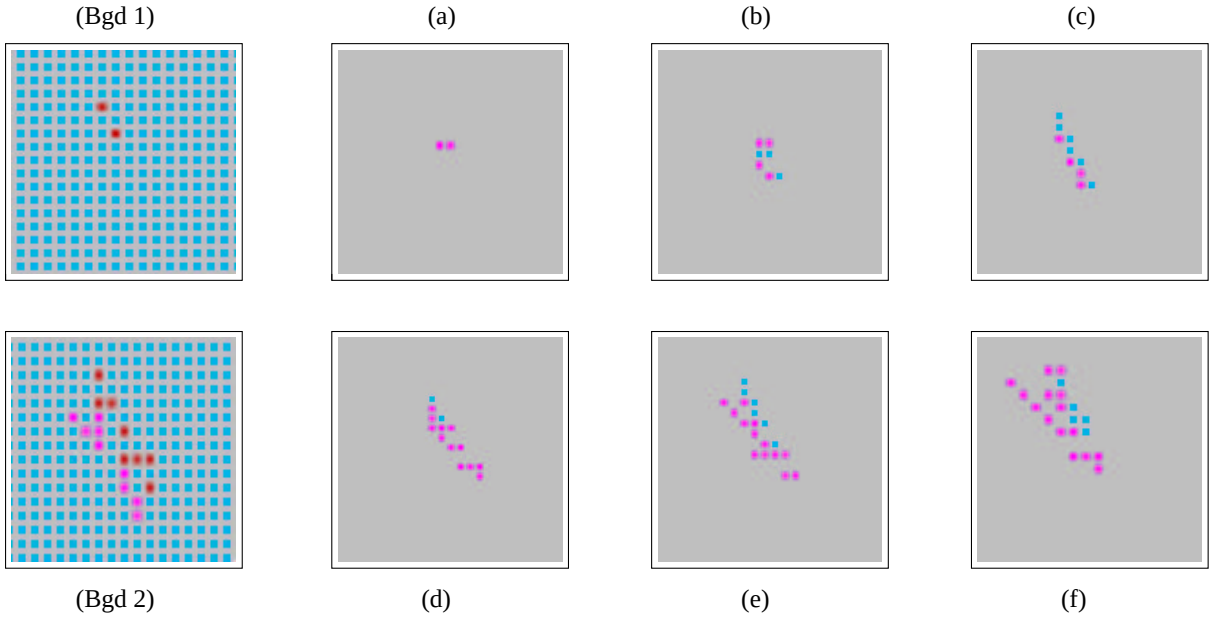


Figure 3.10 Development of an individual that is able, without any environmental constraints, to grow and then stabilize its size and position. The program of this individual may be found in section A.2. In (Bgd 1) and (Bgd 2), we see the environmental layer at about the time steps of figure (b) and (e) respectively. One can see the growth from (a) to (d), (d) being in the first step on the infinite cycle; and three characteristic pictures of its "adult" cycle (d) and (f).

The evolutionary runs on this tasks were not very successful. Actually, the fitness described above is the one with which the best 'organism' was found. This individual is presented in Figure 3.10, and its program may be found in Annex A, in section A.2. Many other fitness functions were tested, including limiting a special shape or privileging in a first stage of evolution poor individuals, but none gave rise to good results. Most of the "best"

programs found were actually cheating by really growing after step 200 (or any other limit of the different fitnesses). This tends to prove that the task is particularly hard for this system. I will discuss this point in section 3.5.

The program found is very large and complex. Even the simplified version[8] turns out to be totally incomprehensible. This shows that GP is not a guarantee of interpretable solutions. What is sure is that it uses the environment as a way to "communicate" between cells. It constructs a structure during the growth period, which it will then use to maintain a stable size. Actually the dynamics is a short growth followed by an infinite loop of cell production and death that maintains the organism stable.

Brittleness of the result

Contrary to the food foraging solver that was found, the program of controlled growth is very sensitive and very brittle. It was evolved with a regular asynchrony mode, using $n_i/s_t = 50$. If any of these criteria are changed it falls completely apart. This is due to the fact that, during the cycle, once grown, there is a precise timing between the top cell in blue that breeds in directions 6 and 7 and the bottom cells in pink that breed in direction 0. Thus, if a cell reproduces once more than what its timing was supposed to be, the cells will not meet for "synchronization" and the top cell will never stop to breed.

3.4.3 An aside: The bloat problem

During the evolution of these two tasks (and many others for which the results were not as conclusive as the one presented here), the problem of program bloat occurred acutely. Bloat is not specific to *Phuon* and is a problem common to many GP systems as we saw in Chapter 2. From the very first runs, after a few tens of generations, it was not possible anymore to compile the program in the pseudo-assembler in the space allocated (16Kb). The first remedy was not on the cause but rather on the effect by simplifying the program to remove all the unreachable code before compiling it. This turned out to be useful in terms of computational time (though marginally) but not against program bloat.

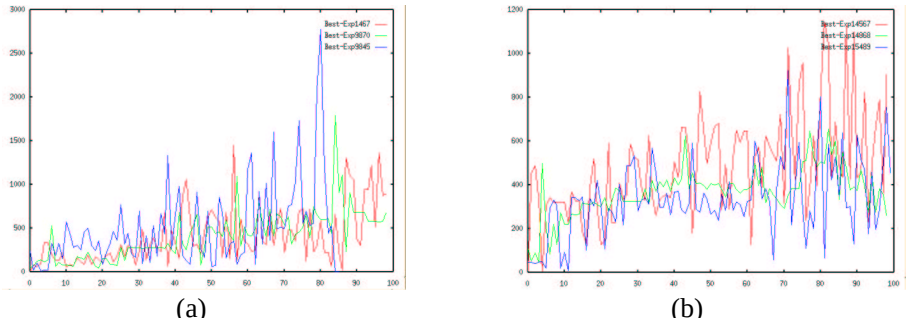


Figure 3.11 The problem of program Bloat in *Phuon*. The evolution of the tree size versus the number of steps for two evolutionary runs of the size control problem with the same parameter. In (a) without fitness penalty and tree-structure simplification, and in (b) with both these techniques.

[8] A version where all the non-reachable blocks, such as If (False), are removed.

There were two efficient solutions found against this problem. One specific to *Phuon* which consisted in simplifying the trees themselves when their sizes were too large. This implied a loss in genetic material but considerably reduced the tree size, which tends to show that most of this growth is due to *introns*. The other solution, which was combined with the first one, was a penalty on the fitness. This is the reason I mention this problem as it affects the results found by the system. To try to minimize the impact of this penalty on the solutions found, I devised a technique based on two fitnesses. A primary fitness based on the behavior of the program was designed as a multiple of 10, while the penalty, the secondary fitness, was bounded between 0 and -10. As truncation selection was used, this scheme allowed the selection of good individuals first and then, with the same performance, short ones. The lower limit of -10 for the secondary fitness was then increased when the bloat problem tended to be become more pronounced. Figure 3.11 illustrates the use of these two techniques combined.

3.5 Concluding Remarks

The results presented in this chapter, though interesting, are limited and do not fulfill one of our original motivations: problem solving. This partial disappointment should not be deceptive. *Phuon* produced what it could. The *a posteriori* discussion of what are really the potentials and the limits of such a system therefore comes naturally.

The food foraging example seems to demonstrate that such a developmental system can provide and even favor good qualities of adaptation: adaptation to a changing environment but also adaptation to faulty functioning. These qualities are attributable mainly to the continuous cycle of death and birth/growth. In effect, this cycle acts like biological cells whereby this cycle of death/replacement allows for most self-repair functions. However, as with natural cells, our system tends to suffer from cancer: cell division rapidly becomes uncontrolled. The poor results on the controlled growth task in absentia of environmental guides illustrates this natural disposition of *Phuon*. Interestingly enough if we look at the work of De Garis [68], his own system suffers from the same tares. I tried many experiments that aimed at morphogenesis, but for no shape, be it 'L', 'M', square, or circle did I manage to evolve successful cell programs. The reasons for this impossibility are difficult to identify for sure. In the introduction, I voluntarily left aside the question of the explosion of parameters in such a complex system as *Phuon*, but this turned out to be a real problem. As illustrated with the food foraging example every parameter influences greatly the end-result, independently of the program run. Moreover, one has to take into account the problems of the evolutionary algorithm which are not negligible by far. The relation between any of the parameters is intricate and most of the times impossible to work out. Nevertheless, through the experiments, we identified some problems inherent to the system. The major source of problems that seems to stand out is the difficulty of cell-cell communication. This appears clearly in the morphogenesis task which was only solved through communication, and also in other (unsuccessful) experiments I ran on problems like the convex hull, the Traveling Sales-

man Problem or more simply the Bridge[9]. This interaction problem is due to many factors, among which the two most relevant seem to be the necessity for active communication, and the passive move of cells. As noted earlier, (p.37), communication may be passive through the reading of the state of a neighboring cell — however this state does not necessarily represent the state of the computation in the cell and must be set actively by the cell read — but is often active, necessitating the reading of the In-Buffer, synchronized with a meaningful write from a neighbor in this buffer. And this brings us to the other problem: the passive move. Cells change place because of the replication of other cells. This, combined with the multitasking, implies that a cell may well execute one instruction, be swapped out, displaced to a totally new neighborhood, and then execute the next instruction. If that instruction was in reference to the neighborhood at the previous time step, then its execution becomes meaningless. Obviously, these two constraints impaired fatally the possibilities of *Phuon*.

This lack of results implied a major redesign of the system towards much more simplification. But then, it appeared quickly that a better understanding of simpler cellular systems should be gained first before moving forward. This is why all the forthcoming chapters concentrate on Cellular Automata, the most basic of such systems. Through this simpler model, yet still complex, I will come back to all the problems tackled in *phuon*. I will first explore the question of problem solving with CAs. Afterward, I will study the dynamics of the evolution of such systems, and finally I will extend the CA model to return, separately, to the questions of Asynchrony and Fault-Tolerance.

[9]Two points in the environment must be joined by a quasi-straight line.

Beauty is something wonderful and strange that the artist fashions out of the chaos[.]
W. Somerset Maugham, The Moon and Six pence, Ch. XIX.[191]

Chapter 4

Cellular Automata for Problem Solving

4.1 Introduction

What is a problem-solving cellular automata ? As any interesting question with intuitive straightforward answers, there is none definitive. In this chapter, through the study of mostly mathematical properties of both uniform and non-uniform CAs, I am going to provide some answers, at the price of limiting the scope of this question. More precisely, by concentrating mainly on the density classification task, I will ponder what computation by means of CAs really is, and argue that it is *'visual'* computation.

First, in section 4.2, I will define the three computational tasks used in this thesis to study CA behavior: Density classification, Synchronization, and Random Number Generation. This chapter will be almost entirely devoted to the theoretical study of the first of these, density classification, but will use the second one, synchronization, to illustrate a generalization of non-uniform CAs. The third task is used as an illustration of the diversity of what may be problem solving by means of CAs. Besides, it is one of the tasks for which the evolution-of-solution process is studied in chapter 5.

In section 4.3 I propose an *empirical* scheme to scale non-uniform CAs that were found by evolution. One of the main critiques made against non-uniform CAs, especially when found by evolution, is their non-scalability, i.e, the "impossibility" to adapt the solution found to any grid size. The scheme proposed, though empirical, dismisses this critique and generalizes any non-uniform CA. The discussion about the validity of the method will be an occasion to propose a definition of what is an emergent global behavior, in the scope of CAs.

The main part of this chapter is section 4.4, devoted entirely to the density task. I first extend to two-state, non-uniform, CAs the proof of the impossibility to solve the density task in its original form. I then prove that a particular elementary CA, CA 184 in Wolfram's notation, solves perfectly a modified version of the density problem. Thereby, I prove that the computational power of a specific cellular automata is in itself a non-evident problem. Finally, I prove two necessary conditions upon CAs for solving a generalized form of the

density task, and thus open the problem to higher dimensions. As an aside I prove the impossibility of solving a newly defined task, the grouping task.

Finally, we conclude in section 4.5 by discussing shortly the question of what is problem solving by means of cellular automata and its corollary: what is the computational power of cellular automata.

4.2 Computational Tasks for One-Dimensional Cellular Automata

In this section, I present three computational tasks accomplished by means of Cellular Automata. These are at the heart of this and the subsequent chapters, constituting a prime example of cellular automata computation. Actually, the first two are instances of *emergent computation*, while the third one uses the temporal dynamics of CA. As we saw in chapter 2, there are four main ways of considering computation in CAs. The first two tasks presented here are naturally of the second kind of the third type: they take a global input and produce a global output while computing locally. The third one, however, produces its result all along the run, locally.

The density task. The one-dimensional density task is to decide whether or not the initial configuration contains more than 50% 1s, relaxing to a fixed-point pattern of all 1s if the initial density of 1s exceeds 0.5, and all 0s otherwise (Figure 4.1a). Packard was the first to introduce this version problem [147]. As noted by Mitchell *et al.* [132], the density task comprises a non-trivial computation for a small radius CA ($r \ll N$, where N is the grid size). Density is a global property of a configuration whereas a small-radius CA relies solely on local interactions. Since the 1s can be distributed throughout the grid, propagation of information must occur over large distances (i.e., $O(N)$). The minimum amount of memory required for the task is $O(\log N)$ using a serial-scan algorithm, thus the computation involved corresponds to recognition of a non-regular language. Note that the density task cannot be perfectly solved by a uniform, two-state CA, as proven by Land and Belew [106]. This said nothing, however, about how well an imperfect CA might perform. One such CA, known as the GKL rule [66], can correctly classify approximately 82% out of a random sample of initial configurations, for a grid of size $N = 149$ [8]. Recently, researchers have focused on the use of artificial evolution techniques, demonstrating that high-performance (though imperfect) CAs can be evolved to solve this problem ([8, 132, 179, 185]). This impossibility however applies to the above statement of the problem, where the CA's final pattern (i.e, output) is specified as a fixed-point configuration. However, as I will prove in section 4.4.3, if we change the output specification, there exists a two-state, $r = 1$ uniform CA that can perfectly solve the density problem. Besides this problem may be extended to a more general version: A d-dimensional CA is said to classify density if it falls in one of two "effectively" distinguishable classes of configuration according to whether the density of the d-dimensional input configuration is lower than or above a threshold ρ.

The synchronization task. The one-dimensional synchronization task was introduced by Das *et al.* [39] and studied among others by Hordijk [87]. Sipper [180, 181] proposed using non-uniform CAs[1]. In this task the CA, given any initial configuration, must reach a final configuration, within M time steps, that oscillates between all 0s and all 1s on successive time steps (Figure 4.1b). As with the density task, synchronization also comprises a non-trivial computation for a small-radius CA. In the case of uniform CA, there is a perfect $r = 3$ CA, in the non-uniform case, many perfect $r = 1$ CAs do exist.

Obviously, in the non-uniform case there is an immediate solution consisting of a unique 'master' rule, alternating between '0' and '1', whatever the neighborhood, and all other rules being its 'slave' and alternating according to its right neighbor state only. However, Sipper used non-uniform CAs to find perfect synchronizing CAs only by means of evolution. It appeared that this "basic" solution was never found by evolution and, in fact, the "master" or "blind" rule 10101010, rule 170, was never part of the evolved solutions. This is simply due to the fact that this rule has to be unique for the solution to be perfect, which is contradictory to the natural tendency of the evolutionary algorithm used (See chapter 5, for an analysis of the tendencies of this algorithm). In this thesis, I only used evolved solutions to this task.

The random number generation task (RNG). Random numbers are needed in a variety of applications, yet finding good random number generators is a difficult task [148]. To generate a random sequence on a digital computer, one starts with a fixed-length seed, then iteratively applies some transformation to it, progressively extracting as long as possible a random sequence. Such numbers are usually referred to as *pseudo*-random, as distinguished from true random numbers resulting from some natural physical process. In the last decade cellular automata have been used to generate random numbers [89, 227].

Sipper and Tomassini [186, 187] applied the cellular programming algorithm to evolve random number generators. Essentially, the cell's fitness score for a single configuration (refer to Figure 5.2) is the entropy of the temporal bit sequence of that cell, with higher entropy implying better fitness (this should not be confused with the entropy measures defined in Section 5.2.2). This fitness measure was used to drive the evolutionary process, after which standard tests were applied to evaluate the quality of the evolved CAs. The results obtained suggest that good generators can indeed be evolved (Figure 4.1c); these exhibit behavior at least as good as that of previously described CAs, with notable advantages arising from the existence of a "tunable" algorithm for obtaining random number generators. Recently Tomassini, Sipper and Perrenoud [212] described a *two-dimensional*, non-uniform CA that produces very high quality random numbers.

[1] As a reminder, non-uniform CAs are CAs where each automaton may have a different rule.

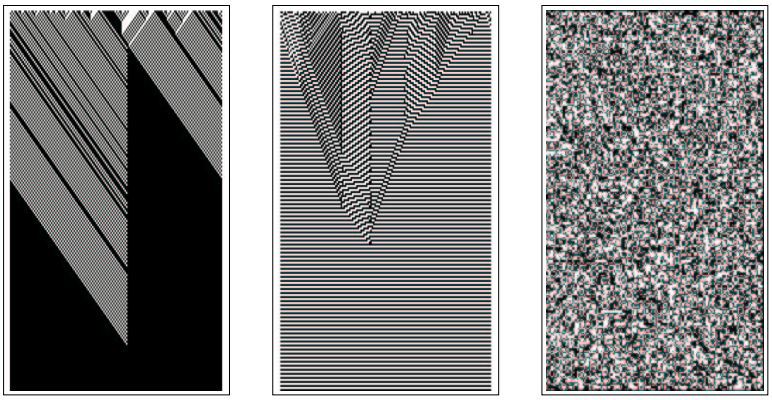


Figure 4.1 Demonstration of three evolved non-uniform CAs. Grid is one-dimensional, with radius $r = 1$ and size $n = 150$. White squares represent cells in state 0, black squares represent cells in state 1. The pattern of configurations is shown through time (which increases down the page). Initial configurations were generated at random. (a) The density task. Initial density of 1s is greater than 0.5 and the CA relaxes to a fixed pattern of all 1s, correctly classifying the initial configuration. (b) The synchronization task. Final pattern consists of an oscillation between a configuration of all 0s and a configuration of all 1s. (c) Random number generation. Essentially, each cell's sequence of states through time is a pseudo-random bit stream.

4.3 Scalability of Non-Uniform Cellular Automata

In this section we concentrate on *non-uniform, one-dimensional* CAs, our interest lying in the scalability issue. For uniform CA, scaling is not an issue as such as there is only a unique way to consider larger lattice sizes and that is to duplicate the existing rule. However, this form of *simple* scaling does not bring about *task* scaling; as demonstrated, e.g., by [35] for the density task, performance decreases as grid size increases. Actually, I conjecture, given the necessary conditions on the density task we demonstrate later in this chapter, that these CAs "cheat" to solve the task, more exactly they rely on a local density. Then, obviously, as the grid gets bigger, the chances of this guess to be accurate diminishes. In the case of non-uniform CAs the question is more complex. Besides this task scaling problem which remains, this simple scaling is not possible anymore, each rule belonging to a precise locus (modulo a translation in the case of periodic boundaries). But this question is of particular importance in the case of solutions found by evolutionary computation. In effect, the evolutionary runs demand lots of computational and time resources, an it would be useful to be able to quickly adapt the solution found to any desired grid size. Though Sipper and Tomassini had attained successful systems for a random number generation task using a simple scaling scheme involving the duplication of the rules grid [187], below we present a more sophisticated, empirically-obtained scheme that has proved successful. While not proving it, we will strongly argue in favor of its universality.

Given a, possibly evolved, non-uniform CA of size N, our goal is to obtain a grid of size N', where N' is given but arbitrary (N' may be $> N$ or $< N$), such that the original

performance level is maintained. This requires an automated procedure to determine which rule should be placed in each cell of the size N' grid, so as to preserve the original grid's "essence", i.e., its emergent global behavior; thus, we must determine what characterizes this latter behavior. We first noted two basic rule structures of importance in the original grid:

- The *local structure* with respect to cell i, $i \in \{0, \ldots, N-1\}$, in an $r = 1$ CA is the set of three rules in cells $i-1$, i, and $i+1$ (indices are computed modulus N since the grid is circular). This extends naturally for any r to the set of $2r+1$ rules in cells $i-r, ..., i, ..., i+r$.

- The *global structure* is derived by observing the *zones* of identical rules present in the grid. For example, for the following evolved $N = 15$ grid:

$R_1R_1R_1R_1$	R_2R_2	R_3	$R_4R_4R_4R_4$	R_1	$R_5R_5R_5$

where R_j, $j \in \{1, \ldots, 5\}$, denotes a distinct rule, the number of zones is 6, and the global structure is given by the list $\{R_1, R_2, R_3, R_4, R_1, R_5\}$.

We thought at the time that if these structures were preserved the scaled CA's behavior was identical to that of the original one. The heuristic principle we derived was thus to expand (or reduce) a zone of identical rules which spanned at least three cells, while keeping intact zones of length two or less. It is straightforward to observe that a zone of length one or two should be left untouched, so as to maintain the local structure. However we remarked that zones of length three were also to be left untouched to conserve the global behavior even if there was no a-priori reason as changing its size would have maintained both the global and the local structure.

Actually the "essence" of the CA may be represented in one structure, which is a bit more complex than both the local and the global structures defined earlier. This new structure includes the two old ones but also adds a third kind of interaction. It is defined as follows:

- Let s_i be the rule at position i (all positions are assumed to be modulus the size of the CA). Let $\mathcal{N}_i^r = (s_{i-r}, ..., s_i, ...s_{i+r})$, and $N_i^r = N_j^r$ if and only if $s_{i-r} = s_{j-r}$ and ... and $s_{i+r} = s_{j+r}$. Then:

  The essential structure of a one-dimensional non-uniform CA of radius r is the graph G_{CA}, where:
  — $\mathcal{N}_i^r$ is a node of G_{CA} if and only if $\mathcal{N}_i^r \neq \mathcal{N}_{i-1}^r$
  — $(\mathcal{N}_i^r, \mathcal{N}_j^r), i \neq j$ is an edge of G_{CA} if and only if $\forall k, i < k < j, \mathcal{N}_i^r = \mathcal{N}_k^r$
  — $(\mathcal{N}_i^r, \mathcal{N}_i^r)$ is an edge of G_{CA} if and only if $\mathcal{N}_i^r = \mathcal{N}_{i+1}^r$.

We remark that besides the local and the global structure, this takes into account what one could call the neighborhood interactions. This means that if this essential structure is preserved then the tuple of neighborhood rules of any cell in any neighborhood is left unchanged. For instance, the "leftmost" cell of the CA above has neighborhood rules (R_5, R_1, R_1), the left neighbor of this cell has neighborhood rules (R_5, R_5, R_1) and the right neighbor (R_1, R_1, R_1). Then any new CA conserving the essential structure will exhibit the same property.

It is clear that a simple heuristic to leave this essential structure unchanged is to add cells to any block of rules of size greater than or equal to $2r + 2$. In other word, our scaling algorithm gives us a minimal size of the CA, but no upper limit or no unreachable size above that lower ground. The minimal size is obviously attained when all blocks of cells of size bigger than $s_{min} = 2r + 2$, are reduced to this size, s_{min}. Thus, the CA above cannot be further reduced.

As an example of this procedure, consider the above $N = 15$ CA; scaling this grid to size $N' = 25$ results in:

$R_1R_1R_1R_1R_1R_1R_1R_1$	R_2R_2	R_3	$R_4R_4R_4R_4R_4R_4R_4R_4R_4R_4$	R_1	$R_5R_5R_5$

Note that the essential structure is preserved. We tested our scaling procedure on several CAs that were evolved to solve the synchronization task. The original grid sizes were $N = 100, 150$, which were then scaled to grids of sizes $N' = 200, 300, 450, 500, 750$. In all cases the scaled grids exhibited the same (perfect) performance level as that of the original ones. An example of a scaled system is given in Figure 4.2.

Discussion on the validity of this scheme, and how it relates to emergence: There is no way to effectively prove the universality or the validity of the scheme given that there is no *a priori* way to determine why a particular CA gives rise to a specific emergent global behavior. Actually, all the validity of the scheme reside in the definition of what is an *emergent global behavior*. In fact, I will reverse the problem and state the following definition:

Definition: Given a q state, non-uniform, one-dimensional CA, with rules $R_o, ..., R_n$, which on any and *every* input, $I \in q^n$ gives rise to the *global* behavior, B_I, then If any q state, non-uniform, one-dimensional CA, with rules $R'_o, ..., R'_m$, respecting the essential structure of the original CA, gives rise to the identical "scaled" global behavior B'_I, then the original CA (and the others) exhibit a truly *emergent global behavior*.

The validity of the scaling scheme resides on a number of features and thus this definition implies several limitations of what may be an emergent global behavior. First and foremost that the global behavior exhibited is perfect. In fact, I would conjecture that any imperfect behavior is the result of some sort of "cheating", i.e., no real computation occurs, and so no universality may be derived from such CA. Second, that the computation does not reside in any sort of special timing for collisions, as varying the block sizes reduces to nil such a brittle arrangement. Finally, that the computation is global, the cells are not differentiated, as any translation of the input should still give rise to the expected global behavior, and all the cells are used as input. This last point is really important as it precludes the idea of having "computing structures" designed in the CA as it is often the case in collision based computing (cf. [152]).

These restrictions, in my opinion, define well emergence in that scope. A property that is the fruit of *all* the cells *interacting locally*, that is *global*, and *universal*. Thus it is a property

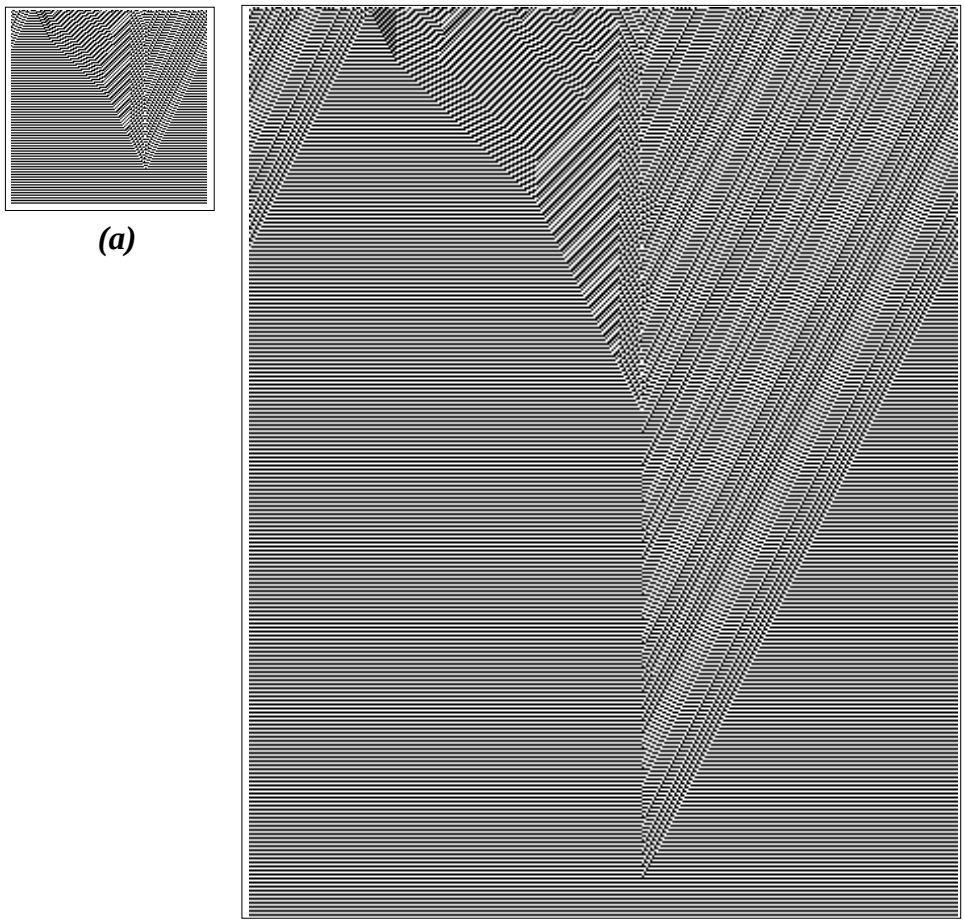


Figure 4.2 Scaling of one-dimensional synchronization task: Operation of a synchronous, non-uniform CA, with connectivity radius $r = 1$. (a) Evolved CA of size $N = 149$. (b) Scaled CA of size $N' = 350$.

that is not brittle, in the sense that it appears on any external conditions (the input states or size), but rather depends on the minute spatial arrangement of specific local behavior.

4.4 The Density Task

In this section, I concentrate on the density task problem in its original form, in its modified form and finally in its generalized form presented in section 4.2. This task has attracted much interest from various researchers in the Cellular Automata community as it seemed to be a prime example of global computation by means of CA. Our interest is rather mathematical and thus I will not consider imperfect solutions here[2].

I will first define a few notations that will be used throughout this section. I will then

[2]I study the process of evolving these imperfect solutions in chapter 5.

prove that the impossibility of designing a perfect, uniform, two-state, one-dimensional CA to solve the density classification task in its original form also holds for *non-uniform* CAs. This proves that the task is utterly impossible in its original definition. I will then show that by changing, but not complexifying, the task finds a solution in the simplest class of *uniform* CAs. Finally, I will then define a most general form of the density task to derive necessary conditions, even in higher dimensions, on the CA behavior to solve it.

4.4.1 Notation and definitions

A *configuration* is the state of all cells of the CA at a given time step. The *transition rule*, s, is the complete lookup table, delineating a cell's state at the next time step for every possible local configuration of neighboring states. The *successor function*, S, is derived by simultaneously applying s to the entire configuration yielding the configuration at the next time step. In the case of non-uniform CAs s_i denotes the transition table of cell i. σ denotes a configuration of states, σ_0 denotes the input configuration at time $t = 0$, and σ_t denotes the configuration at time step t, resulting from t successive applications of S to σ_0, i.e., $\sigma_t = S^t(\sigma_0)$.

Let $I(\sigma)$ be the number of 1s of configuration σ. Density $D(\sigma)$ thus equals $I(\sigma)/|\sigma|$, where $|\sigma|$ is the length (i.e., number of cells) of σ. The bitwise inversion of configuration σ is denoted $\overline{\sigma}$.

Following Wolfram's [223], the transition rule s can be written as a bit string; $D(s)$ is then defined as the density of *1s* in this bit string.

For one-dimensional CAs, $\sigma^{(i,j)}$ denotes the $|i - j|$ bits of configuration σ positioned between bits i and $(j - 1)$, inclusive. $(\sigma)^k$ is the concatenation of k configurations σ.

σ	A configuration of the cellular automata
σ_0	The input configuration, the configuration at $t = 0$
s, s_i	The transition rule, the transition rule of cell i
S	The results of applying s to all cell synchronously once
$S_i(\alpha)$	The results of applying S to all cell synchronously once to the partial configuration α applying rules from cell i
S^t	The results of applying s to all cell synchronously t times
σ_t	$\sigma_t = S^t(\sigma)$
σ^i, σ^{i-j}	The state of the cell i in σ, the states of cells i to $j - 1$ in σ
$I(\sigma)$	The number of 1s of σ
$\|\sigma\|$	The size of σ
$D(\sigma)$	The density of 1s in σ
$D(s)$	The density of 1s in the transition rule s
r	The radius of the CA considered (always uniform radius)

Table 4.1 Notations for the subsequent proofs.

In this the following subsection, we consider 2-state, one-dimensional *non-uniform CA*. In subsection 4.4.3, we consider a particular 2-state, r=1, one-dimensional *uniform CA*,

namely CA 184. Finally in subsection 4.4.4, we consider d-dimensional toroidal CAs, whose radius, r, is defined as an extension of the von-Neumann neighborhood: a cell has r neighbors on both sides of each dimension; in addition, the cell itself is included in its neighborhood.

All the notations are summed up in table 4.1. The general version of the density-classification problem is defined as follows:

Definition. Considering a toroidal, 2-state CA, a successor function S is said to be a *perfect density classifier*, if S, when applied to an arbitrary initial configuration of any length, progresses toward a configuration that allows to distinguish whether the density of 1s in the original configuration is greater or smaller than a predetermined threshold, ρ.

4.4.2 No two-state, non-uniform cellular automata can classify density

I propose here to extend[3] Land and Belew's proof to non-uniform CAs. More exactly, I prove that there is no two-state one-dimensional, non-uniform CA, which perfectly relaxes to an all 1s (resp. all 0s) fixed point configuration on any initial configuration whose density is greater (resp. lower) than a predetermined threshold ρ.

The idea of the proof is to show that given the partial configuration $0^{2r}10^{2r}$, then the number of 1s at the next time step, for any i, $I(S_i(0^{2r}10^{2r}))$, can neither be 0 nor 1 nor more than 1, thereby proving the impossibility to find such a perfect CA.

Theorem 2 *For a given neighborhood radius r, a density ρ and one-dimensional grid size N, $N > (4r/(1-\rho))$, there does not exist a two-state* non-uniform CA *such that on any input of size N, σ_0, it correctly relaxes to the fixed point all 1s when $D(\sigma_0) > \rho$ and to the fixed point all 0s when $D(\sigma_0) < \rho$.*

To prove that no CA can classify density by falling into either of the two fixed point all 0's or all 1's according to whether $D(\sigma_0)$ is greater than or less than a pre-specified ρ, $\frac{1}{N} < \rho \leq \frac{1}{2}$, we need first to prove three lemmas.

Lemma 2.1: If $D(\sigma_0)$ is greater than (respectively less than) ρ then for all t, $D(S^t(\sigma_0))$ is greater than (respectively less than) ρ.

Proof: Obvious, identical to the uniform case.

Lemma 2.2: Let s_i be any transition rule from a perfect CA, then $s_i(0^{2r+1}) = 0$ and $s_i(1^{2r+1}) = 1$.

Proof: Assume there exists i, such that $s_i(0^{2r+1}) = 1$, then $S(0^N) \neq 0^N$. But this contradicts the theorem assumption of the existence of a fixed point, therefore for all i, $s_i(0^{2r+1}) = 0$. Identically, we find that for all i, $s_i(1^{2r+1}) = 1$.

Lemma 2.3: Let S be the successor function of a perfect CA, and let α be the partial configuration $0^{2r}1^{2r}$ and β be the partial configuration $1^{2r}0^{2r}$ then $\forall i,j \;\; I(S_i(\alpha)) + I(S_j(\beta)) = 2r$.

Proof: First consider the configuration $\sigma = 0^{N-k}1^k$, where k is such that $k < \rho N < k+1$, then from Lemma 2.2 we know that $S(\sigma) = A0^{N-k-2r}B1^{k-2r}$, where $A = S_i(0^{2r}1^{2r})$ and

[3]This is not an extension as such of the Land and Belew's proof in the sense that it does not exploit the same idea.

$B = S_{i+k}(1^{2r}0^{2r})$. As we know from Lemma 2.1 that $I(S(\sigma)) \leq k$, then $I(A) + I(B) \leq 2r$. Now consider the configuration $\sigma' = 0^{N-k-1}1^{k+1}$, where k is such that $k < \rho N < k+1$, then $S(\sigma) = A'0^{N-k-1-2r}B'1^{k+1-2r}$, where $A' = S_i(0^{2r}1^{2r})$ and $B' = S_{i+k+1}(1^{2r}0^{2r})$. Given the value k, we have $I(A') + I(B') \geq 2r$. However, we can align σ and σ' so that $A = A'$, and then from the two inequalities we derive that $I(B') \geq I(B)$. With $\beta = 1^{2r}0^{2r}$, this means that $I(S_{i+k+1}(\beta)) \geq I(S_{i+k}(\beta))$. As the grid is toroidal, it is impossible that there exists an i such that $I(S_{i+k+1}(\beta)) > I(S_{i+k}(\beta))$, hence we conclude that $I(S_{i+k+1}(\beta)) = I(S_{i+k}(\beta))$ for any i. Then $I(B) = I(B')$ and thus we derive from the two inequalities that for all i, j $I(S_i(0^{2r}1^{2r})) + I(S_j(I1^{2r}0^{2r})) = 2r$, thereby proving the lemma.

We can now prove the theorem 2.

Proof of Theorem 2:

There are three possibilities for $I(S_i(0^{2r}10^{2r}))$:

- $I(S_i(0^{2r}10^{2r})) = 0$. Let us consider the configuration $\sigma = \alpha 0^{2r}10^{2r}$ and $\sigma' = \alpha 0^{2r}00^{2r}$, where $|\alpha|$ is such that $|\alpha 0^{2r}x0^{2r}| = N$. Then if we define $\delta = S_i(0^{2r}10^2r)$ and $\delta' = S_i(0^{2r}00^{2r})$, we can align identically σ and σ', so as to have $S(\sigma) = \beta\delta$ and $S(\sigma') = \beta'\delta'$, where $\beta = S_{i+1}(0^{2r}\alpha 0^{2r})$ and $\beta' = S_{i+1}(0^{2r}\alpha 0^{2r})$, thus $\beta = \beta'$. From Lemma 2.2, we know that for all i, $S_i(0^{2r}00^{2r}) = 0^{2r+1}$, as we assumed that $I(S_i(0^{2r}10^{2r})) = 0$, we then have $\delta = \delta'$. Hence, $I(S(\sigma)) = I(\beta\delta) = I(\beta'\delta') = I(S(\sigma'))$.

  However, as we assumed that $N > (4r/(1-\rho))$, we can define[4] α such that $I(\sigma) > \rho N$ and $I(\sigma') < \rho N$, but then from Lemma 2.1, it would be impossible that $I(S(\sigma)) = I(S(\sigma'))$.

  — We thus conclude that there is no i such that $S_i(0^{2r}10^{2r}) = 0$.

- $I(S_i(0^{2r}10^{2r})) > 1$. Consider the configuration $\sigma = 0^{2r}10^{N-2r-k}1^{k-1}$, $k < \rho N < k+1$. From the conditions on N stated in the theorem we know that $N - 2r - k \geq 2r$, thus $I(S(\sigma)) = I(S_i(0^{2r}10^{2r})) + I(S_{i+2r+1}(0^{2r}1^{k-1}0^{2r})$. But from lemmas 2.2 and 2.3, we straightforwardly derive that for any j, $I(S_{i+1}(0^{2r}1^{k-1}0^{2r}) = k - 1$, thus as $I(S_i(0^{2r}10^{2r})) > 1$, we have $I(S(\sigma)) \geq k + 1$, which contradicts Lemma 2.1.

  — Hence there is no i such that $I(S_i(0^{2r}10^{2r})) > 1$.

- $I(S_i(0^{2r}10^{2r})) = 1$. This actually reflects two cases, at the next step, either the 1 remains at the same cell, or the 1 is shifted. The former is easy to disqualify as a configuration $0^{N-1}1$ aligned on that cell i would never reach the correct classification 0^N. For the latter consider again the configuration $0^{N-1}1$, then the 1 shifted brings us back to the starting configuration $0^{N-1}1$, shifted by a few places. But we saw that whatever the shift at the next step there can be no place where the 1 is absorbed ($I(S_i(0^{2r}10^{2r}) = 0$), or multiplied ($I(S_i(0^{2r}10^{2r}) > 1$), thus it must be shifted forever and so $0^{N-1}1$ never classify to 0^N.

  — Hence for all i, $I(S_i(0^{2r}10^{2r})) \neq 1$

[4] Strictly speaking, for the sake of simplicity, we ignore here the case where $\lceil \rho N \rceil = \lfloor \rho N \rfloor$ as it would not change the general argument but would require to consider a configuration of the style $\alpha 0^{2r}10^{2r}\alpha' 0^{2r}10^{2r}$, lengthening uselessly the proof.

Hence, all three possibilities being impossible, we conclude that no two-state, non-uniform CA can classify density by reaching the fixed states 0^N or 1^N. QED.

4.4.3 A simple CA that solves the density problem

In this subsection I prove that there exists a *uniform* Cellular Automata that solves perfectly the *density classification problem* upon defining an output specification different than the original one.

In its original form (relaxing to all 0s or all 1s), it had been proven by [106] that for a one-dimensional grid of fixed size N, and for a fixed radius $r \geq 1$, there exists no two-state one-dimensional CA rule which correctly classifies all possible initial configurations. We just saw the extension of this proof to the non-uniform case. Actually up to the date of the proof I am going to present (1996, [26]), only convergence to one of two fixed-point configurations was considered. Here, I show that a perfect CA density classifier exists, upon considering a different output specification of no greater complexity.

Consider the two-state, $r = 1$ rule 184 CA, defined as follows:

$$\sigma_{t+1}^i = \begin{cases} \sigma_t^{i-1} & \text{if } \sigma_t^i = 0 \\ \sigma_t^{i+1} & \text{if } \sigma_t^i = 1 \end{cases}$$

where σ_t^i is the state of cell i at time t. Upon presentation of an arbitrary initial configuration, the grid relaxes to a limit-cycle, within $\lceil N/2 \rceil$ time steps, that provides a classification of the initial configuration's density of 1s: if this density > 0.5 (respectively, < 0.5), then the final configuration consists of one or more blocks of at least two consecutive 1s (0s), interspersed by an alternation of 0s and 1s; for an initial density of exactly 0.5, the final configuration consists of an alternation of 0s and 1s. The computation's output is given by the state of the consecutive block (or blocks) of same-state cells (Figure 4.3); as proved in this paper, this rule performs perfect density classification (including the density=0.5 case). We note in passing that the reflection-symmetric rule 226 holds the same properties of rule 184 studied below.

As the input configuration is random, this entails a high Kolmogorov complexity; intuitively, for a given finite string, this measure concerns the size of the shortest program that computes the string [116]. Both the fixed-point output of the original problem, as well as our own "blocks" output, involve a notable reduction with respect to this complexity measure. It has been noted by [132] that the computational complexity of the input is that of a non-regular language since a counter register is needed whose size is proportional to $\log(N)$, whereas the fixed-point output of the original problem involves a simple regular language (all 0s or all 1s) [86]; we note that our novel output specification also involves a regular language (a block of two state-0 or state-1 cells). We thus conclude that our newly proposed density classifier is as viable as the original one with respect to these complexity measures, while surpassing the latter in terms of performance. Lee *et al* [115] very recently briefly mis-argued that our CA required "global memory" due to the necessity to scan through the configuration. However, this is a misinterpretation of the output condition considered here: Firstly, if we scan through our grid a simple three state automata can detect any two

succeeding 1s or 0s, thereby giving out the good result, and not necessitating any sort of global memory; Moreover, the need for scanning is unnecessary as the resulting two cell block result will pass through all the cells in N time steps.

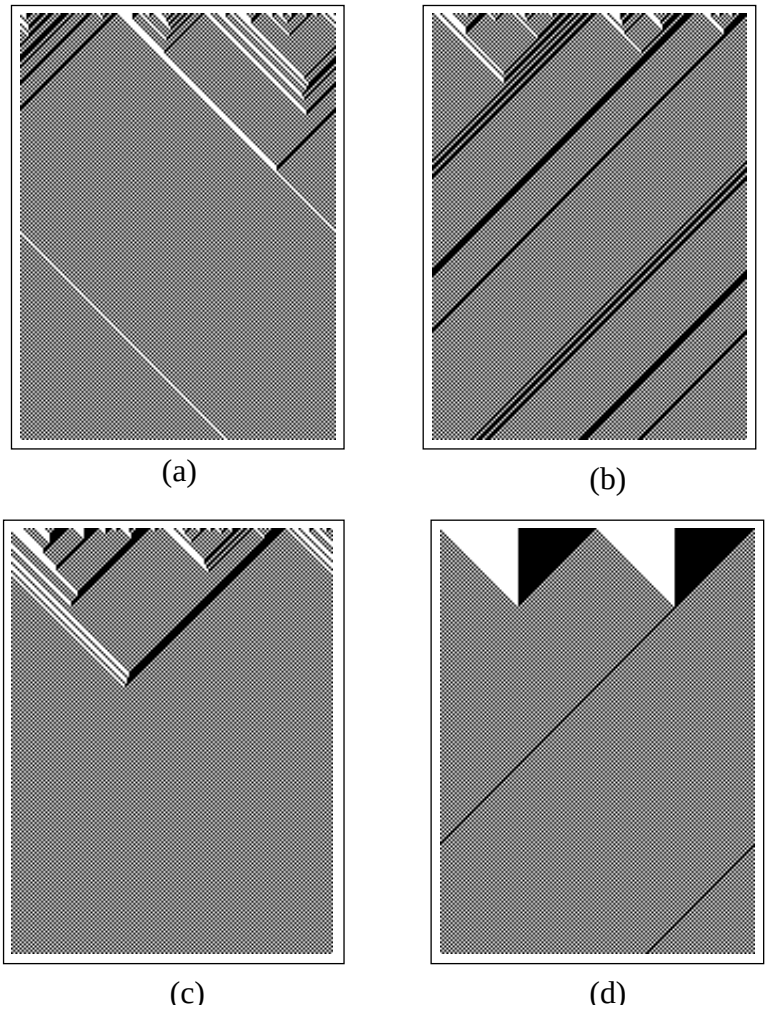


Figure 4.3 Density classification: Demonstration of rule 184 on four initial configurations. White squares represent cells in state 0, black squares represent cells in state 1. The pattern of configurations is shown for the first 200 time steps, with time increasing down the page. Initial configurations in figures (a)-(c) were randomly generated. (a) Grid size is $N = 149$. $D(S(0)) = 0.497$, i.e., 75 cells are in state 0, and 74 are in state 1. The final configuration consists of an alternation of 0s and 1s with a single block of two cells in state 0. (b) $N = 149$. $D(S(0)) = 0.537$. The final configuration consists of an alternation of 0s and 1s with several blocks of two or more cells in state 1. (c) $N = 150$. $D(S(0)) = 0.5$. The final configuration consists of an alternation of 0s and 1s. (d) $N = 149$. Initial configuration consists of a block of 37 zeros, followed by 37 ones, followed by 37 zeros, ending with 38 ones. The final configuration consists of an alternation of 0s and 1s with a single block of two cells in state 1. In all cases the CA correctly classifies the initial configuration.

In the remainder of this section, I prove rule 184's ability to perfectly classify density. Throughout, I assume that cellular indices are computed modulus the grid size N (grid is circular), and that they are in the range $\{0, \ldots, N-1\}$; for brevity we omit this range hereafter.

Theorem 3 *For a finite-size CA of size N, let $\sigma = \{\sigma_t^0, \ldots, \sigma_t^{N-1}\}$ be the grid configuration at time step t, let $D(\sigma_t^{i,(i+k-1)})$ be the density of 1s at time t over a block of k cells at positions $\{i, \ldots, i+k-1\}$, and let $T = \lceil N/2 \rceil$. Then:*

1. *If $D(\sigma_0) > 0.5$ then:*
   (a) *there exists a pair of adjacent cells $i, i+1$ such that*
   $\sigma_T^i = 1$ *and* $\sigma_T^{i+1} = 1$,
   (b) *for all i, $\sigma_T^i = 0 \Rightarrow \sigma_T^{i+1} = 1$.*
2. *If $D(\sigma_0) < 0.5$ then:*
   (a) *there exists a pair of adjacent cells $i, i+1$ such that*
   $\sigma_T^i = 0$ *and* $\sigma_T^{i+1} = 0$,
   (b) *for all i, $\sigma_T^i = 1 \Rightarrow \sigma_T^{i+1} = 0$.*
3. *If $D(\sigma_0) = 0.5$ then for all i, $\sigma_T^i \neq \sigma_T^{i+1}$).*

The proof of this theorem involves four lemmas, proved below.

Lemma 3.1.— For any configuration σ of size N, $D(S(\sigma_t)) = D(\sigma_t))$, $t \in \{0, 1, \ldots\}$, where S is the 184 transition function.

Proof: Taking note of the grid's circularity, any configuration σ_t can be expressed as 0^N, 1^N or $1^{a_1}0^{b_1}, \ldots, 1^{a_n}0^{b_n}$, where $n \geq 1$, $a_i, b_i > 0$, $i \in \{1, \ldots, n\}$, and $\sum_{i=1}^{n} a_i + b_i = N$. In the first two cases, we have $S(0^N) = 0^N$ and $S(1^N) = 1^N$, and the Lemma holds. In the latter case, it follows directly from the rule's definition that a block $1^{a_i}0^{b_i}$ at time t is transformed into $1^{a_i-1}010^{b_i-1}$ at time $t+1$. Thus, each such block conserves its density over one time step and the lemma is proven.

Corollary 3.1.1.— For any configuration σ of size N, $D(S(\sigma_t)) = D(\sigma_0))$, $t \in \{0, 1, \ldots\}$, where S is the 184 transition function.

Proof: Follows directly from Lemma 3.1 by recursive application.

Lemma 3.2.— Given a block $x0^\alpha 1^\beta y$, $x, y \in \{0, 1\}$, $2 \leq \alpha, \beta \leq N-2$, at time t, beginning at cell i (i.e., cell i is the block's leftmost cell), then at time $t+v$, $v = \min(\alpha, \beta) - 1$, and beginning at cell $i+v$:

1. If $\alpha > \beta$ there is a block $x0^{\alpha-\beta+1}1y$.
2. If $\beta > \alpha$ there is a block $x01^{\beta-\alpha+1}y$.
3. If $\alpha = \beta$ there is a block $x01y$.

Proof: Applying the transition rule to a block $x0^\alpha 1^\beta y$ at time t, beginning at cell i, results at time $t+1$ in a block $x0^{\alpha-1}1^{\beta-1}y$ beginning at cell $i+1$. By simple recursion, at time $t+u$, $u \leq \min(\alpha, \beta) - 2$, there is a block $x0^{\alpha-u}1^{\beta-u}y$, beginning at cell $i+u$.

Consider the case $\alpha > \beta$, and let $u = \beta - 2$. At time $t+u$, there is a block $x0^{\alpha-\beta+2}1^2y$, beginning at cell $i+u$. Applying the transition rule results at the next time step, $t+v$, $v = \beta - 1$, in a block $x0^{\alpha-\beta+1}1y$, beginning at cell $i+v$. The case $\beta > \alpha$ follows analogously.

For $\alpha = \beta$, at time $t + \beta - 2$ there is a block $x0^21^2y$, beginning at cell $i + \beta - 2$, which yields a block $x01y$ at the next time step, beginning at cell $i + \beta - 1$ $(= i + v)$.

Lemma 3.3.— Given a block 0^α (respectively, 1^α), $1 \leq \alpha \leq N$, at time t beginning at cell i, then at time $t-u$, $u \leq t$, there was a block 0^α (1^α) beginning at cell $i-u$ ($i+u$).

Proof: By contradiction (we prove the 0^α case, with the 1^α case following analogously). Suppose at time $t-1$ there exists $j \in \{i-1, \ldots, i+\alpha-2\}$, such that $\sigma_{t-1}^{j} = 1$. Given that $\sigma_t^{j+1} = 0$, this implies that $\sigma_{t-1}^{j+1} = 1$ and $\sigma_{t-1}^{j+2} = 0$; however, this results in $\sigma_t^j = 1$ and $\sigma_t^{j+2} = 1$, at least one of which must be within the 0^α block beginning at cell i, thus contradicting the assumption. Therefore, at time $t-1$ there is a block 0^α beginning at cell $i-1$; the lemma is proven by recursively applying this argument.

Lemma 3.4.— For a finite-size CA of size N, no two blocks 0^α and 1^β, $2 \leq \alpha, \beta \leq N-2$, can coexist at time $\lceil N/2 \rceil$.

Proof: By contradiction. If two blocks 0^α and 1^β coexist at time $\lceil N/2 \rceil$, then by lemma 3.3 they coexisted at time 0 (each shifted by $\lceil N/2 \rceil$ cells). This means that over the $\lceil N/2 \rceil$ time steps the blocks had been displaced by one cell per time step, the 0^α block "moving" to the right, the 1^β block moving to the left. Thus, both blocks would "meet" after $\lceil (N - (\alpha+\beta))/2 \rceil$ time steps at the latest, satisfying the conditions of lemma 3.2. This implies that at most one block would remain after $\lceil (N-(\alpha+\beta))/2 + \min(\alpha,\beta) - 1 \rceil$ time steps. This latter expression $< \lceil N/2 \rceil$, thus proving the lemma.

We now prove theorem 3.

Proof of Theorem 3:

According to corollary 3.1.1, the density of the initial configuration is preserved at each successive time step. According to lemma 3.4, after $\lceil N/2 \rceil$ steps only 0^α or 1^β blocks exist, $2 \leq \alpha, \beta \leq N-2$, but not both (except for density=0.5, where no such block exists). This means that the "correct" block must exist, with no occurrence of the "incorrect" one, thereby proving the theorem. We also note that after $\lceil N/2 \rceil$ time steps the number of cells in state 1 short (respectively, in excess) of $\lfloor 0.5N \rfloor$ is given by $(\sum_{i=1}^{m} a_i) - m$, where m is the number of 0^{a_i} (1^{a_i}) blocks, and a_i are their respective sizes.

Note that in order to "read" the output one can either terminate the CA's execution after $\lceil N/2 \rceil$ time steps, or, alternatively, let it continue running (for a maximum of $N-1$ additional time steps) until the (cycling) two-cell, same-state block arrives at two predetermined cells.

Are there any other density classifiers in the two-state, $r = 1$ class of CAs? An exhaustive check shows that rules 184 and 226 are the only ones that perform perfect density classification with respect to the output specification discussed in this paper [5].

In summary, we have shown that a locally-specified, $r = 1$ CA of any finite size N can classify the global density of bits for an arbitrary initial state configuration. It has previously been determined that this problem cannot be resolved by two-state CAs of any radius, if one insists on a fixed-point output. By changing the output specification, without increasing its complexity, perfect density classification can be attained. It is interesting that the system giving rise to this (difficult) emergent computation exists in the simplest class

[5] At the time, each of the 256 two-state, $r = 1$ CA rules was tested on 1000 randomly generated initial configurations for a grid of size $N = 149$, where a correct output is considered to be that specified by the theorem. As expected only rules 184 and 226 yielded a success rate of 100% followed by rule 57 and 99 trailing markedly behind at 60%. Interestingly, these latter two rules produce patterns that are visually similar to rules 184 and 226; however, the simple aforementioned test reveals their complete inadequacy.

of one-dimensional CAs, namely two-state, $r = 1$. This raises the intriguing question of whether other such simple CAs exist, which, while not capable of universal computation, may nonetheless prove highly efficient in solving specific tasks. This also brings out the question of the specification of the problem. With no more complexity we moved from a dead-end to a simple solution. This suggests that the question of what is computation, or more exactly what is the computational power of a CA is not straightforward. After proving some necessary conditions in the next subsection, we will discuss this important problem in subsection 4.5.

4.4.4 Necessary conditions on d-dimensional CA density classifiers

In this subsection I prove two necessary conditions that a *uniform* CA of any dimension must satisfy in order to classify density in its generalized form: (1) the density of the initial configuration must be conserved over time, and (2) the rule table must exhibit a density of 0.5.

A perfect density classifier must conserve density: The one-dimensional case

I now prove that a perfect CA density classifier cannot alter the density of the input configuration. I first prove this result for one-dimensional CAs for sake of simplicity and then provide a 'straightforward' extension of the proof to any dimension.

Theorem 4 *Let S be a successor function of a perfect one-dimensional density classifier. Then:* $\forall \sigma_0, \forall t, D(\sigma_0) = D(S^t(\sigma_0))$.

The proof of this theorem involves five lemmas proved below.

Lemma 4.1: Let S be a perfect density classifier successor function. Then, $\forall \sigma_0, \forall t$, $D(\sigma_0) < \rho \Rightarrow D(S^t(\sigma_0)) < \rho$ and $D(\sigma_0) > \rho \Rightarrow D(S^t(\sigma_0)) > \rho$.

Proof: Follows straightforwardly from the above definition of the density-classification problem.

Lemma 4.2: Let s be the transition rule of a perfect density classifier with radius r. Then, $s(0^{2r+1}) = 0$ and $s(1^{2r+1}) = 1$.

Proof: If $s(0^{2r+1}) = 1$ and $s(1^{2r+1}) = 1$, or $s(0^{2r+1}) = 0$ and $s(1^{2r+1}) = 0$, then the input configurations 0^n and 1^n, where n is the size of the CA, are classified as belonging to the same class, thus contradicting s's being a perfect density classifier transition rule. If $s(0^{2r+1}) = 1$ and $s(1^{2r+1}) = 0$, then 0^n and 1^n give rise to a cycle of alternating configurations, thus contradicting s's being a perfect density classifier[6].

[6]In theory, we could insist that s stop at a given, predetermined time step, thus enabling the use of $s(0^{2r+1}) = 1$ and $s(1^{2r+1}) = 0$ in a classifier. This would impose an additional temporal constraint I wish to avoid for the sake of simplicity. Remark that Theorem 4 would still hold if I were to include this constraint.

Lemma 4.3: For any one-dimensional input configuration σ_0 of size n, and for any density threshold ρ, there exist m_0, m_1 such that $D(0^{m_0}\sigma_0 1^{m_1}) > \rho$ and $D(0^{(m_0+1)}\sigma_0 1^{(m_1-1)}) < \rho$.

Proof: Assuming $1/(1-\rho)$ is not an integer, then, setting $m_0 + m_1 = \lceil n/(1-\rho) \rceil - n$, it is straightforward to see that if $I(\sigma_0) = 0$, we can set $m_1 = \lceil n/(1-\rho) \rceil - n$ and $m_0 = 0$, with the result that $D(0^{m_0}\sigma_0 1^{m_1}) > \rho$ and $D(0^{(m_0+1)}\sigma_0 1^{(m_1-1)}) < \rho$. Now, if $I(\sigma_0) \neq 0$, then decreasing m_1 by $I(\sigma_0)$ and increasing m_0 by the same amount will satisfy the lemma.

If $1/(1-\rho)$ is an integer, then setting $m_0 + m_1 = \lceil n/(1-\rho) \rceil - n + 1$ leads to the same result.

Lemma 4.4: Let S be the successor function for a one-dimensional CA, σ_0 an initial configuration, and p an integer, such that $I(S(\sigma_0)) = I(\sigma_0) + p$. Then, $I(S((\sigma_0)^k) = I((\sigma_0)^k) + kp$.

Proof: As our CAs are toroidal, $S((\sigma_0)^k) = (\sigma_1)^k$. Then, $I(S((\sigma_0)^k)) = I((\sigma_1)^k) = k * I(\sigma_1) = k * I(\sigma_0) + kp = I((\sigma_0)^k) + kp$.

Lemma 4.5: Let S be a successor function of a perfect one-dimensional density classifier, and let r be the radius of the CA. For any configuration σ_0, if $I(S(\sigma_0)) = I(\sigma_0) + p$ then $-4r \leq p \leq 6r$.

Proof: Let σ_0 be a configuration such that $I(S(\sigma_0)) = I(\sigma_0) + p$. Define a configuration v_0, such that $v_0 = 0^{m_0} R_1 \sigma_0 R_2 1^{m_1}$, where $R_2 = \sigma_0^{(0,r)}$ and $R_1 = \sigma_0^{(n-r,n)}$ and $m_0, m_1 \geq 2r+1$.

Then, given Lemma 4.2 and our definition of R_1 and R_2, we conclude that $S(v_0) = C_1 0^{m_0-2r} C_2 \sigma_1 C_3 1^{m1-2r}$, where C_1 is the $2r$-bit-long configuration obtained at the border of $1^{2r}0^{2r}$, C_2 is the r-bit-long configuration obtained at the border of $0^{2r}R_1$, and C_3 is the r-bit-long configuration obtained at the border of $R_2 1^{2r}$.

From Lemma 4.3 we know that we can define m_0, m_1 such that $D(v_0) > \rho$ and that if we decrease m_1 by 1 and increase m_0 by 1, $D(v_0) < \rho$. (Note that we can increase both m_0 and m_1 by $2r+1$ so that $m_0, m_1 \geq 2r+1$ as required earlier). Then, as $D(v_0) > \rho$, we know that $D(v_1) > \rho$ (Lemma 4.1), which, given the chosen values of m_0, m_1, implies that $I(v_1) \geq I(v_0)$. Expanding $I(v_1)$ and $I(v_0)$, we can derive that $I(C_1) + I(C_2) + I(C_3) + p - 2r - I(R_1) - I(R_2) \geq 0$.

Analogously, if we define m_0, m_1 such that $D(v_0) < \rho$ and that if we decrease m_0 by 1 and increase m_1 by 1, $D(v_0) > \rho$. Then, as $D(v_0) < \rho$, we know that $D(v_1) < \rho$ (Lemma 4.1), which, given the chosen values of m_0, m_1, implies that $I(v_1) \leq I(v_0)$ from which we derive that $I(C_1) + I(C_2) + I(C_3) + p - 2r - I(R_1) - I(R_2) \leq 0$.

Hence, we know that $I(C_1) + I(C_2) + I(C_3) + p - 2r - I(R_1) - I(R_2) = 0$, meaning that p—the variation of number of $1s$ between σ_0 and σ_1—is exactly equal to $I(R_1) + I(R_2) - I(C_1) - I(C_2) - I(C_3) + 2r$. Given the length of R_1, R_2, C_1, C_2, C_3, we compute that $-4r \leq p \leq 6r$.

We are now able to prove Theorem 4.

Proof of Theorem 4. We will proceed by contradiction.

Assume there exists a configuration σ_0, such that $I(S(\sigma_0)) = I(\sigma_0) + p$, p a non-zero integer. From Lemma 4.4 we know that we can create a configuration $\tau_0 = (\sigma_0)^k$, such that $I(S(\tau_0)) = I(\tau_0) + kp$. However, if we set $k = 7r$, where r is the radius of the CA in question, then we have a configuration τ_0, wherein $I(S(\tau_0)) = I(\tau_0) + 7rp$, which contradicts Lemma

4.5, since $p \neq 0$.

Hence $p = 0$, and thus, for all configurations σ_0, $I(S(\sigma_0)) = I(\sigma_0)$.

A perfect density classifier must conserve density: The d-dimensional case

This section is an extension of the previous one. However, I chose to split the proof into two parts, the one-dimensional case and its generalization, to provide a simpler and more straightforward approach to the proof.

Theorem 4(d) *Let S be a successor function of a perfect d-dimensional density classifier.*

Then: $\forall \sigma_0, \forall t, D(\sigma_0) = D(S^t(\sigma_0))$.

Lemmas 4.1(d) and **4.2(d)** follow straightforwardly from their one dimensional counterpart. I now provide the proof of the other lemmas for the d-dimensional case.

Lemma 4.3(d): For any d-dimensional input configuration σ_0 of size $n_1 * ... * n_d$, where $n_1 * ... * n_d$ are the dimensions along each dimension, and for any density threshold ρ, there exist two d-dimensional configurations A_0 and B_0 of size $m_1 * n_2 * n_3 * ... * n_d$ and $p_1 * n_2 * n_3 * ... * n_d$, with $\forall x \in A_0, x = 0$ and $\forall x \in B_0, x = 1$, such that $D(A_0\sigma_0 B_0) > \rho$ and $D(A'_0\sigma_0 B'_0) < \rho$, where A'_0 is the block of 0 of size $(m_1 + 1) * n_2 * n_3 * ... * n_d$ and B'_0 is the block of 1 of size $(p_1 - 1) * n_2 * n_3 * ... * n_d$.

Proof: It follows quite naturally from the one-dimensional lemma 4.3, that if such a combination of "borderline" m_1, p_1 exist with a variation of 1, then they obviously exist with a variation $n_2 * n_3 * ... * n_d$, given these are non null.

Lemma 4.4(d): Let S be the successor function for a perfect d-dimensional CA, σ_0 an initial configuration, and p an integer, such that $I(S(\sigma_0)) = I(\sigma_0) + p$. Then, $I(S((\sigma_0)^k) = I((\sigma_0)^k) + kp$, where $(\sigma_0)^k$ is the stacking up along *one* dimension of k configuration σ_0.

Proof: As our CAs are toroidal, $S((\sigma_0)^k) = (\sigma_1)^k$. Then, $I(S((\sigma_0)^k)) = I((\sigma_1)^k) = k * I(\sigma_1) = k * I(\sigma_0) + kp = I((\sigma_0)^k) + kp$.

Lemma 4.5(d): Let S be a successor function of a perfect d-dimensional density classifier, and let r be the radius of the CA in the first dimension. For any configuration σ_0 of dimension $n_1 * ... * n_d$, if $I(S(\sigma_0)) = I(\sigma_0) + p$ then $-4r * (n_2 * ... * n_d) \leq p \leq 6r * (n_2 * ... * n_d)$.

Proof: Quite straightforwardly, following the one-dimensional proof, if we define $R_1 = \sigma_0^{\{(0,r),n_2,...,n_d\}}$ and $R_2 = \sigma_0^{\{(n_1-r,n),n_2,...,n_d\}}$ then we can define a d-dimensional υ_0, $\upsilon_0 = 0^{\{m_1,n_2,...,n_d\}} R_1 \sigma_0 R_2 1^{\{p_1,n_2,...,n_d\}}$. Given that we proved Lemma 4.2(d) and 4.3(d) we can follow exactly the same proof as for the one dimensional case, to obtain the bounding range for p.

Proof of Theorem 4(d). We will proceed by contradiction.

Assume there exists a d dimensional configuration σ_0, such that $I(S(\sigma_0)) = I(\sigma_0) + p$, p a non-zero integer. From Lemma 4.4(d) we know that we can create a configuration $\tau_0 = (\sigma_0)^k$, the stacking along dimension one, such that $I(S(\tau_0)) = I(\tau_0) + kp$. However, if we set $k = 7r * n_2 * ... * n_d$, where r is the radius of the CA in question along that dimension, then we have a configuration τ_0, wherein $I(S(\tau_0)) = I(\tau_0) + (7r * n_2 * ... * n_d) * p$, which contradicts Lemma 4.5(d), since $p \neq 0$.

Hence $p = 0$, and thus, for all configurations σ_0, $I(S(\sigma_0)) = I(\sigma_0)$.

A perfect density classifier's rule must exhibit a density of 0.5

Having obtained a necessary condition on the global successor function S, I prove in this section a theorem relating to the local transition rule, s, namely, it must exhibit a density of 0.5. Thus, from a constraint on the function, I derive a constraint on the form.

Theorem 5 *Let s be the transition rule of a perfect, 2-state, toroidal density classifier of any dimension. Then, for any density threshold of 1s, ρ, $D(s) = 0.5$.*

The proof of this theorem involves five lemmas and a result on consecutive-l graphs proved by [42].

A *consecutive-l graph*, $G(l, n, q, h)$, is an n-node directed graph, wherein exists an edge, (i, j), iff $j \in \{qi + k(\text{mod n}) : h \leq k \leq h + l - 1\}$. Du *et al.* [42] proved that such a graph contains a Hamiltonian cycle if $q = l, r = 0$, and $h \geq gcd(n, q) \geq 2$.

Lemma 5.1: For any radius r, one-dimensional, 2-state toroidal CA, there exists a configuration σ_0 of length 2^{2r+1}, such that all 2^{2r+1} neighborhoods are present once and only once.

Proof: Consider the directed graph G, whose vertices are the 2^{2r+1} binary numbers $0, ..., 2^{2r+1} - 1$, defined as follows: there is an edge from vertex v_n to vertex v_m, iff the last $2r$ bits of v_n are identical to the first $2r$ bits of v_m. Then finding a Hamiltonian cycle in G is equivalent to finding an input configuration σ_0 satisfying the conditions of the Lemma.

The set of edges of G can be defined as follows: $i \rightarrow j$ if $j \in \{2i + k(\text{mod n}) : 0 \leq k \leq 1\}$. We thus obtain a *consecutive-2* directed graph, $G(l, n, q, h)$, with $q = l = 2$ and $h = 0$. As the number of nodes n is a power of 2, we have $q = l, h = 0$ and $h \geq gcd(n, q) \geq 2$. Thus, following the results of [42], G contains a Hamiltonian cycle, thereby proving the lemma.

Lemma 5.2: Let σ_0 be a d-dimensional configuration of length 2^{2dr+1}, such that all 2^{2dr+1} neighborhoods of a *d-dimensional* CA are present once and only once. Then, $\overline{\sigma_0}$, the bitwise inversion of σ, is also such a configuration.

Proof: Consider any two of the 2^{2dr+1} possible neighborhoods of $\overline{\sigma_0}$: $\overline{a}$ and $\overline{b}$. Then, by definition, there exist a, b, the two corresponding neighborhoods of σ_0. As each neighborhood is present once and only, $a \neq b$, and thus $\overline{a} \neq \overline{b}$. Given that there are only 2^{2dr+1} neighborhoods in $\overline{\sigma_0}$, and given that there are 2^{2dr+1} possible different neighborhoods for a d-dimensional CA, then all neighborhoods are present once and only once in $\overline{\sigma_0}$.

Lemma 5.3: For any r, there exists a one-dimensional, 2-state configuration σ_0 of length 2^{2r}, such that for any 2 blocks a, b of σ_0 of length 2^{2r+1}, $a \neq b$.

Proof: One may see that the proof of *Lemma 5.1* still holds for even power of 2. Thus we know that there exists a configuration of length 2^{2r}, such that any blocks $a_i ... a_{(i+2r-1) \, mod \, 2^{2r}}$ is different from any other block $a_j ... a_{(j+2r-1) \, mod \, 2^{2r}}$, $i \neq j$. It is obvious that in such a configuration, any block $a_i ... a_{(i+2r) \, mod \, 2^{2r}}$ is thus different from any other block $a_j ... a_{(j+2r) \, mod \, 2^{2r}}$, $i \neq j$.

Lemma 5.4: For any d-dimensional, 2-state toroidal CA, and for any radius r, there exists a configuration, wherein all 2^{2dr+1} possible neighborhoods are present once and only once.

Proof: We will prove this lemma by induction.

The base of the induction, $d = 1$, is proved by Lemma 5.1.

Induction step: Assume a d-dimensional configuration σ_0 that includes all 2^{2dr+1} possible neighborhoods, each present once and only once.

We next construct $\beta_0 = \alpha_1 ... \alpha_{2^{2r}}$, the $d+1$-dimensional configuration, by "stacking up" along the $(d+1)$-th dimension 2^{2r} $\alpha's$, where $\alpha \in \{\sigma_0, \overline{\sigma_0}\}$. We construct the sequence $\alpha_1 ... \alpha_{2^{2r}}$, such that any block $\alpha_i ... \alpha_{(i+2r) \bmod 2^{2r}}$ is different from any other block $\alpha_j ... \alpha_{(j+2r) \bmod 2^{2r}}$, $i \neq j$. One can see this is possible: if we denote the case $\alpha = \sigma_0$ by 0 and the case $\alpha = \overline{\sigma_0}$ by 1, we can then invoke Lemma 5.3.

From the induction assumption and from Lemma 5.2, we know that along each hyperplane α_i there are 2^{2dr+1} different neighborhoods. Each of these neighborhoods includes along its $(d+1)$-th dimension the sequence of bits $b_{(i-r) \bmod 2^{2r}} ... b_{(i+r) \bmod 2^{2r}}$. We know that this sequence is different for each hyperplane from the construction constraint that any block $\alpha_i ... \alpha_{(i+2r) \bmod 2^{2r}}$ is different from any other block $\alpha_j ... \alpha_{(j+2r) \bmod 2^{2r}}$, $i \neq j$. Thus, all 2^{2dr+1} different neighborhoods on hyperplane α_i are different from all 2^{2dr+1} different neighborhoods on hyperplane α_j, $i \neq j$. Then, we know that we have $2^r * 2^{2dr+1} = 2^{2(d+1)r+1}$ different neighborhoods in β_0, which is also the maximum number of possible neighborhoods. Thus, β_0 is a configuration of dimension $d+1$, in which all $2^{2(d+1)r+1}$ possible neighborhoods are present once and only once. This proves the induction step, d to $d+1$.

Lemma 5.5: Let σ_0 be a d-dimensional configuration, such that all 2^{2dr+1} possible input states are present once and only once, in any dimension d. Then, $D(\sigma_0) = 0.5$.

Proof: The density of all neighborhoods, i.e., the density of all the numbers from 0 to 2^{2dr+1}-1 is 0.5. When "moving" along σ_0 to collect all neighborhoods, each bit is counted exactly the same number of times, namely, $2dr + 1$ times. Thus, the density of σ_0 is the same as the density of all possible neighborhoods, i.e., 0.5.

We are now able to prove Theorem 5.

Proof of Theorem 5: Assume configuration σ_0 contains all 2^{2dr+1} possible neighborhoods once and only once. From Lemma 5.4 we know that such a σ_0 exists. From Theorem 4 we deduce that—given that S is a perfect density classifier successor function—$D(S(\sigma_0)) = D(\sigma_0)$, which, from Lemma 5.5, we know to be 0.5. Moreover, as all 2^{2dr+1} possible neighborhoods are present once and only once, then $D(S(\sigma_0)) = D(s)$, and hence $D(s) = 0.5$.

I should point out that recently Boccara and Fukś presented a proof that a toroidal, uniform, *one-dimensional* CA is number-conserving on inputs of any size if it is number conserving for all the configurations of length $4r + 1$,[21]. This property may turn out to be useful if we are to seek number-conserving CAs. In the same paper, they also prove a generalized version of the result above. However, one should note their result is only for one-dimensional CAs while ours is valid for any dimension.

4.4.5 An aside: No uniform CA solves perfectly the sorting task

Oliveira *et al* [145] have recently proposed to evolve solutions for a new task: the sorting task. This task is an extension of the sorting task presented by Sipper *et al* [184] to toroidal

CAs. We prove here that no CA solves perfectly this task.

Theorem 6 *There exists no toroidal, uniform, two-state CAs such that:*
$\forall \sigma_0, \exists t, \Rightarrow\ S^t(\sigma_0) = 0^{n_0}1^{n_1}$, *where* $n_1 = I(\sigma_0)$.

The proof of this theorem involves just one lemma.

Lemma 6.1: Let S be a perfect sorting CA then S conserves density.

Proof: Quite evidently if there exists t and σ_0 such that $I(S^t(\sigma_0)) \neq I(\sigma_0)$ and S correctly sorts σ_0, then S incorrectly sorts $S^t(\sigma_0)$. By contradiction, we conclude that S necessarily conserves density.

Proof of Theorem 6: Let's consider the configuration $\sigma_0 = 10^m10^p$, where m, p larger than the radius of the CA considered. From lemma 6.1, we know that there are necessarily two and only two 1s in the succeeding configuration. As m and p are larger than the radius of the CA, and the CA is uniform, then the two ones may only be *both* shifted to the left, or *both* to the right or *both* unmoved, thereby proving that there exists *no* t such that $S^t(\sigma_0) = 110^{m+p}$. QED.

We can note here that the proof of lemma 6.1 is also a corollary of theorem 4. If we consider the two separate sets of configuration A and B defined as follows $A = \{0^{n_0}1^{n_1} | n_0 < n_1\}$, $B = \{0^{n_0}1^{n_1} | n_0 > n_1\}$, then it would classify the input configurations according to whether their density is above or below $\rho = 0.5$ into the distinct classes A and B.

4.5 Concluding Discussion

We have shown that according to the task definition, density classification could change status dramatically in terms of "CA" computability. It moved from utterly impossible for two-state CAs, (uniform and non-uniform!), to "basic" as it found a solution in the simplest class: the elementary CAs. These results led us to wonder on what was required to solve that task in its essential form. Essential in the sense of stripping down its definition to the minimal form beyond which the task loses its meaning. From these minimal assumptions, we derived that a perfect CA density classifier must conserve in time the density of the initial configuration, and that its rule table must exhibit a density of 0.5. Thus, non-density-conserving CAs (such as the GKL rule) are by nature imperfect, and, indeed, any specification of the problem which involves density change precludes the ability to perform perfectly density classification.

The necessary condition of density conservation brings out the question of what is computation by means of cellular automata. If we conserve density, then computation here is only re-ordering the "1s" among the "0s", which strictly speaking means no computation. As a consequence, there is no loss of information through time[7]. What we are looking for, hence, is not a simplification of the input. This remark is far from being intuitive: the original question was it not to reduce any configuration, 2^N bits, to an answer yes or no, 1 bit? So beyond the aid that this result on necessary conditions might give in the search for locally interacting systems that compute the global density property, it is its consequences on the question of computation that makes it important. This question was already made

[7]This is not true of spatial information as rule 184 is not invertible.

particularly pregnant when it was discovered that the computability of the task was more dependent on its definition rather than its inherent difficulty.

All these considerations call for a reflection on the question of what constitutes computation in CAs. I am not pretending, here, to give a definitive answer nor a universal one, but rather some tracks for thought. If computation here is only reordering, then what constitute the result is actually the *visual efficiency* of the final (or temporal) configuration of the CA. That is, if one watches Figure 4.3, one can instantly say if the original configuration was holding more 1s than 0s or the contrary. Of course, this translates also into the fact that a simple three state automaton can then classify the density, but the most convincing argument that CA 184 does the job is that it is visually efficient. It is hard to define formally what is this efficiency, but we can say, without doubt, at least in this case that it relies on patterns that become visibly obvious as they stand out of a regular background. This view is also what changed between the impossibility of the task in its original form to its evident solution. So the question of computation is finally the hazardous meeting of an actual computation by CAs with the "good" look from an outside observer.

In the future, our search for a perfect density classifier, in dimensions higher than one for instance, will thus be limited to density conserving CAs, but more importantly will surely rely on defining an automatic observer. We could imagine a mechanical search (either exhaustive of by means or evolutionary computation) where the success criteria would be given by an artificial observer. We evoked in a preceding chapter the work of Wuensche [231] in which he proposed a measure of input entropy, based on the frequency of look up of the different neighborhoods. These measures were devised in order to find "interesting" CAs, interesting in the sense of Wolfram's class III. Wuensche follows the ideas originated by Gutowitz and Langton: is there a quantitative way to define class III CAs, [75], and can we find them by means of artificial evolution [76]? But all these works were somehow seeking an artificial observer. Class III was firstly (and still is) only defined on criteria based on observation. So our aim, in future research, will be the same in spirit. However it will be totally different in practice. Actually, what we said before was that interesting CAs for us were CAs which produced regularity, or more exactly irregularity (patterns) on regularity, and these are surely to be found in class II rather than class III. Nevertheless, Wuensche's frequency measure may be a good start for our work. He introduced the idea of filtering the output of the CAs by omitting in the visual representation the neighborhoods that were used the most often. For instance, this would lead in CA 184 to discard patterns 101 and 010. This is definitely an interesting path to follow to discard regularity, and thereby provides a first step in the direction of an automatic observer. Our future research will concentrate on extending this idea to two-dimensional CAs. If one may wonder why try to solve the density task for two and higher dimension, besides the pure research interest, an answer is that it could be a way to speed-up the computation. In one dimension, obviously, we can define easily a CA with radius $r = k$ to produce a linear k speed-up. But as the computation here is basically dependent on the time a cell takes to go through the grid, then one may hope that in two dimensions we may find a CA solving the density task in $O(\sqrt[2]{n})$ time. If this is the case, it is probable that a *theoretical* d-dimensional CA would solve the task in $O(\sqrt[d]{n})$ time.

Nous ne contemplons pas une nature gouvernée par la Loi, mais
par un processus aléatoire et sans but, et le régne lugubre
de la chance remplace les principes directeurs de la raison[1].

Emmanuel Kant, *Idée d'une Histoire universelle au point de vue cosmopolitique. [94].*

Chapter 5

Evolution of Cellular Automata

5.1 Introduction

In this chapter, we propose to study statistically a peculiar kind of evolutionary algorithm. The motivation of this work is quite evident when we think about the lack of understanding of the inner workings of evolutionary algorithms in general, examples of which we saw in both chapters 2 and 3. However, our purpose here is not to give some results about evolutionary computation in general, but rather to concentrate on a special kind of algorithm: Structured, fine-grained parallel evolutionary algorithms. To do this, we study *the cellular programming algorithm* presented by Sipper in [180].

The cellular programming algorithm is a means to evolve the rules for problem-solving non-uniform Cellular Automata. As we saw in chapter 4, designing Cellular Automata rules to attain a predefined global behavior is not computationally tractable nor mathematically easy. Actually, most of the time, it is impossible to systematically deduce global behavior from the local rules. So, it is quite natural that evolutionary computation techniques are used to design cellular automata rules [132]). Specifically, Sipper proposed in the mid 90's to evolve non-uniform Cellular Automata [179]. These allow more complex behavior than uniform CA at, almost, no extra cost in terms of hardware. However, the search space for such CA is bigger by several order of magnitude than the one for uniform CA. The cellular programming algorithm not only succeeds in finding good problem solving CA but does it faster than a classical GA. It is a prime example of an evolutionary algorithm which fully exploit their inherent parallelism, evolve a strictly spatially structured population, and aim at a good general fitness rather than one super-individual.

We begin, in the first subsection, by treating in more detail the question of parallelism which is at the core of cellular systems and their evolution. We then present in subsection 5.1.2 the cellular programming algorithm. Section 5.2 introduces the various statistical measures used in the analysis of cellular evolutionary algorithms. In Section 5.3, we analyze

[1]We are not contemplating a Nature governed by Law, but rather a random and goalless process, and the dismal reign of chance replaces the guiding principles of reason.

the dynamics, both genotypic and phenotypic of the cellular programming algorithm when used to evolve solutions to three different problems: density, synchronization, and random number generation. Finally, we conclude in Section 5.4. The work presented in this chapter is an augmented and completed version of the results published by Capcarrere *et al* [27].

5.1.1 Parallel evolutionary algorithms

One of the most basic aspects of evolutionary algorithms is their inherent parallelism: the existence of a population implies, *ipso facto*, that there are several individuals evolving in parallel. This has not escaped practitioners in the field, who have indeed explored the issue of parallel evolutionary algorithms. One can cite two basic motivations underlying these parallelization efforts. First, there is the wish to reduce (often quite markedly) the necessary run time, thus expediting the emergence of a solution to the problem at hand. A second motivation lies in the algorithmic benefits resulting from a parallel implementation that echoes evolution in nature—the field's fundamental inspiration.

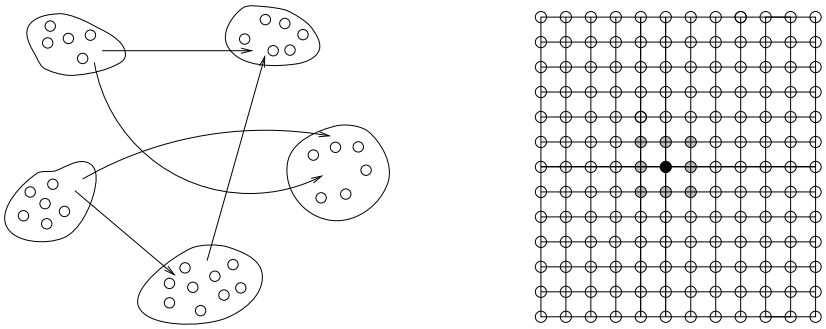


Figure 5.1 The two basic models of parallel evolutionary algorithms: (a) the coarse-grained island model, and (b) the fine-grained grid (or cellular) model.

A basic tenet of parallel evolutionary algorithms is that the population has a spatial structure. A number of models based on this observation have been proposed, the two most important being the *island* model and the *grid* model. The coarse-grained island model features geographically separated subpopulations of relatively large size. Subpopulations exchange information by having some individuals migrate from one subpopulation to another with a given frequency and according to various migrational patterns (Figure 5.1a). This can work to offset premature convergence, by periodically reinjecting diversity into otherwise converging subpopulations. In the fine-grained grid model individuals are placed on a toroidal d-dimensional grid (where $d = 1, 2, 3$ is used in practice), one individual per grid location (the fine-grained approach is also known as *cellular* [211, 220]; see Figure 5.1b). Fitness evaluation is done simultaneously for all individuals, with genetic operators (selection, crossover, mutation) taking place locally within a small neighborhood. From an implementation point of view, coarse-grained island models, where the ratio of computation to communication is high, are more adapted to multiprocessor systems or workstation clusters, whereas fine-grained cellular models are better suited for massively parallel machines

or specialized hardware. Hybrid models are also possible, e.g., one might consider an island model in which each island is structured as a grid of locally interacting individuals. For recent reviews of parallel evolutionary algorithms (including several references) the reader is referred to [24, 205].

Though such parallel models have empirically proven worthwhile [9, 32, 122, 124, 146, 196, 211], there seems to be lacking a better understanding of their workings. Gaining insight into the mechanisms of parallel evolutionary algorithms is the underlying motivation of our chapter. Specifically, concentrating on cellular models, our objectives are: (1) to introduce several statistical measures of interest, both at the genotypic and phenotypic levels, that are useful for analyzing the workings of fine-grained parallel evolutionary algorithms, and (2) to demonstrate the application and utility of these measures on a specific example, that of the cellular programming evolutionary algorithm [180]. Among the few theoretical works carried out to date, one can cite Mühlenbein [135], Cantú-Paz and Goldberg [25], and Rudolph and Sprave [166]. The latter treated a special case of fine-grained cellular algorithms, studying its convergence properties.

5.1.2 Cellular programming

In this chapter we investigate the evolution of *non-uniform cellular automata*. The CA model for which we presented a scaling scheme in chapter 4, so as to generalize the solution obtained by evolution. Such automata function in the same way as uniform ones, the only difference being in the cellular rules that need not be identical for all cells. Our focus here is on the *evolution* of non-uniform CAs to perform computational tasks using *the cellular programming approach*. In this section, we present the cellular programming algorithm, the subject of our statistical analysis. This algorithm was introduced by Sipper in [179].

The cellular programming algorithm

We study 2-state, non-uniform CAs, in which each cell may contain a different rule. A cell's rule table is encoded as a bit string (the "genome"), containing the next-state (output) bits for all possible neighborhood configurations (as in Figure 2.1). Rather than employ a population of evolving, uniform CAs, as with standard genetic algorithm approaches, our algorithm involves a single, non-uniform CA of size n, where the population of cell rules is initialized at random. Initial configurations are then generated at random, in accordance with the task at hand, and for each one the CA is run for M time steps. Each cell's fitness is accumulated over $C = 300$ initial configurations, where a single run's score is 1 if the cell is in the correct state after M iterations, and 0 otherwise. After every C configurations evolution of rules occurs by applying crossover and mutation. This evolutionary process is performed in a completely *local* manner, where genetic operators are applied only between directly connected cells. It is driven by $nf_i(c)$, the number of fitter neighbors of cell i after c configurations. The pseudo-code of the algorithm is delineated in Figure 5.2.

Crossover between two rules is performed by selecting at random (with uniform probability) a single crossover point and creating a new rule by combining the first rule's bit string before the crossover point with the second rule's bit string from this point onward.

Mutation is applied to the bit string of a rule with probability 0.001 per bit.

```
for each cell i in CA do in parallel
    initialize rule table of cell i
    f_i = 0 { fitness value }
end parallel for
c = 0 { initial configurations counter }
while not done do
    generate a random initial configuration
    run CA on initial configuration for M time steps
    for each cell i do in parallel
        if cell i is in the correct final state then
            f_i = f_i + 1
        end if
    end parallel for
    c = c + 1
    if c mod C = 0 then { evolve every C configurations}
        for each cell i do in parallel
            compute nf_i(c) { number of fitter neighbors }
            if nf_i(c) = 0 then rule i is left unchanged
            else if nf_i(c) = 1 then replace rule i with the fitter neighboring rule,
                                    followed by mutation
            else if nf_i(c) = 2 then replace rule i with the crossover of the two fitter
                                    neighboring rules, followed by mutation
            else if nf_i(c) > 2 then replace rule i with the crossover of two randomly
                                    chosen fitter neighboring rules, followed by mutation
                                    (this case can occur if the cellular neighborhood includes
                                    more than two cells)
            end if
            f_i = 0
        end parallel for
    end if
end while
```

Figure 5.2 Pseudo-code of the cellular programming algorithm.

There are two main differences between the cellular programming algorithm and the standard genetic algorithm approach (e.g., [132]): (a) The latter involves a population of evolving, uniform CAs; all CAs are *ranked* according to fitness, with crossover occurring between *any* two individuals in the population. Thus, while the CA runs in accordance with a local rule, evolution proceeds in a *global* manner. In contrast, the cellular programming algorithm proceeds *locally* in the sense that each cell has access only to its locale, not only during the run but also during the evolutionary phase, and no global fitness ranking is performed. (b) The standard genetic algorithm involves a population of *independent*

problem solutions; the CAs in the population are assigned fitness values independent of one another, and interact only through the genetic operators in order to produce the next generation. In contrast, our CA *coevolves* since each cell's fitness depends upon its evolving neighbors. This may also be considered a form of symbiotic cooperation, which falls, as does coevolution, under the general heading of "ecological" interactions (see Mitchell [131], pages 182-183). In summary, cellular programming is a local, coevolutionary, parallel genetic algorithm.

5.2 Statistical Measures for Cellular Evolutionary Algorithms

In this section, we present the statistical measures that we use in the next section for the analysis of cellular evolutionary algorithms. Most of these are rather intuitive, but do take into account the structural specificities of the parallel algorithm we consider. They give a good objective account of both the visual side, the phenotypic aspects, and the hidden side, the genotypic activity. Credits for these measures goes to Andrea Tettamanzi who formalized them.

5.2.1 Basic definitions and notation

Let us first formally define the basic elements used in this chapter (a summary of which is provided in Table 5.1). A *population* is a collection of individuals, each represented by a genotype. A genotype is not necessarily unique—it may occur several times in the population. In addition, if the population has a topology, the spatial distribution of the genotypes is of interest. Let n be the number of individuals in the system. Let R_i, $1 \leq i \leq n$ be the genome of the ith individual. Let Γ be the space of genotypes and $G(\Gamma)$ be the space of all possible populations. Let $f(\gamma)$ be the fitness of an individual having genotype $\gamma \in \Gamma$. When the cells are arranged in a row, as is the case in the example of Section 5.1.2, a population can be defined as a vector of n genotypes $x = (R_1, \ldots, R_n)$; then we have $G(\Gamma) = \Gamma^n$, provided the row of cells is not folded into a circle by connecting the extremity cells as is often done[2]. For example, with a three-bit genotype, the space of genotypes Γ is: {000, 001, 010, ..., 111}; for a population of size $n = 2$, $G(\Gamma)$ is: $\Gamma^2 = \{\{000, 000\}, \{000, 001\}, ..., \{111, 111\}\}$.

For all populations $x \in G(\Gamma)$, an occupancy function $n_x \colon \Gamma \to N$ is defined, such that, for all $\gamma \in \Gamma$, $n_x(\gamma)$ is the number of individuals in x sharing the same genotype γ, i.e., the occupancy number of γ in x. The size of population x, $\|x\|$, is defined as $\|x\| \equiv \sum_{\gamma \in \Gamma} n_x(\gamma)$.

We can now define a share function $q_x \colon \Gamma \to [0, 1]$ giving the fraction $q_x(\gamma)$ of individuals in x that have genotype γ, i.e., $q_x(\gamma) = n_x(\gamma)/\|x\|$.

Consider the probability space $(\Gamma, 2^\Gamma, \mu)$, where 2^Γ is the algebra of the parts of Γ and μ is any probability measure on Γ. Let us denote by $\tilde{\mu}$ the probability of generating a

[2]Actually, if we connect the border (a toroidal CA, for instance), then $G(\Gamma) = \frac{1}{n}\Sigma_{d|n}\phi(d)k^{n/d}$, where n is as defined above, d are the divisor of n, k is the cardinality of $\Gamma, |\Gamma|$, and ϕ is Euler's indicator, that is to say, $\phi(d)$ are all the postive number lesser than d and prime to d.

Γ	Space of genotypes
γ	Genotype in Γ
$f(\gamma)$	Fitness of genotype γ
$G(\Gamma)$	Space of populations
μ	Probability measure over Γ
$\tilde{\mu}$	Probability measure over $G(\Gamma)$
n	Population size
t	Generation
x	Population in $G(\Gamma)$
$n_x(\gamma)$	Occupancy number of genotype γ in population x
$q_x(\gamma)$	Share of genotype γ in population x
P	Probability measure over the trajectories
ϕ	Probability function over fitness
R_i	Genotype of ith individual in the population
X_t	(Random) population at tth generation
Ω	Space of all possible evolutionary trajectories
ω	Evolutionary trajectory

Table 5.1 Nomenclature.

population $x \in G(\Gamma)$ by extracting n genotypes from Γ according to measure μ. It can be shown that it is sufficient to know either of the two measures—μ (over the genotypes) or $\tilde{\mu}$ (over the populations)—in order to reconstruct the other.

The fitness function establishes a morphism from genotypes into real numbers. If genotypes are distributed over Γ according to a given probability measure μ, then their fitness will be distributed over the reals according to a probability measure ϕ obtained from μ by applying the same morphism. This can be summarized by the following diagram:

$$\begin{array}{ccc} \Gamma & \xrightarrow{f} & \mathbb{R} \\ \wr & & \wr \\ \mu & & \phi \end{array} \tag{5.1}$$

The probability $\phi(v)$ of a given fitness value $v \in [0, +\infty)$ is defined as the probability that an individual extracted from Γ according to measure μ has fitness v (or, if we think of fitness values as a continuous space, the probability density of fitness v): for all $v \in [0, +\infty)$, $\phi(v) = \mu(f^{-1}(v))$, where $f^{-1}(v) \equiv \{\gamma \in \Gamma : f(\gamma) = v\}$.

An evolutionary algorithm can be regarded as a time-discrete stochastic process

$$\{X_t(\omega)\}_{t=0,1,2,\ldots}, \tag{5.2}$$

having the probability space $(\Omega, \mathcal{F}, P)$ as its base space, $(G(\Gamma), 2^{G(\Gamma)})$ as its state space, and the natural numbers as the set of times, here called *generations*. Ω might be thought of as the set of all the evolutionary trajectories, $\mathcal{F}$ is a σ-algebra on Ω, and P is a probability measure over $\mathcal{F}$.

The transition function of the evolutionary process, in turn based on the definition of the genetic operators, defines a sequence of probability measures over the generations.

Let $\tilde{\mu}_t$ denote the probability measure on the state space at time t; for all populations $x \in G(\Gamma)$,

$$\tilde{\mu}_t(x) = P\{\omega \in \Omega : X_t(\omega) = x\}. \tag{5.3}$$

In the same way, let μ_t denote the probability measure on space $(\Gamma, 2^\Gamma)$ at time t; for all $\gamma \in \Gamma$,

$$\mu_t(\gamma) = P[\kappa = \gamma | \kappa \in X_t(\omega)]. \tag{5.4}$$

Similarly, we define the sequence of probability functions $\phi_t(\cdot)$ as follows: for all $v \in [0, +\infty)$ and $t \in N$,

$$\phi_t(v) = \mu_t(f^{-1}(v)). \tag{5.5}$$

5.2.2 The statistical measures

In this section we shall introduce several statistics pertaining to cellular evolutionary algorithms, which can be divided into two classes: genotypic statistics, which embody aspects related to the genotypes of individuals in a population, and phenotypic statistics, which concern properties of individual performance (fitness) for the problem at hand. Table 5.2 provides a summary of our statistics.

Notation	Formula	Explanation
$n_x(\gamma)$		Occupancy number of γ in population x
$q_x(\gamma)$	$n_x(\gamma)/\|\|x\|\|$	Share of γ in population x
$\nu(x)$	$\frac{\sum_{i=1}^{n}\sum_{j\in N(i)}[R_i \neq R_j]}{\sum_{i=1}^{n}\|\|N(i)\|\|}$	Frequency of transitions in population x
$H(x)$	$\sum_{\gamma\in\Gamma} q_x(\gamma)\log\frac{1}{q_x(\gamma)}$	Entropy of population x
$D(x)$	$\frac{n}{n-1}\sum_{\gamma\in\Gamma} q_x(\gamma)(1-q_x(\gamma))$	Genotypic diversity of population x
$E[\phi_x]$		Performance of population x
$\sigma^2(x)$	$\mathrm{Var}[\phi_x]$	Phenotypic diversity of population x
$\rho^2(x)$	$\frac{1}{n}\sum_{i=1}^{n}\left[1-\frac{1+\|\|N(i)\|\|f(R_i)}{1+\sum_{j\in N(i)}f(R_j)}\right]^2$	Ruggedness of population x

Table 5.2 Summary of statistics introduced in this section.

Genotypic statistics

One important class of statistics consists of various genotypic diversity indices (within the population) whose definitions are based on the occupancy and share functions delineated below.

Occupancy and share functions

At any time $t \in N$, for all $\gamma \in \Gamma$, $n_{X_t}(\gamma)$ is a discrete random variable with binomial distribution

$$P[n_{X_t}(\gamma) = k] = \binom{n}{k} \mu_t(\gamma)^k [1 - \mu_t(\gamma)]^{n-k}; \tag{5.6}$$

thus, $E[n_{X_t}(\gamma)] = n\mu_t(\gamma)$ and $\mathrm{Var}[n_{X_t}(\gamma)] = n\mu_t(\gamma)[1 - \mu_t(\gamma)]$. The share function $q_{X_t}(\gamma)$ is perhaps more interesting, because it is an estimator of the probability measure $\mu_t(\gamma)$; its mean and variance can be calculated from those of $n_{X_t}(\gamma)$, yielding

$$E[q_{X_t}(\gamma)] = \mu_t(\gamma) \quad \text{and} \quad \mathrm{Var}[q_{X_t}(\gamma)] = \frac{\mu_t(\gamma)[1 - \mu_t(\gamma)]}{n}. \tag{5.7}$$

Structure

Statistics in this category measure properties of the population structure, that is, how individuals are spatially distributed; obviously, such statistics apply only to populations that have a spatial structure (e.g., Figure 5.1b).

Frequency of transitions:

The frequency of transitions $\nu(x)$ of a population x of n individuals (cells) is defined as the number of borders between homogeneous blocks of cells having the same genotype, divided by the number of distinct couples of adjacent cells. Another way of putting it is that $\nu(x)$ is the probability that two adjacent individuals (cells) have different genotypes, i.e., belong to two different blocks.

Formally, the frequency of transitions $\nu(x)$ can be expressed as

$$\nu(x) = \frac{\sum_{i=1}^{n} \sum_{j \in N(i)} [R_i \neq R_j]}{\sum_{i=1}^{n} \|N(i)\|}, \tag{5.8}$$

where $[P]$ denotes the indicator function of proposition P, and $N(i)$ is the neighborhood of cell i, i.e., the set of cells spatially adjacent to it. In the example studied in Section 5.1.2, we have a one-dimensional grid structure, whereby Equation 5.8 reduces to

$$\nu(x) = \frac{1}{n} \sum_{i=1}^{n} \left[R_i \neq R_{(i \mod n)+1} \right]. \tag{5.9}$$

Diversity

There are a number of conceivable ways to measure genotypic diversity, two of which we define below: population entropy, and the probability that two individuals in the population have different genotypes.

Entropy:

The (bit) entropy of a population x of size n is defined as

$$H(x) = \sum_{\gamma \in \Gamma} q_x(\gamma) \log \frac{1}{q_x(\gamma)}. \tag{5.10}$$

Entropy takes on values in the interval $[0, \log n]$ and attains its maximum, $H(x) = \log n$, when x comprises n different genotypes.

Diversity indices:

The probability that two individuals randomly chosen from x have different genotypes is denoted by $D(x)$.

Index $D(X_t)$ is an estimator of quantity

$$\sum_{\gamma\in\Gamma} \mu_t(\gamma)\,(1-\mu_t(\gamma)) = 1 - \sum_{\gamma\in\Gamma} \mu_t(\gamma)^2, \tag{5.11}$$

which relates to the "breadth" of measure μ_t.

Proposition 1 *Let x be a population of n individuals with genotypes in Γ. Then,*

$$D(x) = \frac{n}{n-1} \sum_{\gamma\in\Gamma} q_x(\gamma)(1-q_x(\gamma)). \tag{5.12}$$

Proof: We choose the first individual at random with uniform probability: it will have genotype γ with probability $q_x(\gamma)$. We then choose a second individual without replacement (i.e., from the remaining $n-1$ individuals). This time, the probability that it will have genotype γ is $\frac{n_x(\gamma)-1}{n-1}$, given that the first individual has genotype γ. Therefore, the probability that it will *not* have genotype γ is

$$1 - \frac{n_x(\gamma)-1}{n-1} = \frac{n-n_x(\gamma)}{n-1}. \tag{5.13}$$

Now, we have to sum up this probability over all possible genotypes that can be extracted first, weighted by the probability of extracting them, which is q_x:

$$D(x) = \sum_{\gamma\in\Gamma} q_x(\gamma) \frac{n-n_x(\gamma)}{n-1}. \tag{5.14}$$

Observe that

$$\frac{n-n_x(\gamma)}{n-1} = \frac{n-nq_x(\gamma)}{n-1} = \frac{n}{n-1}(1-q_x(\gamma)). \tag{5.15}$$

Substituting in Equation 5.14 yields the thesis. QED

We observe that for all populations $x \in G(\Gamma)$,

$$D(x) \geq \frac{H(x)}{\log n}. \tag{5.16}$$

In other words, $D(x)$ rises more steeply than entropy as diversity increases.

An interesting relationship between D and ν is given by the following proposition.

Proposition 2 *Given a random one-dimensional linear population x of size n, the expected frequency of transitions will be given by*

$$E[\nu(x)] = D(x). \tag{5.17}$$

Proof: We express the expected frequency of transitions:

$$E[\nu(x)] = \frac{1}{n} \sum_{i=1}^{n} P[\gamma_i \neq \gamma_{(i \mod n)+1}] = \frac{1}{n} \sum_{i=1}^{n} D(x) = D(x). \tag{5.18}$$

(Note that this can also be proven for dimensions higher than one by generalizing the above proof.) QED

Phenotypic statistics

Phenotypic statistics deal with properties of phenotypes, which means, primarily, fitness. Associated with a population x of individuals, there is a fitness distribution. We will denote by ϕ_x its (discrete) probability function.

Performance

The performance of population x is defined as its average fitness, or the expected fitness of an individual randomly extracted from x, $E[\phi_x]$.

Diversity The most straightforward measure of phenotypic diversity of a population x is the variance of its fitness distribution, $\sigma^2(x) = \mathrm{Var}[\phi_x]$.

Structure

Statistics in this category measure how fitness is spatially distributed across the individuals in a population.

Ruggedness:

Ruggedness measures the dependency of an individual's fitness on its neighbors' fitness (a sort of phenotypic entropy). Ruggedness of population $x \in G(\Gamma)$ of size n can be defined as follows:

$$\rho^2(x) = \frac{1}{n}\sum_{i=1}^{n}\left[1 - \frac{1 + \|N(i)\| f(R_i)}{1 + \sum_{j\in N(i)} f(R_j)}\right]^2. \tag{5.19}$$

Notice that $\rho^2(x)$ is independent of the fitness magnitude in population x, i.e., of performance $E[\phi_x]$.

For a one-dimensional population of size n, Equation 5.19 becomes:

$$\rho^2(x) = \frac{1}{n}\sum_{i=1}^{n}\left[1 - \frac{1 + 2f(R_i)}{1 + f(R_{(i \mod n)+1}) + f(R_{(i-2 \mod n)+1})}\right]^2. \tag{5.20}$$

The computational tasks

In the next section we study the cellular programming algorithm on the three computational tasks presented in chapter 4, in section 4.2: density, synchronization, and random number generation (RNG). We use thus one-dimensional, 2-state, $r = 1$ cellular automata.

5.3 Results and analysis

Using the different measures presented in Section 5.2.2, we analyzed the processes taking place during the execution of the cellular programming algorithm presented in the previous section. This was carried out for all three tasks delineated in 4.2 (density, synchronization, and RNG). From previous experiments, we knew that they present different degrees of difficulty for the evolutionary algorithm, with density being the hardest and the other two being easier (though non-trivial [179, 182]). All experiments were run for CAs of size 150. The number of experiments varied with the tasks: 100 for RNG, 96 for density, and 103 for synchronization.

First, we present features commonly observed for all tasks. We then compare these results with those obtained via a control task to distinguish what may be properties intrinsic

to the algorithm itself from idiosyncrasies common to the three tasks studied. Finally, the specific properties of each tasks are described in the subsequent subsections.

5.3.1 Common features

There are a number of trends found to be common to all three tasks. A notable feature we observed is the significant decrease in entropy (H) which is initially naturally high, due to the population randomness. This can be considered as an increase in order, which is also reflected in the trends exhibited by other measures, described below.

In all runs the entropy falls from a high of approximately 0.8 to a low of approximately 0.7 within the first 20 generations, and from then on generally tends to decline. Though this latter decline is not monotonous, for all three tasks—whatever the outcome, successful or unsuccessful—the entropy ends up below 0.5. This fall in entropy is due to two factors. First, we can observe in all runs a steep drop in the transition frequency (ν) in the first few generations, followed by an almost continuous drop in the subsequent generations, whose slope is task dependent (see Figure 5.3). Though it may be intuitive that, given the possibility of rule replication between neighboring cells after each generation, blocks will tend to form, our measures now provide us with quantitative evidence. Note that the transition frequency (ν) progresses towards an oscillatory state about values below 0.3 (including the synchronization case, though this occurs at a later time—around generation 160—not shown in Figure 5.3). The second factor involved in the lower entropy is the number of rules. One can see directly that a low ν implies few rules. This is corroborated by the diversity (D) measure decreasing trend.

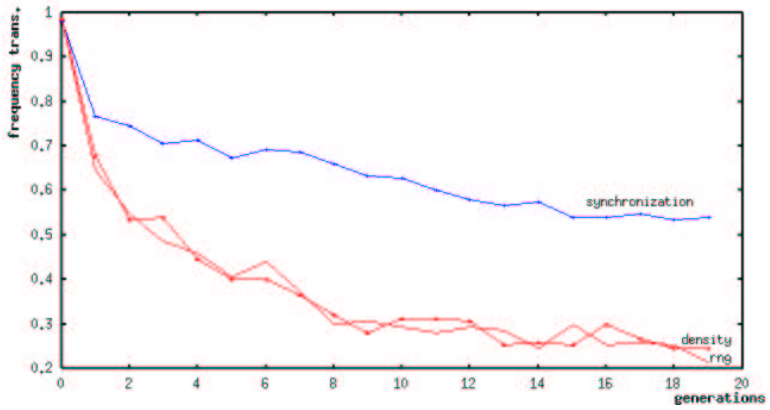


Figure 5.3 Progression of the transition frequency (ν) over the first 20 generations of typical runs.

For the three tasks studied herein the objective is to reach a high average fitness over the entire population, rather than consider just the highest-fitness individual cell. Thus, intuitively we can expect that the phenotypic variance will tend to be minimized, and we can factually check that both the fitness variance (σ^2) and ruggedness (ρ^2) are always very low towards the end of an evolutionary run. A clear correlation between genotypic and

phenotypic measures we observed is that a higher fitness variance results in a much steeper decrease of ν in the first steps when it is very high, thereby rendering the grid more uniform (see Figure 5.3). This trend in the very first steps is an indication on the subsequent dynamics of the run. We will see that the normalized entropy is lower for RNG and Density classification than for the synchronization task. Usually the evolved CA had less than 10 different rules out of the 256 possible ones. We found that fitness variance (σ^2) is about ten times higher for RNG and density than for synchronization, which underlies the difference in the declivity of the ν curves in Figure 5.3.

In conclusion, there is an increase in order, meaning that the population tends to evolve towards a small number of rules organized in blocks, i.e., a low H and ν. As we will see in the following subsections, which detail properties specific to each task, the exact values of H, D, and ν differ. Nonetheless, the general tendencies prevail.

5.3.2 The control task

To be able to distinguish peculiarities of the evolutionary runs due to the tasks from the specificities of the co-evolutionary algorithm itself, we propose in this section to study the dynamics of the algorithm on a control task. To define a control task, we have to remember the purposes of this statistical work. There are two of them: to understand the inner workings of the algorithm (genotypic statistics) and to correlate what happens at the genotypic and phenotypic levels. Thus we need a control task where the phenotypic behavior is not related to the genotypic workings of our algorithm. This may seems paradoxical as the very principle of evolutionary algorithms is to establish this link between genotype and phenotype. So to do that we have to fool the algorithm. There are two ways to accomplish this. Either affect the workings of the cellular automata to disconnect the global behavior from the local rule, the genome, or affect the "vision" of the algorithm, namely the fitness function. It is clear that if we attribute a uniformly random fitness to each cell, then we get the desired gap between the genotypic changes and the phenotypic behavior.

As we can observe clearly in Figure 5.4, the general trend of ordering seems to be inherent to the algorithm. D, H and ν first fall down sharply and then level off to a limit value as the average curves show. More precisely, ν exhibit the most significant decrease, thereby implying the rapid formation of blocks. However, if this implies a diminution of the number of rules, the relatively high stabilization of the diversity measure (as compared to the other tasks) demonstrates the liveliness of a certain variety of rules present. The fact that diversity remains high also shows that the repartition of the rules is quite uniform, i.e., there is no block taking over. Though one would expect such a behavior, given the nature of the task, interestingly enough, it proves that the tendency to form blocks is not a tendency to total uniformity in general. This underlines the properties of over-ordering of both RNG and density classification. It is not directly observable in Figure 5.4, but the first twenty generations of the control task confirm our hypothesis that a high fitness variance at the beginning of the evolutionary runs induces a rapid decrease in the transition frequency. Nevertheless, the very random nature of the control task maintains a rather high diversity and entropy, unlike, as we will see, the case for RNG and diversity. Let's now concentrate

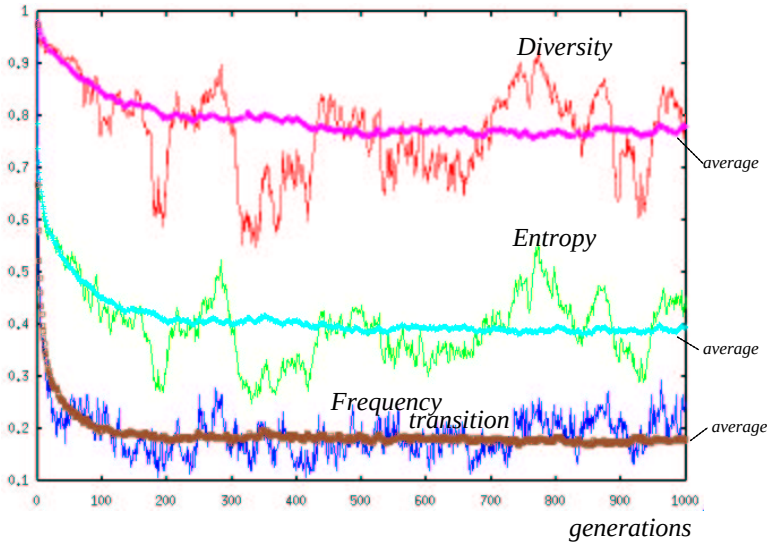


Figure 5.4 Control task: we can see here the diversity $D(x)$, the entropy $H(x)$, and the frequency of transitions $\nu(x)$ for a "typical" run of the cellular programming algorithm on the control task. What appears here as thick lines are the values of $D(x)$, $H(x)$, and $\nu(x)$ averaged over 100 runs of the evolutionary algorithm.

on each of the tasks studied.

5.3.3 Random number generation (RNG)

The evolution of random number generators is highly successful in terms of final fitness. Though high fitness does not necessarily entail good random behavior, Sipper and Tomassini [186, 187] have shown that such evolved CAs do fare well on several randomness tests. For all runs, we obtained normalized fitness values higher than 0.99, in relatively few generations (approximately 100).

One can note that there is a steady increase in fitness during the first 60 generations, accompanied by a relatively sharp descent in entropy (H) which drops from a high of 0.8 to a low of approximately 0.3. We see in Figure 5.5 that the fitness average is already high (approximately 0.98) when the entropy reaches the low-value range. We then observe that the entropy stabilizes, without stopping its descent, and the average fitness remains generally high. During the stabilization phase the genotypic "variance" (ν) declines to almost 0; indeed, the evolved CA usually contained less than 10 different rules, with three or four rules eventually covering almost all the CA grid. This stabilization is not a leveling off like for the control task. Diversity D actually keeps on its descent at about the same rate all along the run. Given that the transition frequency remains almost stable during after the initial steep fall, this demonstrates that a block, or a few blocks of rule are taking over the CA. It finally translates into an eventual diminution of ν. Hence, the cellular algorithm while maintaining the fitness high tends to simplify the "global" genome. A property rather uncommon in evolutionary algorithms.

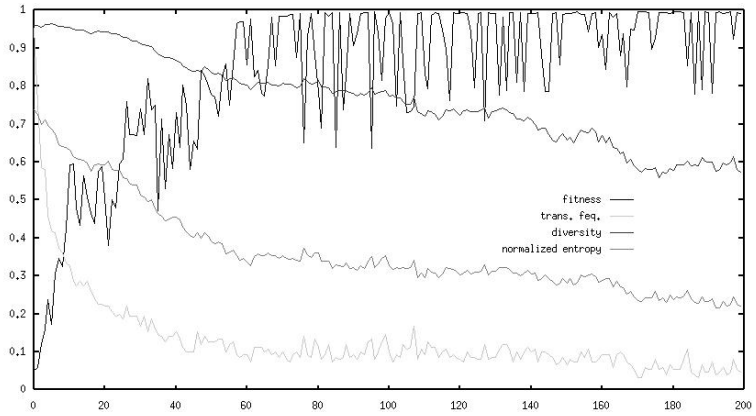


Figure 5.5 Fitness and entropy (H) vs. time for a typical run of the RNG task. We observe that D, H and ν decline rapidly while fitness increases, and then keep on their descent but at a much slower pace.

5.3.4 Synchronization

The evolutionary dynamics of the synchronization task were found to exhibit at most four fitness phases: a low-fitness phase, followed by a rapid-increase phase, continuing with a high-fitness phase and ending with a medium high, stable fitness phase. Note that for this task a successful run is considered to be one where a perfect fitness value of 1.0 is attained. The evolutionary runs can be classified into six distinct classes. Two of the classes represent successful runs (Figures 5.6a and 5.6b), and the other four represent unsuccessful runs (Figures 5.7a, 5.7b, 5.7c and 5.7d). This classification is based on the number of phases exhibited during the evolution. We should note that these six classes do not happen with equal probabilities. To give a rough idea of this distribution, we may say that we reach a perfect solution about two-third of the times. Hence the four type of unsuccessful runs represent only one third of the runs.. We now present the results of our experiments according to these four fitness phases.

Phase I: Low fitness. This phase is characterized by an average fitness of 0.5, with an extremely low variance. However, while exhibiting phenotypic (fitness) "calmness," this phase is marked by high underlying genotypic activity: the entropy (H) steadily decreases, and the number of rules strongly diminishes. An unsuccessful type-a run (Figure 5.7a) results from "genotypic failure" in this phase. To explain this, let us first note that for the synchronization task, only rules with neighborhood 111 mapped to 0 and 000 mapped to 1 (cf. Figure 2.1) may appear in a successful solution. Let us call this the "good" quadrant of the rule space, and define the "bad" quadrant to be the one that comprises rules mapping 111 to 1 and 000 to 0. In some experiments, evolution falls into the bad quadrant, possibly due to a low fitness variance. Only the mutation operator can possibly hoist the evolutionary process out of this trap. However, it is usually insufficient in itself, at least with the mutation rate used herein. Thus, in such a case the algorithm is stuck in a local minimum, and fitness never ascends beyond

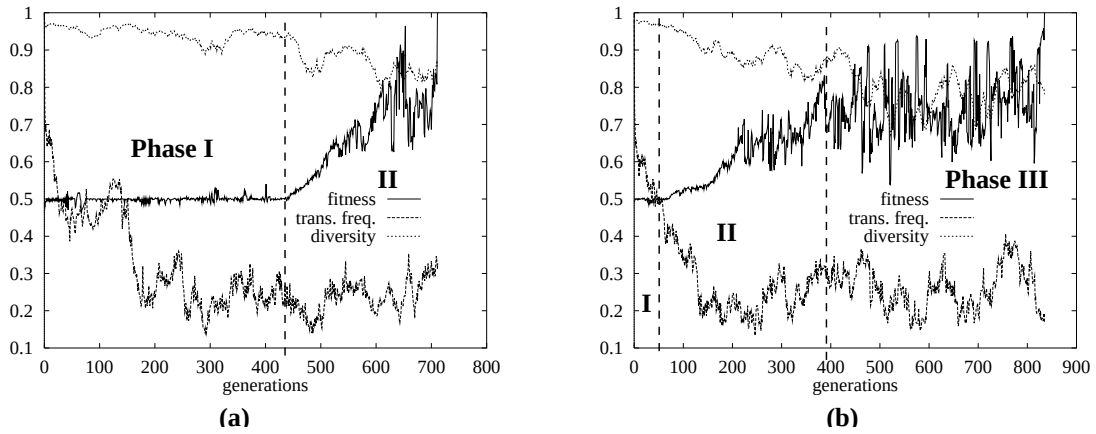


Figure 5.6 The evolutionary runs for the synchronization task can be classified into six distinct classes, based on the four observed fitness phases: phase I (low fitness), phase II (rapid fitness increase), phase III (high fitness) and phase IV (medium high fitness). Here we present the two classes of successful runs. (a) Successful run, exhibiting but the first two phases. The solution is found at the end of phase II. (b) Successful run, exhibiting three phases. The solution is found at the end of phase III.

0.53 (Figure 5.7a). We may surmise that this fall in the bad quadrant prevents any block of rules to attain good fitness. Hence, as we see in the Figure, diversity remains really high while transition frequency settles at about the same rate as for the control task. This is intuitively sound as this fall in the bad quadrant is "almost" equivalent to random fitness.

Phase II: Rapid fitness increase. A rapid increase of fitness characterizes this phase, its onset marked by the attainment of a 0.54 fitness value (at least). This comes about when a sufficiently large block of rules from the good quadrant emerges through the evolutionary process. In a relatively short time after this emergence (less than 100 generations), evolved rules over the entire grid all end up in the good quadrant of the rule space; this is coupled with a high fitness variance (σ^2). This variance then drops, while the average fitness steadily increases, reaching a value of 0.8 at the end of this phase. Entropy never stabilizes before the end of this task. While transition frequency always settle down after 200 to 400 generations, whatever the fitness outcome. On certain runs a perfect CA was found directly at the end of this stage, thus bringing the evolutionary process to a successful conclusion (Figure 5.6a).

Phase III: High fitness. The transition from phase II to phase III is not clear cut, but we observed that when a fitness of approximately 0.82 is reached, the fitness average then begins to oscillate between 0.65 and 0.99. During this phase the fitness variance also oscillates between approximately 0 and 0.3. While low, this variance is still higher than that of phase I. Whereas in phases I and II we observed a clear decreasing trend for entropy (H), in this phase entropy exhibits an oscillatory pattern between values of approximately 0.3 and 0.5. We conclude that when order (low entropy) is too

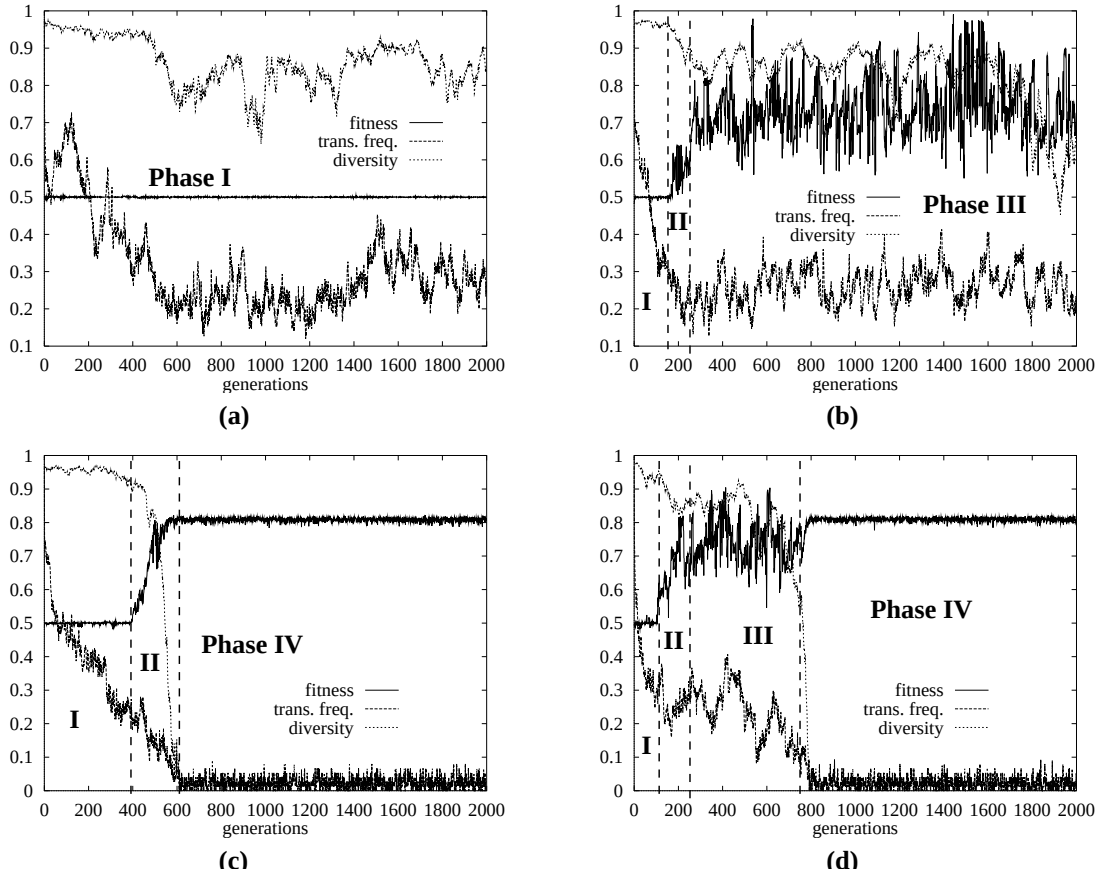


Figure 5.7 We can see here the four kind of unsuccessful evolutionary runs for the synchronization task out of the six distinct classes, based on the four observed fitness phases: phase I (low fitness), phase II (rapid fitness increase), phase III (high fitness) and phase IV (medium high fitness). Here we present the four types of unsuccessful runs. (a) Unsuccessful run, "stuck" in phase I. (b) Unsuccessful run, exhibiting three phases. Phase III does not give rise to a perfect solution. (c) Unsuccessful run, exhibiting three phases: I, II, IV. Falling into the Phase IV trap directly from phase II. (d) Unsuccessful run, exhibiting all four phases. Falling into the phase IV trap after reaching phase III. A block of 127 takeover.

high, disorder is reinjected into the evolutionary process, while remaining in the good quadrant of the rule space; hence the oscillatory behavior. This is quite particular of this task. As we saw, for RNG entropy steadily decreased to value around 0.25. and as we will see, this oscillatory behavior of H is in-between much lower values for the density task. In fact, the synchronization behavior is close to the control task behavior (from an entropy point of view). On certain runs it took several hundred generations in this phase to evolve a perfect CA—this is a success of type b (Figure 5.6b). Finally, on other runs no perfect CA was found, though phase III was reached and very high fitness was attained. This is a type-b unsuccessful run (Figure 5.7b) which does not differ significantly from type-b successful runs.

Phase IV: Stable medium high fitness. This phase may appear after phase II (Fig-

ure 5.7c) or phase III (Figure 5.7d). It is characterized by an extremely low fitness variance, even lower than phase I, as it reaches 0 sometimes and an average global fitness of 0.8. But what marks unmistakably phase IV is a *complete* drop of transition frequency to reach values close to zero and, thus, a very *sharp* drop of diversity and entropy to also attain values close to zero. This can only be explained by one block taking over or almost taking over all the grid, and, this is the case. Just as for type-a unsuccessful runs we fall here into an evolutionary trap. What happens is that the uniform cellular automata with rule 127[3] do fare well on our fitness criteria (0.8). No single rule can perturbate the global efficiency of a 127 rule block (see Figure 5.8). Thus if a mutation occurs, it will always be wiped out after one generation. Then obviously as cross-over is of no help against a block of 127 cells, the trap is perfect and the algorithm never gets out of it.

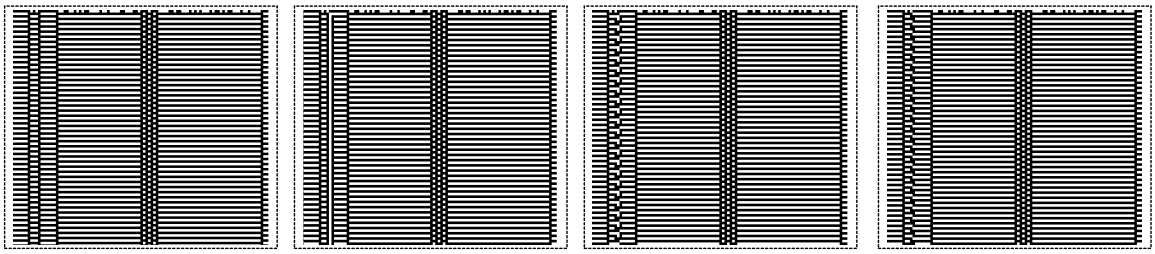


Figure 5.8 This figure demonstrates the efficiency of rule 127 for the synchronization task and its insensitivity to perturbation. The rightmost space time diagram (time flowing downward) illustrates rule 127, on a random initial configuration. The other diagrams show rule 127 containing a different rule at cell 10 (0 being the rightmost cell) on the same initial configuration.

5.3.5 Density

Density is arguably the most difficult task of the three. Indeed, this task necessitated the longest evolution times, with fitness never exceeding 0.95. Thus, following Sipper [179], we define a successful density CA to be one with average fitness higher than 0.9.

One can distinguish three fitness phases: a rapid increase phase (I), a stable phase (II) and an unstable phase(III). Note that they are different from the synchronization task phases. These phases also differ from those identified by Mitchell, Crutchfield, and Hraber [132], when applying a genetic algorithm to the evolution of uniform CAs that solve the density task (briefly, they observed four phases, which they dubbed "epochs of innovation"; the difference between the dynamics of their algorithm and ours is not altogether surprising, in view of the very different natures of the two). Phase I occurs in every run from the first generation onward. It may be followed by phase II or III or a "mix" of the two. These dynamics describe three main classes of runs: type-a exhibiting phase I and II

[3] in Wolfram notation

only, type-b exhibiting phase I and III only and finally type-c exhibiting all three phases (see respectively Figure 5.9a, 5.9b and 5.9c). The last two types may be successful, by the criteria stated earlier, while the type-a runs are always unsuccessful.

Phase I: Rapid fitness increase. During this phase the frequency transition (ν) and the entropy (H) rapidly decrease, approximately, from 0.8 and 0.98 respectively to about 0.2 and 0.4 respectively within the first 10 to 20 generations, while average fitness increases to approximately 0.8. Diversity (D) decreases also during this phase from about 0.98 to 0.8. With this task, rules that are from the "bad" quadrant can nonetheless exhibit high fitness, between approximately 0.75 and 0.85 (here the "bad" quadrant of the rule space is the one in which the genome maps 111 to 0 and 000 to 1). This presents an obstacle to evolution, possibly leading to a local minimum, where only the mutation operator may be of help (as is phase II). However this may be the reason why "evolution" is instantaneous. Fitness rises from the very first step and onward until it reaches about 0.8. (see Figure 5.10a, b and c). There is no warning in this phase to know if we are going to fall in phase II or in phase III.

Phase II: Fitness stabilization. After this initial burst of high evolutionary activity in phase I, the algorithm either enters phase II or phase III. In phase II, the average fitness levels off and begins to oscillate in a narrow band of values mostly between 0.8 and 0.85. Actually, phase II comes in two flavors. The first one, which is typical of type-a run (see Figure 5.9a), is characterized by an almost nil entropy, frequency of transitions and most notably an almost nil diversity. Both H and ν oscillate between 0 and 0.1 while D is generally below 0.2 with some rare bursts above. This first kind of phase II is also marked by a relatively low fitness variance. The second type of phase II exhibits the same fitness an equivalently low fitness variance. However, as may be seen in Figure 5.9c, though it is distinguished by very low values of D particularly, but also H and ν, much lower than in phase III, these are not as low as in the first kind of phase II. This shows that in this second kind of phase II there is still non negligible genotypic activity, which often leads this kind of phase to develop into phase III. However, if the entropy falls below 0.1 durably, then it turns into the first type of phase II and the run is bound to fail. So what is this phase II really ? Actually, it matches a large block of 127 cells. This may seem strange as it is a rule in the "bad" quadrant. However, as we are measuring fitness on a single step, and as rule 127 tends after one step to synchronize on the local majority side, then when we take our measure after an even number of steps, it naturally fares well (about 0.8 on average). The difference between the first and the second kind of phase II is the size of the block of rules 127. Usually in the second kind, there is a block of another rule, creating enough perturbations to finally overcome the block of 127, while in the first kind, the block is so large, that given the very high stability of rule 127 (there is no information transfer, no particle in this rule), it turns out to be an evolutionary trap, just like the phase IV in the synchronization task.

Phase III: Unstable fitness. This phase, the onset of which is the hallmark of successful runs, is characterized by the destabilization of the average fitness (see Figure 5.9b

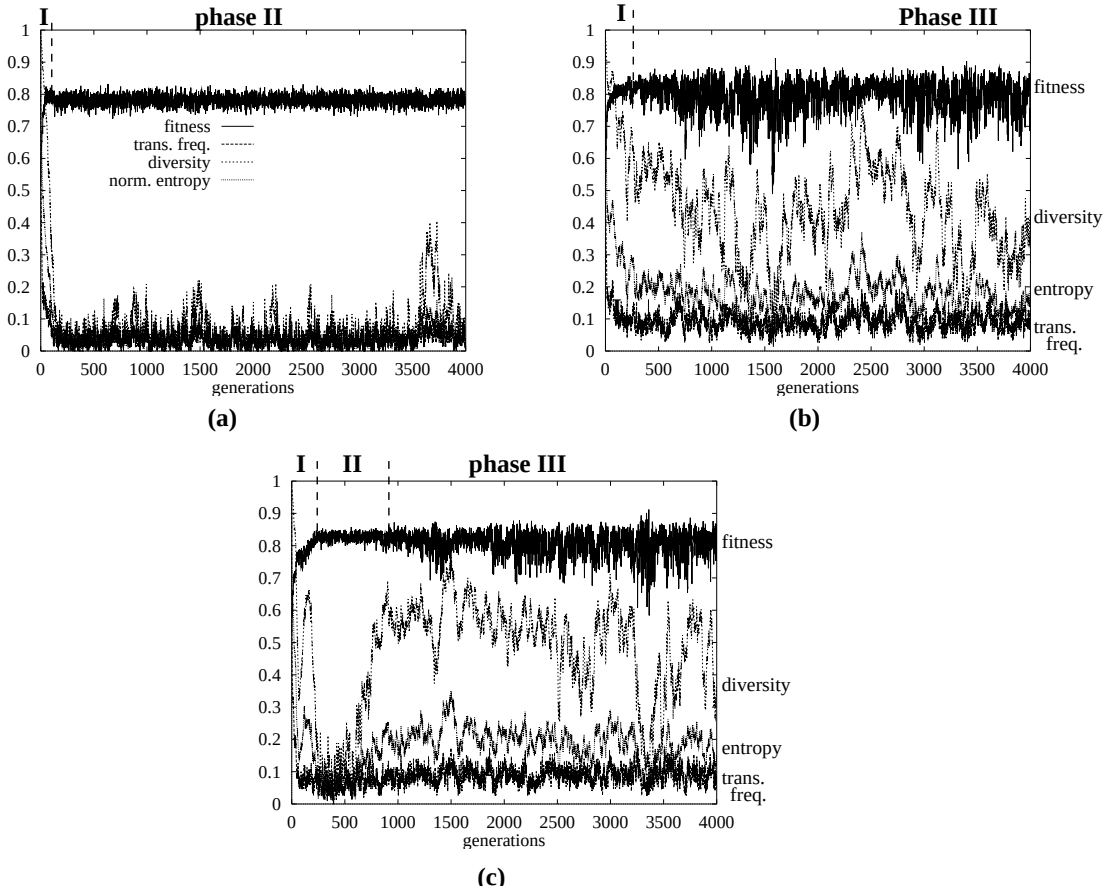


Figure 5.9 The evolutionary runs for the density task can be classified into three distinct classes, based on the three observed fitness phases: phase I (rapid fitness increase), phase II (fitness stabilization), and phase III (unstable fitness). (a) Unsuccessful run, exhibiting only the first two phases. (b) Successful run, exhibiting only phase I and III. (c) Successful run, exhibiting the three phases. The solution is found during phase III. (Note that phase II for a successful run exhibits higher H, D and ν values than that of its counterpart in an unsuccessful run; however, phase III can still be readily distinguished by a net increase in these values.)

and c). It exhibits highly oscillating fitness behavior, with values possibly differing by as much as 0.3 (between approximately 0.65 and 0.95) from one generation to the next. This suggests that the fitness landscape is highly rugged, which may explain the relative difficulty of the task. In this phase diversity (D) is really high compared to phase II. However, we may note that the values between 0.2 and 0.7 are quite below that of any RNG or synchronization runs, or for the control task. This tendency to a low diversity and entropy, a tendency to order, explains the relatively high occurrences of phase II, an over-ordering phase.

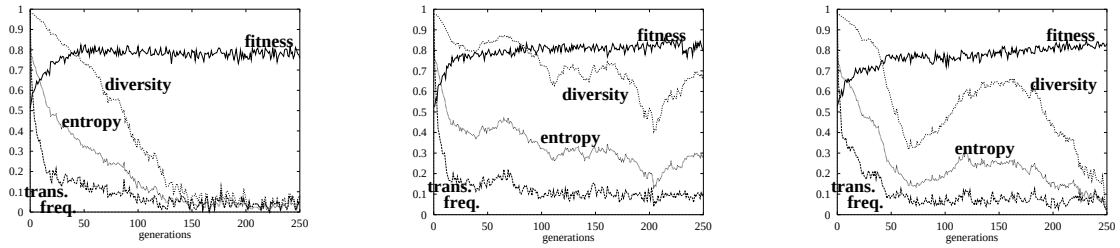


Figure 5.10 The first 250 generations of the evolutionary runs for the density task presented in Figure 5.9. (a) We can see clearly in this zoom, the very low values of entropy, diversity and transition frequency characterizing the first kind of phase II, typical of type-a unsuccessful runs. (b) Relative high values for phase for type-b successful runs. (c) Successful run, exhibiting the low values of H, D and ν characterizing the second kind of phase II at the very end of this graph, around generation 250. These values are still higher than in (a).

5.4 Concluding Remarks

In this chapter we demonstrated the application of several statistical measures of interest, both at the genotypic and phenotypic levels, which are useful for analyzing the workings of fine-grained parallel evolutionary algorithms in general. The cellular programming evolutionary algorithm, which we employed to evolve solutions to three different problems: density, synchronization, and random number generation, revealed more of its workings through these statistics.

We observed a number of features common to all three tasks, most of which were confirmed on the control task. They may therefore represent inherent algorithmic properties. First and foremost among these concerns the notable difference between activity at the genotypic level and at the phenotypic level. At several stages of the evolutionary process, though seemingly little phenotypic (observable) activity is taking place, the population is in fact teeming with underlying genotypic activity. This latter gives rise at certain points in time to discernible phenotypic effects, evident by fitness phase transitions. This may be an evidence that our fitness criteria are not fine enough, at least in the early stage. Besides, and more importantly, we noted a strong decrease in diversity (D), transition frequency (ν), entropy (H), and number of rules. This provides quantitative evidence that the population tends to evolve towards a more ordered state with respect to the initial state—which is random and therefore highly disordered. As we saw, there is a risk of over-ordering which translates usually into fitness traps. This may be an argument to introduce a genotypic measure in the fitness, to counter this problem.

Each of the tasks also exhibited a number of specific properties. For the RNG task a high-fitness solution was always evolved, in usually no more than 100 generations with a relatively high order. The synchronization task was seen to undergo (at most) four fitness phases: a low-fitness phase, followed by a rapid increase in fitness, continued with a high-fitness phase and ended by a high fitness evolutionary trap. The nature of these stages,

or the absence of some of them, served to distinguish between six types of evolutionary runs. Finally, the density task was seen to undergo (at most) three evolutionary phases: a rapid-increase phase, followed by a highly stable phase, ending with an unstable phase. A successful solution was observed to emerge only during the third phase (which was not necessarily reached, e.g., in certain failed runs). For these last two tasks, the genotypic measures gave a good insight of both traps, and a good, albeit not a perfect one, to predict the outcome of the runs.

Parallel evolutionary algorithms have been receiving increased attention in recent years. Gaining a better understanding of their workings and of their underlying mechanisms thus presents an important research challenge. We hope that the work presented in this chapter represents a small step in this direction.

Mais en disant la personne, il faut bien comprendre que je n'entends point cette rencontre miraculeuse et vaine de cellules auxquelles la vie impose sa silencieuse unité – j'entends plutôt cette unité spirituelle qui au-delà des apparences concrètes ou psychiques n'atteint sa véritable raison d'être qu'au moment même où elle consent à se perdre dépouillée des passions trop humaines[1].

Henri Féraud, *Le Génie d'oc et l'homme méditerranéen. [48]*

Chapter 6

From Chaos to Order

6.1 Introduction

Though life was an awe-inspiring model for the first Cellular Automata designers, simplification was a prime constraint in all these studies, not only for obvious practical reasons, but also as a guide to extract the quintessential ideas behind self-replication. Nevertheless, I believe that an ever present quality was omitted for the former reasons than the latter choice: *robustness*. Robustness to faults and robustness to asynchronous dynamics. In this chapter, I propose simple and practical models of robust CAs.

In the first part of this chapter, section 6.2 and section 6.3, I will concentrate on the asynchrony problem. Obviously in CA, this may be regarded as a particular case of fault-resistance. But providing a specific view of this part of the problem, however, allows us to develop perfect or economic solutions. In that section I will first study the (im)possibility of evolving asynchronous binary CA, and then demonstrate the wealth of redundant CAs through the exposition of: 1)perfect, 2)imperfect but economic, and 3)evolved methods.

In section 6.4, I will tackle the more general problem of fault-resistance, which could be renamed as non-deterministic CA. The goal could then be expressed as designing deterministic CA in a nondeterministic environment. The solution I propose is general but is not easily adaptable to any CA. It relies on error correcting code and presents the advantage compared to existing methods of requiring no complexation of the inner CA workings or extra space or memory. As an illustration, I will propose some fault-resistant structures towards the creation of a fault-resistant self-replicating loop.

Both these approaches will be the occasion, in the concluding section, to illustrate the crucial question of the information stored in a CA.

[1] But when I say *the person*, we have to understand not this miraculous and vain encounter of cells upon which life imposes its silent unity — but rather this spiritual unity which beyond its apparences, concrete or psychical, reaches its real *raison d'être* only when it concedes to lose itself pared down of too human passions.

6.2 Asynchronous Cellular Automata

One of the prominent features of the CA model is its synchronous mode of operation, meaning that all cells are updated simultaneously. But this feature is far from being realistic from a biological point of view as well as from a computational point of view. As for the former, it is quite evident that there is no accurate global synchronization in nature. As for the latter, it turns out to be impossible to maintain a large network of automata globally synchronous in practice. Besides, for our interest in this thesis which is concerned with problem-solving CAs, it is interesting to try to delineate what part of the computation relies on the synchrony constraint.

A preliminary study of asynchronous CAs, where one cell is updated at each time step, was carried out by Ingerson and Buwel,[92], where the different dynamical behavior of synchronous and asynchronous CAs was compared; they argued that some of the apparent self-organization of CAs is an artifact of the synchronization of the clocks. Wolfram, [226], noted that asynchronous updating makes it more difficult for information to propagate through the CA and that, furthermore, such CAs may be harder to analyze. Asynchronous CAs have also been discussed from a problem-solving and/or Artificial Life standpoint in Refs. [19, 79, 93, 143, 171, 178, 195]. All these works devoted to asynchronous cellular automata only concentrated on the study of the effects but not on correcting asynchrony or dealing with it.From a theoretical computer science perspective, Zielonka [234, 235] introduced the concept of asynchronous cellular automata. Though the question attracted quite some interest [33, 103, 153], the essential idea behind them was to prove that they were "equivalent" to synchronous CA in the sense that, for instance, they recognize the same trace language or could decide emptiness. From these two fields, we thus know that asynchronous CA are potentially as powerful as synchronous CA, computationally speaking[2], and, nevertheless, that most of the effects observed in the synchronous case are artifacts of the global clock. In this section, we thus propose to develop asynchronous CA exhibiting the same behavior as Asynchronous CA through both design and evolution. As Gacs[63, 64] reminded us, asynchrony may be considered a special case of fault-tolerance. However, if this consideration is nice in its generalization (i.e. a fault-tolerant CA is also asynchronous), it eschews a lot of potential optimization. As we will see later this cost is not only in terms of complexity and/or memory, but also in terms of efficiency. Effectively as we will see, a simple time stamping method allows *perfect* asynchronous behavior, whereas fault-tolerance is always bounded.

In section 6.2.1, I will present the evolution of binary asynchronous CAs. The results of the evolutionary runs lead me to consider redundant CA. I call a CA redundant, when the input, the initial configuration is coded on q states but the CAs work with k states, $k > q$. In section 6.3, I will study both designed and evolved solutions to asynchrony with such CAs, both perfect and imperfect, demonstrating that even perfect solutions may not be the best depending on the computational goal sought.

[2]In the traditional sense, not in the sense of visual computation introduced in chapter 4.

6.2.1 Evolution of non-uniform *binary* asynchronous CA

The issue investigated in this section is that of evolving non-uniform *binary* asynchronous CAs to perform the original density and the synchronization tasks. This is a particularly difficult task as no extra states are given to the CA to cope with the synchronization faults. Hence it should be pointed out that there is no extra cost for the solution presented here compared to the non-uniform synchronous case. However, the principal purpose of this task is to highlight the difficulty of the task, if we are not prepare to concede extra-cost to cope with the asynchronous environment. As we will see in the next section, a few extra bits give rise to very efficient solutions.

I will first present the different models of asynchrony for which solutions were sought, and then present briefly the results for the original, imperfect density task and the synchronization task, both presented in section 4.2.

The model: The grid is partitioned into *blocks* in which synchronous updating takes place (i.e., all cells within a block are updated simultaneously), while the blocks themselves are updated asynchronously (rather than have all blocks updated at once); thus, inter-block updating is synchronous while intra-block updating is asynchronous.[3] The number of blocks per grid, $\#_b$, is a tunable parameter, entailing a *scale* of asynchrony, ranging from complete synchrony ($\#_b = 1$) to complete asynchrony ($\#_b = N$). The scale of asynchrony is further refined by three models of inter-block synchronization (intra-block updating is always synchronous). These three models are:

Model 1 At every time step each block is updated *independently* of the others with probability p_f, chosen so as to insure that at least one block is updated per time step with probability ≥ 0.99.

Model 2 Each time step a different block is chosen at random without replacement, such that every $\#_b$ steps, *all* blocks are updated exactly once. We denote by *logical step* the succession of $\#_b$ time steps necessary for one full update cycle, in which all cells are updated (thus, one logical step is equivalent to one time step in the synchronous model, with respect to cell updating).

Model 3 All blocks are updated in a fixed, random order every logical step. This is similar to the second model, in that each cell is guaranteed to have updated its state every logical step, however, the (random) update order is fixed (rather than selected anew each logical step). Note that though the update order is deterministic, this model is interesting in that cells are not updated in a regular manner; neighboring cells may be updated at different points in time, which renders the computation more difficult.

Perfect cyclic behavior cannot arise in the first model, since the notion of a logical step, i.e., a fixed number of time steps after which all cells will have been updated, does not exist; however, a fixed point, such as that desired for the density problem, can be attained.

[3] A preliminary investigation of a CA-derived model based on the "blocks" idea was carried out in [178].

Models 2 and 3 can be applied to the synchronization problem since cyclic behavior may be attained, if one considers the CA's configuration every logical step, i.e., the alternation between all 0s and all 1s takes place every $\#_b$ time steps.

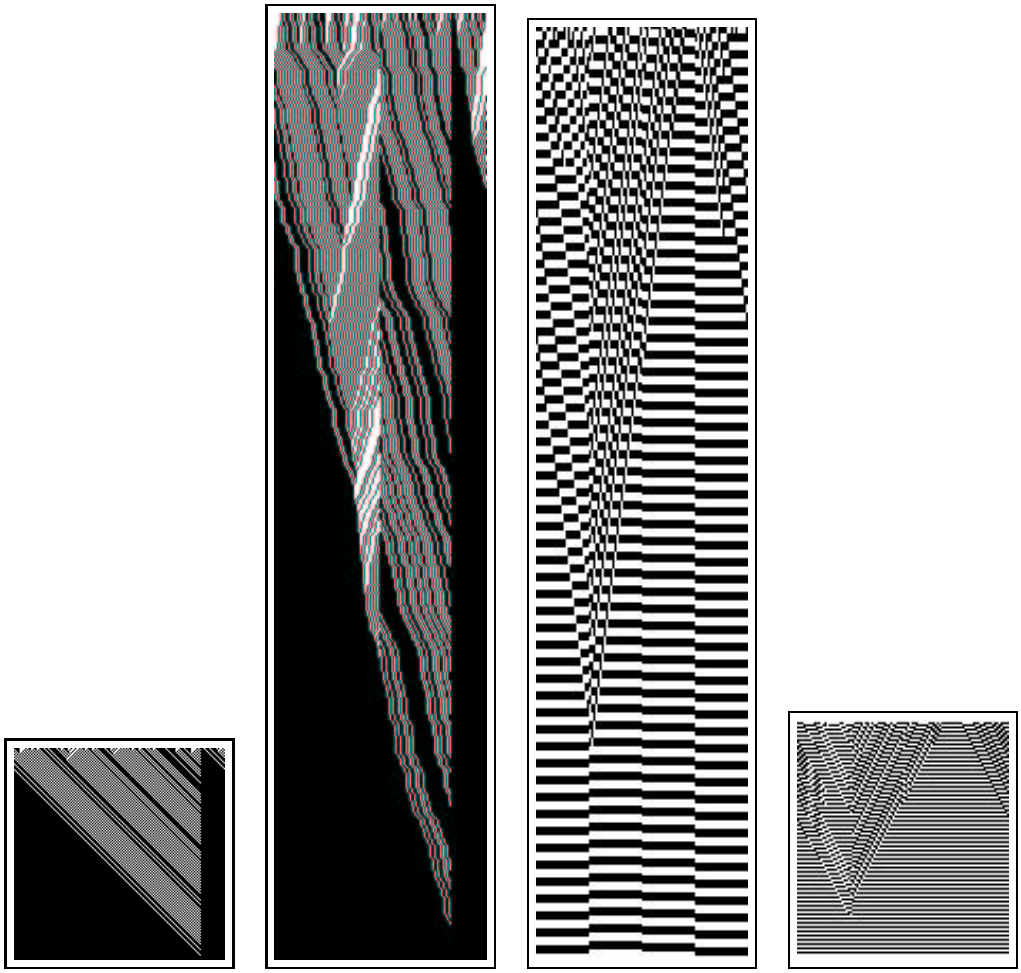


Figure 6.1 In (a) and (b) we can see the result on the one-dimensional density task of two co-evolved, non-uniform, connectivity radius $r = 1$ CAs. In (c) and (d) we can see the result on the one-dimensional synchronization task of two co-evolved, asynchronous (model-3), non-uniform CA, with connectivity radius $r = 1$. White squares represent cells in state 0, black squares represent cells in state 1. The pattern of configurations is shown through time (which increases down the page). (a) A synchronous CA. Grid size is $N = 149$. CA is run for 150 time steps. (b) An asynchronous (model-1) CA. Grid size is $N = 150$, with two 75-cell blocks ($\#_b = 2$). CA is run for 665 time steps. The randomly generated initial configurations in (a) and (b) have a density of 1s greater than 0.5, and the CAs relax to a fixed pattern of all 1s, which is the correct solution. In (c) and (d), CA size is $N = 150$, partitioned into 4 blocks ($\#_b = 4$). (c) The CA's configuration is depicted at every time step. (d) The CA's configuration is depicted at every logical step (= 4 time steps).

Our results for the density task show that model-1 asynchronous CAs can be evolved whose performance is comparable to the synchronous case,[4] provided the number of blocks does not exceed three ($\#_b \leq 3$). As we saw in chapter 4, a perfect solution for this form of the density task cannot be evolved. Actually, this result of comparable performance with low asynchrony to the synchronous case tends to confirm the clever "cheating" hypothesis I presented, as a reasonable information loss is not of great importance. Nevertheless, for $\#_b > 3$, successful asynchronous CAs for density classification did not evolve. Figure 6.1(b) demonstrates the operation of an evolved, non-uniform, model-1 asynchronous CA on the density task.

For the synchronization task, successful model-3 CAs with $\#_b \leq 8$ were evolved (grid sizes considered were in the range $N \in [100, 150]$); applying model 2, no successful CA had emerged from the evolutionary process. Figures 6.1(c) and 6.1(d) demonstrate the operation of an evolved, non-uniform, model-3 asynchronous CA on the synchronization task. The deterministic updating schedule of model 3 renders it easier for evolution to cope with, as compared with model 2. For both, however, an obstacle that hinders the evolutionary algorithm is the need to adapt to block boundaries. A "good" rule in cell i may be of no use, or even detrimental, in cell $i+1$, if a block boundary occurs between these two cells. Two strategies were observed to emerge through the evolutionary process in order to cope with this problem: either specialized rules are evolved at block boundaries (different than the rules present in the rest of the block), or a rule is evolved that is essentially insensitive to the presence or absence of a boundary.

6.3 Design and Evolution of Redundant Asynchronous CA

The relatively weak capability of binary CAs to cope with even limited asynchrony called for the use of redundant CA to deal with full asynchrony. I call redundant CA, a CA which uses more states in the asynchronous mode than is necessary in the synchronous mode to solve the same task. The idea behind redundancy is that the information in a CA configuration is not only the current state and the topology, but also the timing. For instance, in a synchronous CA, rule 184[5], if a block of two or more 1's is present at position i after t steps, it means that there are no block of white cells between the positions $i, i+t$, or more exactly that the density has been calculated for the cells $i-t...i+t$. Hence the information that this block carries is not just 111 but rather a 3-tuple $(111, i, t)$. Therefore, to deal with asynchrony, I add redundancy to store, partially, that timing information explicitly in the state rather than implicitly in the global synchronization as is the case in synchronous CA. As we will see, the redundancy allows for perfect or efficient solutions to be designed and evolved.

In section 6.3.1, I will study a designed time-stamping method that allows perfect resistance to asynchrony, i.e., there is no loss of information. In section 6.3.2, I present a low-cost imperfect solution to the synchronization problem. Using a peculiar property of this task, I

[4] Performance results for the synchronous case are reported in [179].

[5] in Wolfram's notation

develop a lossy strategy that allows for instant correction. Finally in section 6.3.3, I present a solution to asynchrony found through evolution. But first I describe the model used in the rest of this section on asynchrony.

The model: I use the most general fully asynchronous model. Each cell has the same probability p_f of not updating its state at each step. In this case the cell state remains unchanged. Otherwise the uniform or non uniform CA is perfectly classic. Hence, the probability of CA of size N working synchronously for t time steps is $(1-p_f)^{Nt}$.

6.3.1 A simple and efficient time-stamping method

If what is looked for is a method to perfectly correct asynchrony, then all information should be maintained. This is to say, all the 3-tuples (c, i, t), where c is the state of cell i at time t, of the synchronous case should be reconstructible in the asynchronous case. As a definition, I call the equivalent-time t, the minimal time of an asynchronous CA when we can reconstruct the information of the corresponding synchronous CA. The simplest way to achieve this information-keeping task, is that each cell store each time it updates the 2-tuple associating the current time and state. This way if a cell updates only if its current time matches a tuple in each of its neighbors, or waits otherwise, time consistency will be maintained. This solution answers the query but unrealistically, as the number of state grows indefinitely as time goes by. However, if we strip down this idea to the essential, then a practical perfect algorithm may be devised.

The method relies on two essential pieces of data. Firstly, a cell knows if it can update, i.e., if it is ahead of one of its neighbors or not. Secondly, a cell knows its neighbor state corresponding to its own state. This can be nicely summed up as: a cell must know if it can update, i.e., if it is allowed to and if it is able to update.

The method

Quite simply, a time-stamp is added to each cell so that the cell may know if it is ahead of one of its neighbors. The minimum value of the time-stamp, not to confuse between being ahead or being late, is 3. Then if each cell stores both its current and last state, the new CA is both able to know if it can update and how it should update. The simple algorithm below, executed at each update, will thus guarantee that whatever p_f the CA computes correctly.

```
If (for all neighbors)
        neighbors_time-stamp != (my_time-stamp-1)%3)
then
   past_state = current_state
   current_state = RULE(neighbor_states(my_time-tamp))
   my_time-tamp = (my_time-tamp +1)%3
endif
```

where `neighbor_states` *returns the current state of the neighbor if the time-stamps are identical or the past state of the neighbor if the time-stamps are different.*

Hence if the original CA had q states, the new CA has $3 * q^2$ states. This idea of having a phase component and a data component in the state set is not new as such. Toffoli in [207] presented such a solution in the general case of concurrent behavior of automata. A solution he reused with Margolus in [209] to deal with Asynchronous CA. However, it should be pointed out that their solution uses a phase component of 4 states, which, as we see, is not minimal.

Practical results

As the method corrects perfectly the synchrony errors, it is useless to test it for results as such. However, this may be illustrated with the adaptation of rule 184. As shown in chapter 4, this rule needs to lose no bit of information to work correctly. We can see in Figure 6.2, the 184-adapted CA working in a more or less faulty environment. This figure presents also another property that I am going to study now: time-efficiency.

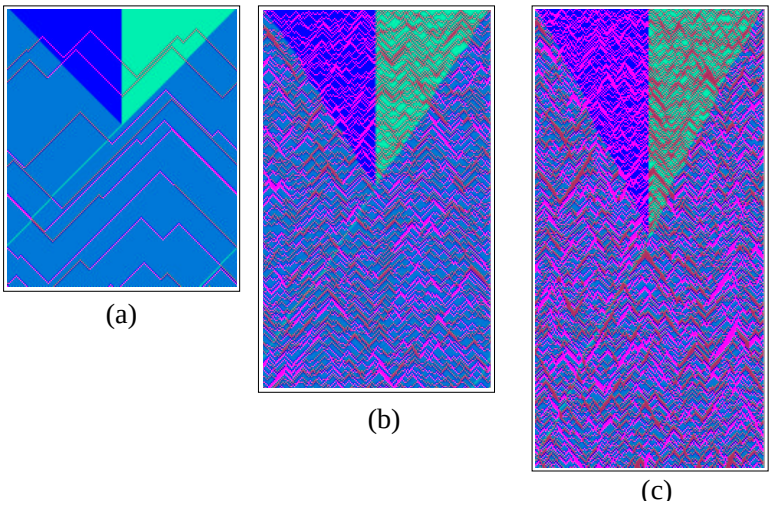


Figure 6.2 In all cases, the equivalent of synchronous CA 184 is run on the same initial configuration, $1^{79}0^{80}$. The blue color represents a data component of 1, green a data component of 0, while the pink represents a cell that did not update (either voluntarily or due to faults). In (a), p_f is 0.001, in (b), p_f is 0.1 and in (c), p_f is 0.2.

This simple algorithm possesses the nice property of using to the limit the inherent parallelism of CA. Quite naturally the healing is local and thus cells at a distance of more than r[6] can keep on working normally. But this would be almost useless if the lateness at the end of the run was equal to the number of synchronization faults. However, the algorithm cancels errors in such a way that the more errors happen the more errors are cancelled. This works in the following way. If a fault occurs, then the lateness propagates through the grid until all cells are late by one step. This "error" propagation proceeds quite naturally at

[6] r is here the radius of the CA considered.

the speed of light[7] as the correction mechanism is driven by the natural mechanism of cell update. But if another error occurs after that first error but above the propagation line, then the error propagation line will meet and "cancel" each other. Basically, all the automata will be late by one more step, but the ratio of lateness versus the number of errors will only be $1/2$. Then it appears clearly that the "best case" happens when all cells suffer from a fault at the exact same time step, then this ratio gets as low as $1/N$ which is its minimal value. In Figure 6.2.a, by looking at the upper part, one may realize more easily the cancellation mechanism, then by comparing it to 6.2.b and 6.2.c, one may see the importance of this ratio. In effect going from left to right, there are, statistically, 100 times more faults in (b) and 200 times more in (c) compared to (a).

Using the initial configuration of Figure 6.2, we can measure this lateness by measuring when the block of two 0's, issued from the computation of rule 184, reaches the left "border". This block reaches the border at time N when $p_f = 0$. I thus define the lateness L_t to be the difference between N and the effective time at which the blocks reaches the border. If I define N_f, to be the total number of faults that occurs during the run of the automata, then the ratio L_t/N_f will decrease from 1 to $1/N$ as p_f augments . We can see in Figure 6.3 the experimental data averaged on 1000 runs of the algorithms that illustrate this relation between p_f and L_t/N_f. We observe that this curve decreases very steeply to almost reach its minimal value as p_f is at 0.1, thereby underlining the importance of this error cancellation property and thus the suitability of this algorithm for dealing with problem-solving asynchronous CA.

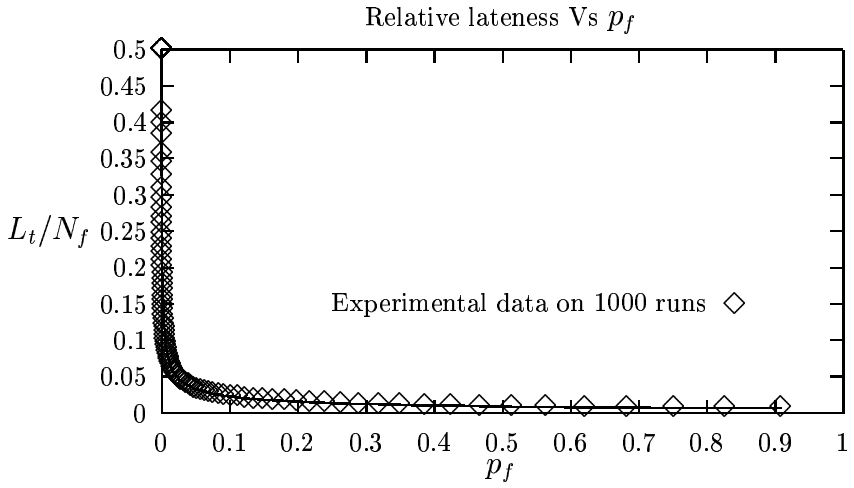


Figure 6.3 Experimental data demonstrating the effectiveness of our simple time-stamping algorithm. We call the relative lateness of rule 184, the ratio of the lateness (in the sense of figure 6.2) by the total number of faults happening in the automata. This graph shows a rapid decrease in this ratio, thereby illustrating the rapid increase in efficiency of the algorithm.

[7]The speed of light in a CA is the maximum distance that a bit of information may travel at each time step, which is r.

6.3.2 Quasi-perfect, efficient, lossy asynchronous synchronization

It is interesting, knowing that there exists a simple algorithm that perfectly recovers from synchronization faults, to study imperfect solutions that optimize other properties. In this subsection, I present an algorithm that corrects instantly the synchronization faults with fewer states, at the cost of imperfect correction of errors and with loss of information. It should be pointed out that the method uses *a priori* knowledge of the task studied, herein the synchronization task.

A 4-state lossy method

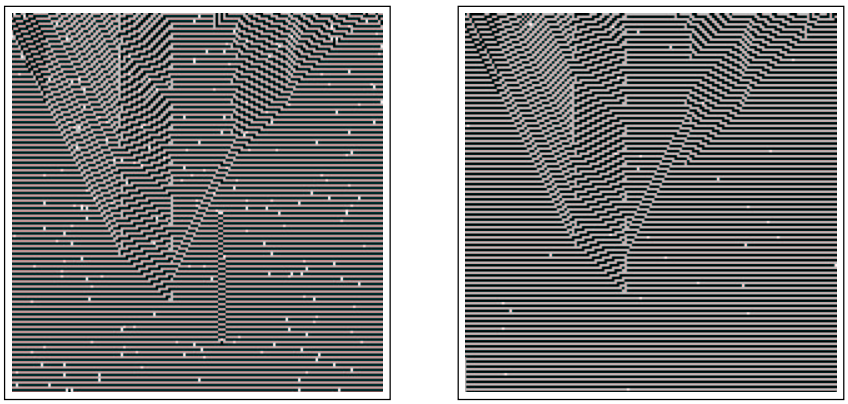
The idea here is to use the specificity of the synchronization task to correct instantly the synchronization errors. A CA solving the synchronization task, when converged, alternates between 0 and 1, thus if a cell is late by one step, at the next time step it will be in the correct state. Then it is easy to derive a 4-state lossy method. We need only a 2-state time-stamp, to detect if we are synchronized or not with the neighbors. Obviously this minimal solution has the drawback of not making clear if the cell is late or ahead. But this is not a problem because whether late or ahead by one or several steps is not of importance if we take into account the fact that the cell alternates between 0 and 1. If the cell is desynchronized by two or any even number of steps, it will be both in the "correct" data state and time state. In the same way, if the cell is desynchronized by one or any odd number of steps it will be both in the "opposite" data state and time state. As the CA is binary, with four states, the cell is able to cope with asynchrony by the following algorithm.

```
If (my-ts == ts-left &&  (my-ts == ts-right) then
  my-state = MyTransitionRule(left-state, my-state, right-state)
endif
If (my-ts != ts-left &&  (my-ts == ts-right) then
  my-state = MyTransitionRule(!left-state, my-state, right-state)
endif
If (my-ts == ts-left &&  (my-ts != ts-right) then
  my-state = MyTransitionRule(left-state, my-state, !right-state)
endif
If (my-ts != ts-left &&  (my-ts != ts-right) then
  my-state = MyTransitionRule(left-state, !my-state, right-state)
endif
```

where `ts` *is an abbreviation for time-stamp.*

The algorithm may be generalized. If a majority[8] of neighbors has a time-stamp identical to the one of the cell then take the neighbor state if its time-stamp is identical and its complementary otherwise. If a majority of neighbors has a time-stamp different from the one of the cell then do the reverse.

[8] As the neighborhood in the case of one-dimensional, regular CA is always odd, the majority is always well defined.



(a) (b)

Figure 6.4 We can see the same non-uniform synchronization task with the 4-state lossy method. The state 1 is represented as black, 0 as gray while white dots represent a cell that has not updated. In (a), p_f is 0.01. We can see a failure happening consecutively to 2 joint faults. We remark that this remnant fault is then corrected by two further faults. In (b) we can see the same CA correcting all the faults ($p_f = 0.001$).

In Figure 6.4, we can see the 4-state lossy method at work for two different p_f values adapted from the same binary non-uniform synchronous CA[9]. What is remarkable, is that this method, developed on the assumption of alternation which is only valid after the CA converged onto the synchronized cycle, does not hinder at all synchronization. There are mainly two advantages to this method compared to the perfect method described in the previous subsection. Firstly, it is low cost as only 4 states are necessary compared to the previous twelve. And, secondly, it corrects errors instantly. This may seem a poor advantage given the task tackled. In effect there is no "result" of computation to be collected here. However, we are very interested here in having a maximum number of synchronized cells. With the perfect method, the late cell "cone" grows from the fault, until the entire grid is swept. For the synchronization task, this leads to having on average only half the cells synchronized at any one time if faults occur. Obviously, this simple method is not perfect. Firstly it is not general, but more importantly, it does not guarantee perfect computation, as one may see in Figure 6.4.(a). Nevertheless, one may also note that another fault at the same site corrects the error. Formally, these unrecoverable errors occur when 2 or more faults occur on the same neighborhood. In effect, as seen in the algorithm, it is always assumed that only one error occurs. However, practically, as is shown in Figure 6.5 the error is less than what could be expected theoretically. This is due to the fact that some cases of two non-joint faults are corrected due to the particularity of the original transition rule. An example of this may be observed in Figure 6.4.(a). The probability of failure, i.e., the probability of a *de facto* uncorrectable error, was experimentally obtained and is compared in Figure 6.5 to the probability of a synchronization fault on the whole automata. The length of 159 was chosen as it is the size of the example of Figure 6.4. We considered the

[9]The original CA was found by evolution.

occurrence of an error in a classic CA to be a total failure, though as it is demonstrated in [188] this is not true as such. This approximation however is not false in the scope of this comparison as our CA is neither indefinitely affected by such an error as it is demonstrated in the previous figure.

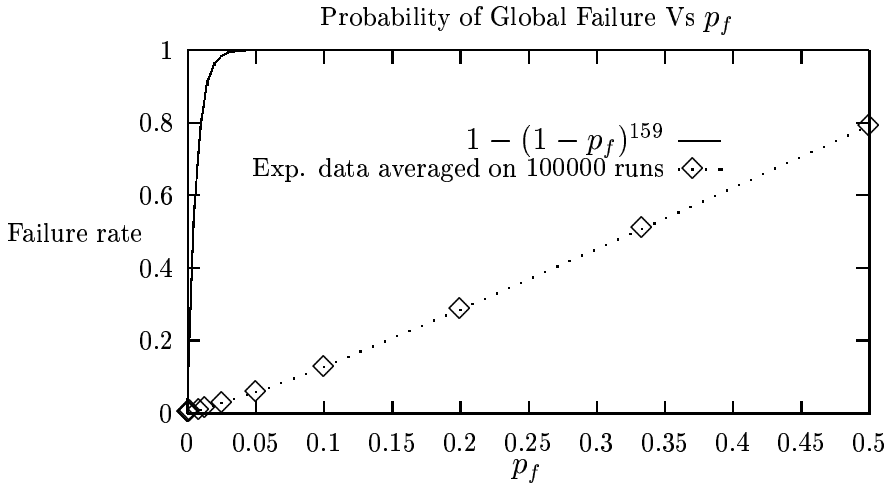


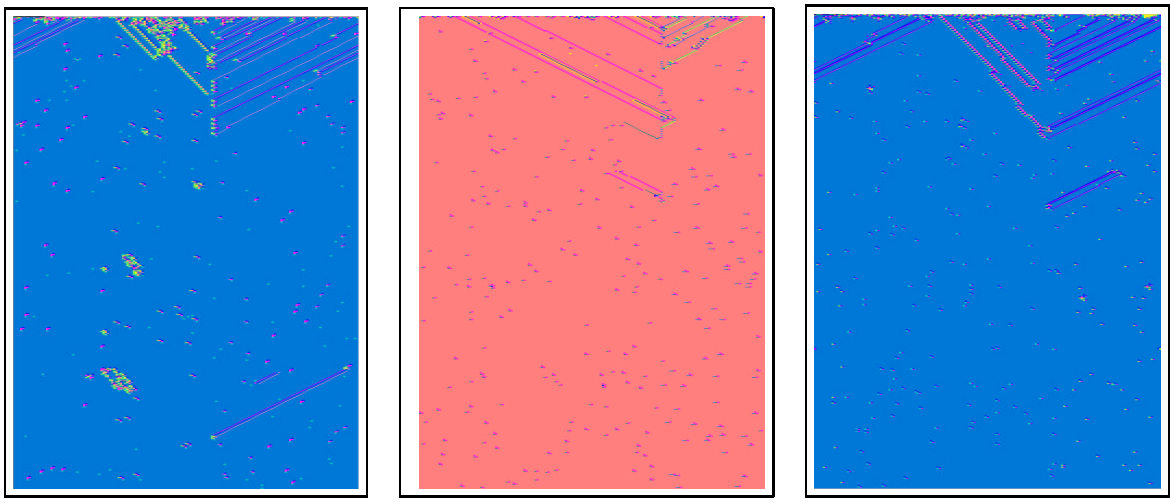
Figure 6.5 The global probability of failure versus the probability of fault of one cell. The plain line represents this global probability for a 159 cells classic automata, while the dotted line is for a cellular automata of the same length using the 4-state lossy method.

6.3.3 Evolution of asynchronous synchronization

To conclude this section on redundant asynchronous CA, I will quickly overview the evolution of such CAs. As we saw in section 6.2.1, the evolution of binary CAs does not produce good results on real asynchrony and could only cope with limited ($\#_b < 4$) regular asynchrony. In the previous subsection, a simple method to deal with full asynchrony was designed using only 4 states. I thus propose here to try to evolve 4-state CAs as we know that this was sufficient to develop good, imperfect, solutions.

The evolutionary algorithm used is the cellular programming algorithm presented in chapter 5, in Figure 5.2. The crossover used is the standard one-point crossover, thereby the number of states makes no difference. So, I evolved a 4-state non-uniform CA in the faulty environment. Asynchrony is then just one of the constraints like the radius or the number of states. Each string of genomes is tested on 100 configurations for 3 probability-of-fault values, $p_f = 0.001, 0.002, 0.006289$. The fitness is the "same" as the one for the binary case. After $1.5N$ time steps, for four steps, each cell gets a point of fitness if it alternates correctly between 1_{mod2} and 0_{mod2}. The first state to be in is determined by which state the majority of cells is in. Hence, as we see, if the 4 states are not distinguished into a data component and a phase component a priori, the definition of the fitness however clearly does this by not distinguishing 1 and 3, and 0 and 2. But as we are going to see, evolution did not use this strategy.

Globally the evolutionary runs are very successful, and if we consider a fitness of 0.98 as equivalent to a fitness of 1.0 in the synchronous case[10], the success rate is equivalent to the evolution of binary CAs in the synchronous case. Figure 6.6 presents three successfully evolved CAs.



(a) (b) (c)

Figure 6.6 Three examples of 4 state non-uniform redundant CA found through evolution that successfully cope with an asynchronous environment. The probability of synchrony faults here is $p_f = 0.002$ in all three figures. The colors represent the following states: blue is 0, cyan-green is 1, yellow is 2 and magenta is 3. The size of the CAs is 159 and there are 400 time steps shown here. The different strategies in (a), (b) and (c) are discussed in the text.

It is interesting to look at the strategies found by evolution to cope with asynchrony. As said before, our prejudices led us to define a fitness encouraging a phase and a data component just as in our solution. However, as appears in Figure 6.6, this is not the case: either the CA settled into the cycle 0,1 or the cycle 2,3 but none of the CA evolved exploited the possibility of falling into the cycle 0,1,2,3. Thus for most of the working the CA is not really using the extra states. Nevertheless, in the three cases above, the unused state in the synchronized cycle is used to go from the initial random configuration to the desired unified state. But, more importantly from our viewpoint, each fault of synchrony, produces one or two unused states in the next time step. This, somehow, allows the CA to "detect" the anomaly and correct it very briefly, in the next 1,2 or 3 time steps. In the worst cases, either a "tumor" develops briefly but is resorbed relatively quickly as in 6.6.a or the error drifts until it reaches a rule, or a group of rules able to absorb it, (6.6.b and c). It is important to note that neither the cell, nor the neighbor knows it suffered from a fault. Hence, this detection is only done through the wrong state of the cell concerned.

Actually if we test the best solution found by evolution (6.6.b), it turns out better than

[10] The faults introduce necessarily some cells in the wrong state.

the designed solutions, both the perfect and the imperfect ones, in terms of the percentage of cells synchronized for low values of p_f's. The test I realized was to measure the proportion of cells synchronized in the same state, in an alternating cycle, for different values of p_f's. Interestingly, all evolved solutions fared better for a probability of faults less than or equal to $1/N$, but in the long run the perfect solution which suffered from the fact that only "half" the grid is synchronized on average on the next N time steps if only one fault occurs, fares better. Then the fact that it does not lose information and that synchronizing trails cancel each other makes it the better solution. We can note also that the designed lossy method achieves its goal of being very good for low values of p_f, however in the middle range it is worse than the evolved solution but maintains performance better in the long run. Finally, as a control I tested the uniform solution. Obviously when the number of faults is very low, by its nature, it correctly resynchronizes. Nevertheless it falls quickly to low values as p_f increases to settle down close to 0.55. We should not be fooled by the lower score of the evolved solution for high p_f's . The unmodified version can take only 2 states and thus would fare 0.5 as the lowest possible score, while the evolved solution can take 4 states and thus could settle as low as 0.25. Figure 6.7 illustrates these results.

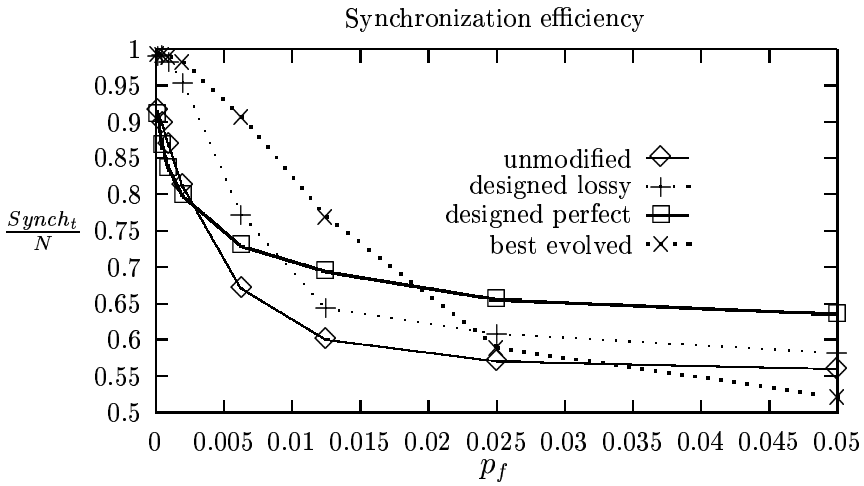


Figure 6.7 The experiments were run on 10 random initial configurations for 10000 time steps. The CA size N is 159. The "unmodified" version is the binary non-uniform CA that was the base used both for the "perfect" and the "lossy" method. The "best evolved" is the one shown in Figure 6.6.b.

6.4 Fault-Tolerant Cellular Automata

Cellular automata considered up to now in this thesis follow a deterministic rule. Hence no faults ever occur in the transition rule nor in the cell current states. Nevertheless if we are to look at real biology, it appears at all levels that either insensitivity to faults or self-repair is a prime condition of survival, the environment being extremely noisy. Fault resistant cellular automata have been designed early in the history of CAs. Some of the first examples may be

found in Nishio [140] and Harao [78]. However, these designs rely on a sort of hierarchical building and error-correcting code. The nice point about these is that they are general, the inconvenient being that they require a large minimal neighborhood (about 49 points to correct 1 error). Others like Peter Gács[63–65], followed an even more theoretical direction. The idea is to design a CA, where, starting from a uniform configuration of all 0s (resp. all 1s), we can guarantee that for all t and all x, $p_0(x,t) > 2/3$; $p_0(x,t)$ being the probability of cell x at time t to be in state 0 (resp. 1). The CA may use any number of states. Gács proved that this is possible to design such a CA in any dimension, including one, in any noisy environment. Practically, however, this is of no help as the number of states required is something between 2^{24} and 2^{400} according to Lawrence Gray's analysis [71].

In this section, I propose to develop fault-tolerant rules on a classical CA structure, via the design of the rule and using about the same complexity as the equivalent fault-intolerant CA.

I will first accurately define what is meant by a faulty environment. Then, in section 6.4.1, after a short reminder of error correcting code theory, I will adapt it to propose a general model of fault-resistant rules and the method used here. Finally, in section 6.4.2, I propose some structures developed with this scheme which goes toward a self-replicating loop.In this section, I first quickly evaluate the impact of the noisy environment on classical self-replicating structures, such as Langton's loop. And whereas the idea of self-replication in CA was motivated by the modeling of life, the current models do not present any quality of self-repair. This leads us, using our method, to propose some tracks for developing such a fault-tolerant self-replicating loop at possibly very little extra cost.

A non-deterministic cellular automata space

The idea of fault-tolerance is rather vague and our intention in this section is not to state any definitive definition. Rather, I propose here, first of all, to delineate the scope of validity of our work, and, secondly, to briefly argue its validity in a practical environment. Besides, as I will show later, this noisy environment is more than enough to irrevocably perturbate and thus destroy any of the most famous self-replicating structures.

Definition of the faulty environment: Formally cellular automata are d-dimensional discrete space, in which each point, called a cell, can take on a finite number of states q. Each cell updates its own state according to the states of a fixed neighborhood of size k following a deterministic rule. A CA is called uniform when this rule is the same for every cell. Hence a CA may be defined by its transition rule S:

$$S : q^k \to q$$

In this model, faults may basically occur at two levels : synchrony (time) and cell transition rule (space). As for the former problem we saw in the previous section that a time-stamping method may prevent all synchronizing defaults. But, more importantly in this section, synchronization faults may be modeled as a particular instance of the general faulty environment presented here. As for the latter problem, which could be further divided into reading and writing errors, we model it as a non-deterministic rule. More precisely,

there is a probability of faults p_f, such that any given cell will follow the transition rule with probability $(1 - p_f)$, and erroneously take any state with probability p_f. This model, though apparently catering only for writing errors, simulates perfectly reading errors or synchronization faults. One may object that we did not mention the major fault problems when a cell simply stops functioning at all. It is not our purpose to treat this kind of 'hardware' problem here, but we may say that our model could be implemented over an embryonnics tissue[125] which deals with this kind of malfunction. I should say also that a permanent error is not fatal in itself, it just makes the neighborhoods containing this cell as fragile as non fault-tolerant CAs.

We now explain how to design fault-tolerant structures.

6.4.1 Fault-resistant rules

In this noisy environment a cell cannot trust entirely its neighborhood to define its future state. In other works, where the aim was to design generic faultless cellular automata, a hierarchical construction was used, e.g., a meta-automata of dimension 2 creates a fault-tolerant 1-d automata. However our approach here is *not* to create a fault-tolerant CA structure, but rather to design the rules of a classical CA so that the emergent, global behavior is preserved. This latter choice allows both a reduced computation time and lesser memory space than the former, the cost being a resulting CA specific to one task. In this section I first briefly present some elements of error-correcting code theory, on which I base the design of fault-tolerant rules presented afterwards.

Error-correcting code

The idea behind the error correcting code theory is to code a message so that when sent over a noisy transmission channel, at reception, one can detect and correct potential errors through decoding, and thus reconstruct the original message.

Formally, let Σ be an alphabet of q symbols, let a code $\mathcal{C}$ be a collection of sequence c_i of exactly n symbols of q, and let $d(c_i, c_j)$ be the Hamming distance between c_i and c_j. We call $d(\mathcal{C})$ the minimal distance of a code $\mathcal{C}$ which is defined as $\min_{i \neq j} d(\mathbf{c_i}, \mathbf{c_j}), c_i, c_j \in \mathcal{C}$. Then the idea of an error-correcting code, introduced by Shannon[173], is to decode, on reception, the word received by the word belonging to $\mathcal{C}$ which is at minimal Hamming distance. Using this simple strategy with a code $\mathcal{C}$ of minimal distance d, allows the correction of up to $(d-1)/2$ errors.

In this theory one may see that the number of words in $\mathcal{C}$, M, and the minimal distance d, play against one another for a fixed n and q. To avoid waste of memory space in our fault-tolerant rules developed in the next subsection, it will be useful to maximize M for a given d. However, it is not always possible to determine it *a priori*, but for $d = 2e + 1$, so that we can correct e errors, this maximum M, $A_q(n, d)$, is bounded. The lower and upper bounds known as Gilbert-Varshamov[216] bounds, are:

$$\frac{q^n}{\sum_{i=0}^{d-1} \binom{n}{i} (q-1)^i} \leq A_q(n,d) \leq \frac{q^n}{\sum_{k=0}^{e} \binom{n}{k} (q-1)^k} \tag{6.1}$$

Theoretical aspects of fault-resistant rules

A rule r in a 2-dimensional cellular automata is the mapping between a neighborhood, v, and future state, c^+, and, if we use Moore neighborhood, may be written as $(v_1, ..., v_9 \rightarrow c^+)$. In the noisy environment, as defined in section 6.4, the values $v_1, ..., v_9$ may be corrupted, thereby inducing in most cases a corrupted value of c^+, propagating in time the original error. The idea behind this method is to see c^+ as the message to be transmitted over a noisy channel, and $v_1, ..., v_9$ as the encoding of the message. Then it becomes clear that if this encoding respects the error correcting code theory constraints exposed above then, even if the neighborhood values are corrupted, it will still be decoded into the correct c^+ future state and the original errors will not last further than one time step.

Of course, the constraints will depend on the number of errors, e, we want to be able to correct for a given neighborhood. We define $(v_1, ..., v_9 \rightarrow c^+, e)$ to be the transition rule r resisting e errors. In consequence, we also define $V(r)$ to be the set of rules co-defined by r, $V(r) = (\overline{x} \rightarrow c^+), d(\overline{x}, v_1..v_9) = e)$. Then it appears clearly that if each rule of the co-defined set of rules is at a minimal Hamming distance of 1 of any other rule of the co-defined set of rules, that is, if each main rule is at minimal Hamming distance of $2e + 1$ of any other main rule, then, reinterpreting the result of Shannon above, we can guarantee that the CA thus defined will be able to function correctly providing that at most e errors happen at any one time step in each neighborhood. As one may note, the size of $V(r)$ is rather large. More precisely, for a neighborhood of size n, and an alphabet of q symbols we have:

$$|V(r)| = \Sigma_{k=0}^{e} (\frac{n}{q})(q-1)^k \tag{6.2}$$

A *conflict* appears when the intersection between every co-defined set of rules is not empty. For instance, for a von Neumann neighborhood, $r_1 = (03234 \rightarrow 1, 1)$ and $r_2 = (03034 \rightarrow 2, 1)$ are in major conflict as the neighborhood (03234) lies both in $V(r_1)$ and $V(r_2)$. To avoid wasting too many possible different neighborhoods, we distinguish *major* and *minor conflicts*. Major conflicts appears when the future state of the conflicting rules are different. However, minor conflicts, when future states of the conflicting rules are identical, are reasonable as it does not prevent error correction. Thus we have the following reinterpreted Shannon's theorem:

Theorem 7 *Given a neighborhood of size n and a collection of rules resisting e errors, if we have:*

$$\forall i \neq j d(r_i, r_j) \geq 2e + 1 \textbf{ or } c_i^+ = c_j^+$$

then we can guarantee that the CA will correct up to e errors occurring in one time step in any neighborhood.

We have now defined theoretically how to conceive a fault-tolerant structure on a classical CA. In the following, we study the practical application of the above defined constraints to peculiar substructures of interest for self-replication. On this more practical aspect of the

question, it would be interesting to know how many different rules are available for use. If we do not take into account the distinction of minor and major conflicts, and as we need all the rules to be at a Hamming distance of $d = 2e + 1$ to guarantee distinct co-defined set of rules, the number of rules available is given by the bounds of Gilbert-Varshamov (eq. 6.1). Nevertheless these bounds are rather large, and do not include the large advantages brought by the minor conflict exception. Moreover, if the waste of possible rules is rather large, it is not more than in classical self-replicating structures, such as Tempesti's or Langton's loops. In the former case, the number of used rules compared to the number of rules available (namely q^n), is well within the Gilbert-Varshamov bounds for a minimal number of errors $e = 1$. And even for $e = 1$, the global probability of failure is well decreased by our scheme for values of p_f between 0.001 and 0.05 as may be seen in Figure 6.8. In effect, the probability of failure of one neighborhood is reduced, roughly[11], to $1 - (1 - p_f)^n - n * p_f(1 - p_f)^{n-1}$ from $1 - (1 - p_f)^n$.

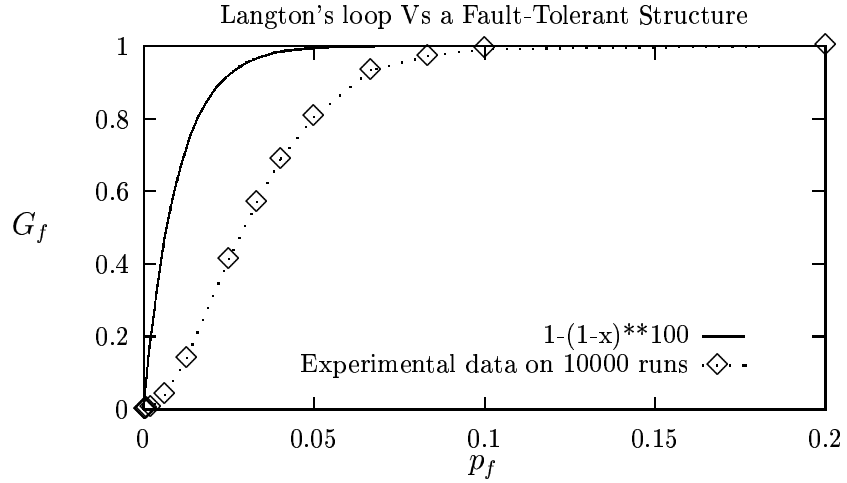


Figure 6.8 In this figure we can see the comparison between the probability of faults of the Langton's loop (which spans about 100 cells) and the experimentally obtained probability of the faults of a 100-cell structure designed according to the fault-tolerant method.

6.4.2 Fault-tolerant self-replication

Self-replicating structures in cellular automata have generated a large quantity of papers [183]. In the Artificial Life field, most of these have been devoted to the study of the self-replicating process in itself, as a model of one of life's primary properties. The underlying motivation of such research could be expressed as a question: What are the minimal information processing principles involved in non-naive self-replication ? I believe that in past studies the property of self-repair, of tolerance to a noisy environment, has been omitted for practical reasons rather than theoretical choices.

[11] I do not take into account here the fact that a cell may "fail" in the correct state.

In this section, I propose to apply practically the theoretical ideas put forward in the previous subsection. I will only consider rules that can correct up to 1 error in their neighborhood, as it seems a reasonable compromise between the fault-tolerance capabilities and the implied "waste" of rules. In the same way of thinking, I will only consider the two-dimensional Moore neighborhood, as it provides a wealth of rules compared to the von Neumann neighborhood, at no additional cost in terms of computational time, memory space, or physical complexity. But let's first have a quick look at how classical self-replicating loops behave in the faulty environment.

Classical self-replicating structures behavior

In this section, I demonstrate the brittleness of some famous and less famous classical self-replicating structures. I first consider Langton's loop[112], then move on to the more complex Tempesti's loop[204], to finish with one of the simplest self-replicating loops described by Chou and Reggia[162]. I do not consider the seminal work of Von Neumann[139] for obvious practical reasons. However one may note that the very accurate nature of this work leads to "natural" brittleness.

Applying this model of noisy environment to the landmark self-replicating structure of Langton shows high sensitivity to failure. Following the notation described above, a probability of fault p_f as low as 0.0001 shows a complete degradation of the self-replication function. Tempesti's structure shows exactly the same type of failure for an even lower p_f. The main cause of this total fault intolerance is due to what we may call irreducible remains (garbage). Both these loops have as default rule, no change. When confronted an unexpected neighborhood, i.e., when the fault only generates unexpectancy and not confusion, the cell remains in its previous state. Then garbage that appears in the void (in the middle of a quiescent zone) is irreducible and remains forever, or at least, and that is the source of this high-sensitivity to faults, until the loop meets it for its inevitable loss.

The Chou and Reggia loop, [162], begins to show significant degradation with a probability p_f of 0.001. This relative resistance is only due to its small size (3x3), and any fault affecting the loop almost inevitably leads to its complete destruction. One may note that this loop suffers, as the preceding examples, from *irreducible remains*.

It is clear that these rules were not designed to be fault-tolerant and thus their size is their only means of defense. Effectively, their probability of failure may be approximated by $(1-(1-p_f)^{size})$. Of course this probability, to be accurate, should be augmented by the number of neighboring cells, and diminished by the non-degrading fault probability, the latter figure being almost nil for the above mentioned structures.

Fault-resistant structures

The aim of this work is to develop a fault-tolerant self-replicating structures. I propose here in detail three essential substructures to attain such a goal: data pipe, signal duplication, and constructing arm. Firstly, I will study some common problems and desirable properties to construct such structures. Secondly, I will show how to practically construct a fault-tolerant data pipe, a structure essential to convey information, and thus be able to construct a

looping structure. Then briefly a fault-tolerant signal duplicator and constructing arm will be presented, as central elements for self-replication. Finally, a fault-tolerant constructing loop is demonstrated in Figure 6.15. The structures presented here were partially developed in common with Daniel Bünzli.

Properties, methods and problems

Our CA's structure is quite classically defined as an "active" structure in a pool, possibly infinite, of quiescent cells (henceforth represented by the '.' symbol). Hence the first necessary rule is $r_q = (\text{.....} \rightarrow ., 1)$. Its co-defined set of rules $V(r_q)$ defines all the rules with one active cell. It has the advantage of covering what I called in earlier the irreducible remains. Effectively, any fault creating a single active cell in a quiescent neighborhood will be reverted to the quiescent state at the next time step. This eliminates one of the main problems encountered by the non-fault-tolerant structures. Although the encounter of our fault-tolerant rules with the remains would not be as deadly as for the classical loops, this prevents accumulation of errors in the same neighborhood over time. Nevertheless, it is important to note that that peculiar properties, consequent to the definition of r_q, forbid any rules with less than 3 active cells in its neighborhood. Its co-defined set of rules would otherwise intersect with $V(r_q)$ and create a major conflict, unless, of course its future state is '.'. This last constraint which may be seen as a disadvantage in terms of the global size of the structure, is not much of a problem from the point of view of error-correction as the fault-tolerance capabilities of our structures only depends on the size of the neighborhood and not on the global structure.

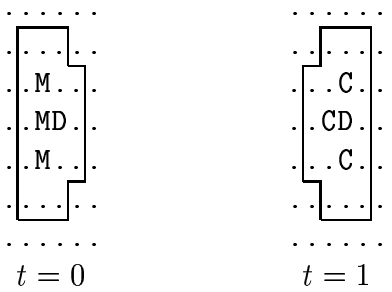


Figure 6.9 `Structure and its neighborhood`

The latter implied property brings us to a more general remark about the method to use when creating fault-tolerant structures. If we observe figure 6.9, one would be tempted not to define the rule $r_1 = (\text{.....}M \rightarrow ., 1)$ arguing that it is covered by $V(r_q)$. However, this would be mistake as $V(r_1) \neq V(r_q)$ and thus we would lose our guarantee of error correction. Hence, minor conflicts are handy as they allow us to define rules such as r_1 and r_q simultaneously, nevertheless they do not make the definition of r_1 unnecessary if we are unsure of the fault-tolerance property.

Besides these kinds of "tricks" one should use, I am currently developing a method to facilitate the creation of fault-tolerant rules. At the moment, the "algorithm" goes as follows:

When going from the structure at one time step to the next one, I define the rules (with $e = 1$) for each cell in the structure and its neighborhood, going from left to right and top to bottom. After each rule definition, I test for major conflicts. This way of proceeding prevents much useless work, but does not discard the design by hand of the initial structure and its successors.

Safe data pipe

Now that we have seen the theoretical aspects and some general practical constraints to define fault-tolerant structures, we are able to define a fault tolerant data pipe. Effectively, self-replicating loops are usually based on such information transmission structures. These allow simply to move information form one part to another of the cellular space. A typical example is given in figure 6.10.

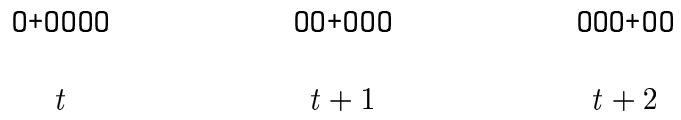


Figure 6.10 `Usual data pipe`

One sees immediately the problem with such structures. It implies the two following rules $r_1 =$`(0...0+0 → +)` and $r_2 =$`(+0...0 → 0)`, which obviously provoke a major conflict. In fact, we remember that each rule must have a Hamming distance of 3 between each other and thus at least 3 cells must change. This last constraint has the advantage of solving the quiescent-rule constraint also. We can see in figure 6.11 an example of a fault-tolerant data pipe, thus respecting that constraint.

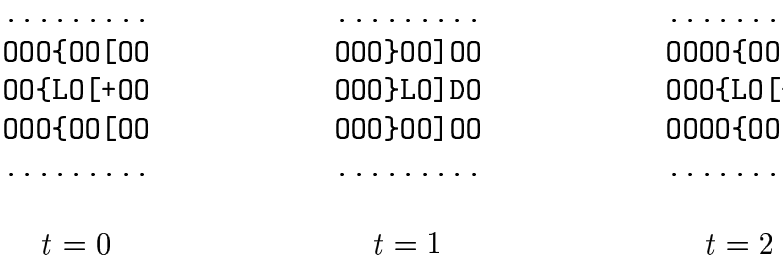


Figure 6.11 `Fault-tolerant data pipe`

Actually, this data pipe is able to transmit trains of data. As one may see, two types of data are transmitted in this example: `L` which uses 3 states to move itself, and `+` which uses 4. As we will see later the 4 states are needed as it is the *head of the data train.*

I will now take a more detailed look at this fault-tolerant data type.

- at $t = 0$, the upper and lower "wings" are necessary to maintain the '0' state situated at the 'east' of the head of the signal, the '+'state. This state '0' is designed to make the head move forward.

- at $t = 1$, we have a transitory structure. This transition is necessary to maintain the integrity of the pipe. Effectively, the upper and lower "wings" have not been moved yet. If they were still in the state '[' then we would end up with a transition rule `(0..00D[[. → [,1)` which would enter in direct conflict with the `(0..000+[. → 0,1)` defined at the preceding time step. This transitional step creates a diversity in the neighborhood, thereby suppressing all the conflicts.

- at $t = 2$, we get back to the original structure, the train of signal (which one may define as L+) has moved forward.

As I noted earlier, a supplementary state was required for the head of the signal, the data '+', than for the rest of the signal 'L'. It can be clearly seen why, if we imagine the situation without the supplementary state (see figure 6.12).

```
000}00]000
000}L0]+00
000}00]000
```
$t = 1$

Figure 6.12 `Problem with the head signal`

In this situation we have to define the rule `(0000000+0 →0,1)`, which then conflicts with the rule `(000000[+[ → 1,1)` defined at the preceding time step. The transitional state 'D' solves this conflict. The neighborhood of L, and any other following signal being more diverse, that supplementary transitional state is then useless.

Signal duplication and the constructing arm

Now that we have, through the detailed definition of a fault-tolerant data pipe, seen the practical aspects of making fault-tolerant structures, we propose in this last section to quickly view a fault-tolerant signal duplicator and a constructing arm. These peculiar substructures are always present in self-replicating loops.

```
.........   .........   .........   .........
000[0X...   000]0X...   0000DX...   0000]X...
00[D0XX..   000]LXX..   000[[XX..   0000]LX..
000[0X...   000]0X...   0000DX...   0000]X...
.........   .........   .........   .........
```
$t = 0$ $t = 1$ $t = 2$ $t = 3$

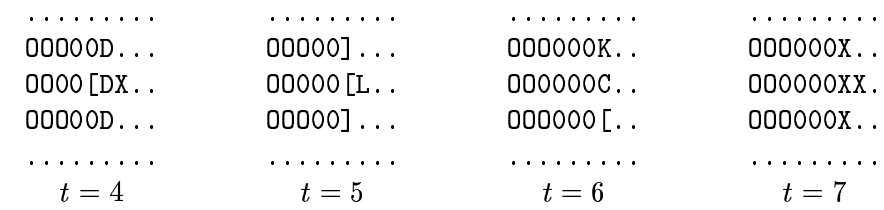
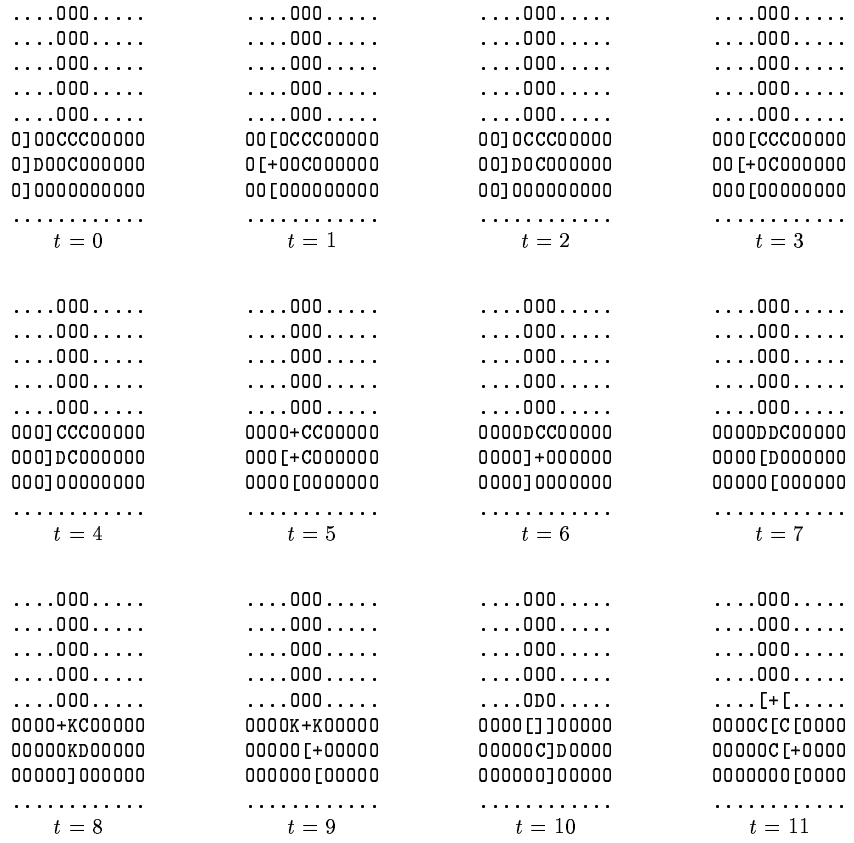


Figure 6.13 `The constructing arm.`

Actually it is the core of self-replication. The data pipe is there to convey information, the duplicator is there to allow self-replication, and the constructing arm is there to create the replicate. We can check that this rather more complex structure only requires the use of three more states besides the state needed by the data pipe described above, two, `C` and `K`, for the duplicator described in Figure 6.14, and one `X` for the constructing arm described in Figure 6.13. This last supplementary state is absolutely necessary as it is a zone where we have to maintain integrity between the void and the structure.



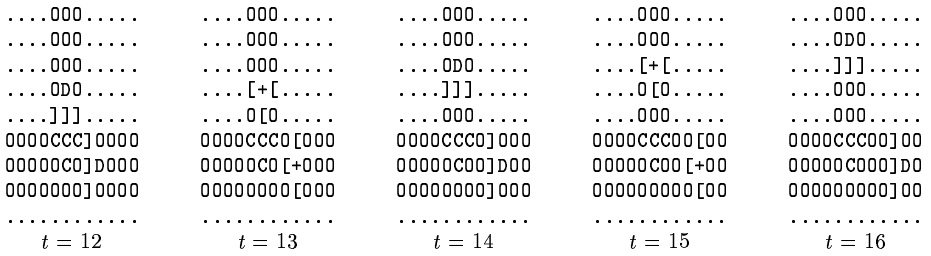


Figure 6.14 Duplication of a signal

6.5 Concluding Remarks

Constructing robust CAs has been around for some time now. Fault-tolerance has been a theme of research since the mid-70's and the question of asynchronous CA was tackled by researchers in the mid 80's and 90's. However, our point here was not really to introduce new methods, most of them rely on known principles, but to illustrate the question of information in CAs and the use one could make of this to acquire robustness.

For the synchrony question, the objective was double. The first one, a minor one, was to demonstrate that the simplest method to correct synchronization faults was excellent for CAs. In effect, it exploits fully the inherent parallelism of CA to minimize the total delay incurred. As an aside, it was interesting to note that this "perfection" was relative and for tasks that lose information, imperfect solutions could turn out to be better. This brings us to the second point which was that the information in a CA was time dependent. Most of the studies on asynchronous CA aimed either at proving that computation in CA was an artifact of the global clock, or that asynchronous mode could lead to interesting global behavior. Our point here is to express clearly that these studies are two faces of the same coin. As clearly demonstrated by rule 184, the real state of a cell, i.e., the information it carries, is a state, a time, and a place. Some tasks, obviously, do not consider all this information, and this is why imperfect solutions may work, but if we are to design asynchronous cellular systems, we have to, in full knowledge of the consequences, choose between "lossy" methods or systems that maintain the integrity of information.

The fault-tolerant CA study started from the following assessment: in 2-dimensional CA, there is a huge waste of possibilities. A large number of the possible states of the structure are usually unused whereas the complexity, at least theoretically, is present. The idea was thus to use this information available for "free". Our loop, shown in Figure 6.15, uses extra information beyond what is needed for the task. To make things clearer, the "184-corrected" CA presented on page 105 is a regular 12-state CA that can work in an asynchronous environment, but this is just due to the fact that it uses extra states to maintain the level of information it needs. Identically, our constructing loop uses, in a hidden way, extra states to maintain the required information level necessary for the task. However, the trick here is to use the available neighborhoods in the classical structure to hide the fact that we need extra states, and thus end up with a structure that is, apparently, inherently fault tolerant.

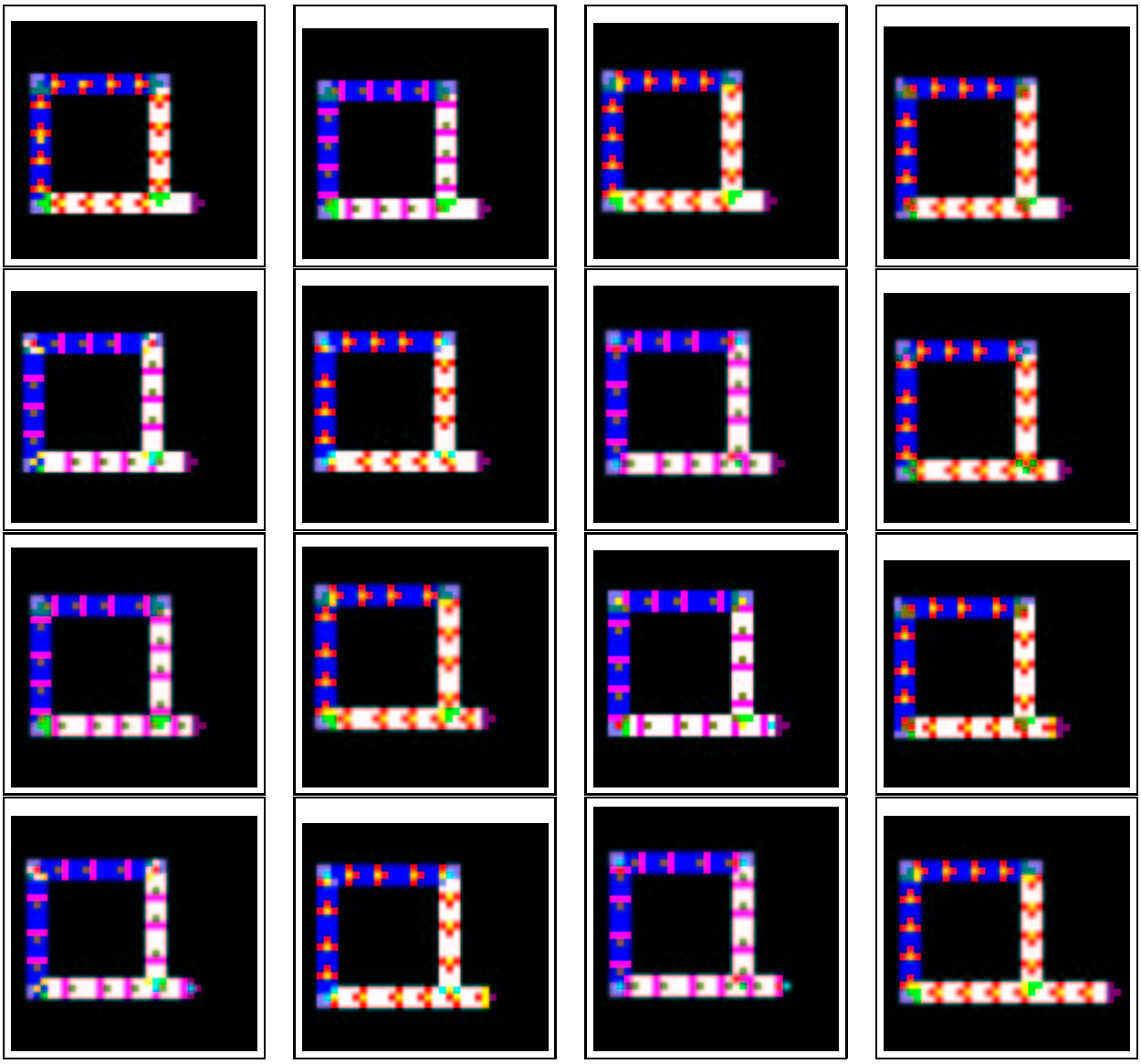


Figure 6.15 This figure shows the complete constructing loop. It conveys and duplicates the signals and constructs an advancing arm. As one may note, in the border, a peculiar cell configuration was necessary to insure the integrity of the loop while not overlapping with the neighborhoods implied by the empty spaces on the side of the loop. Also note that to make the data pipe reversible another "background" was used. The structure as it is here relies on a 10-state CA + 1 quiescent state. This is a little bit more than Langton's loop while the structure does not yet self-replicate, but one should take care that it has not been fully optimized. The figure shows succeeding time steps, except the last one which is taken several time steps later to show the growth of the arm.

La dernière chose que l'on trouve en finissant un ouvrage
est celle que l'on doit mettre en premier[1].
Blaise Pascal, *Pensées*, 757 in [149].

Chapter 7

Conclusions

This thesis investigated computation in cellular systems, and more specifically in cellular automata. This study concerned principally the question of what constitutes computation, but also the two natural corollaries, why are these systems interesting, a pre-question, and how to perform computation in cellular systems, a post-question. This handful of questions, which was put on the table in the introductory chapter were answered throughout this theses. I will now address these in the form of a general conclusion to this work. Then, I will present the main personal contributions excerpted from each preceding chapter, in the form of an ordered list, as is the tradition. Finally I will discuss some of the paths for future works opened by this work.

General Conclusions

The conclusions I am going to draw now concern mainly cellular automata, but their validity extends, at least partially, to cellular systems in general.

The nature of computation in cellular automata is a complex one. As we saw in chapter 2, there are four ways to consider this question, but only the global, emergent type, the second kind of the third type therein, was of real interest to us. In effect, only this type of computation presents the fascinating qualities that one seeks in cellular systems, namely, a global behavior that goes beyond the capabilities of each of the elements and dismisses artifacts of sequential systems developed on a cellular substrate. But the nature, the essence of the computation of this kind is far from obvious. Actually, as gleaned from the density task example studied in this thesis, computation may be as simple as a reordering of 1s and 0s. Thus the simplification of the output in that sense is only simplification in the eye of the beholder. This is particularly important, in my opinion, as it means that computation occurs in creating order (regularity). To be more accurate, it creates a large "ocean" of regularity which allows the onlooker to see the distinguishing patterns of irregularity. Hence, one may conclude from this, and I would, that problem-solving truly emergent CAs are to be found in class II rather than Wolfram's class III CAs. I believe that this conclusion goes well beyond the density task. If we look back on the Cellular Automata literature this

[1] The last thing one finds when finishing a book is the one that must be put first.

feeling for the necessity of a beholder is often explicitly or implicitly quoted. Wolfram's classes, for instance, are only a subjective classification depending on the aspects of the space-time diagrams. Langton's loop, which does not lie in this category of computation, still relies on the impression of life it leaves in the eye of the spectator. Actually, most CAs designed in the past have most often been (inappropriately ?) aimed at human, knowingly or unknowingly. And even the original definition of the density classification task was taking into account that fact. Effectively, even though it was defined according to a model close to the one of figure 7.1(a), that is of a classical computing machine taking an input and finishing in a yes or no state, the specification of the output itself, all 0s or all 1s, was clearly aimed at a human. A few years ago, Ronald, Sipper and myself proposed a definition of emergence which relies on the idea of an observer, [164][2]. In fact, now we can establish our own definition of emergence which is a little different. Beneath the idea of emergence, there is always the key concept that *new properties* that emerge were not present at the level of the individual element. In fact, this is the only tangible aspect of emergence. But, by definition, a property has an effect. A property without effect has no existence. For instance one may say that the emergent property of social insect societies is robustness to attack and to environment hardship. The effect is then their ecological success. However, even if these properties and effects are objective, their consideration and definition is subjective. For instance, the necessity of describing a bunch of H_2O molecules as being liquid is merely a question of viewpoints. It appears as a necessity, given the dramatic effects of the liquid property, but one should remember that the liquid state of the matter is still ill-defined as of today. This inevitable subjectivity lead many to introduce the idea of an observer [14, 45, 164]. So this idea of the necessity of an observer is not new in itself, but I go further in this thesis. In emergent systems, the observer is necessary to establish the emergent properties and its effects, but usually not for the property to happen. However in the case of CAs, computation only happens in the eye of the beholder. The CA in itself cannot get from any feedback, such as the environment for social insect and thus the effect of the property can only take place in the observer. This is due to the simple fact that no element, especially in simple, basic binary CA could "grasp" the global, emergent result...by definition. The object of the effect, "the effectee", is here only the observer. Maybe this is the main point of this thesis, CAs can't be self-inspecting and **emergent computation in CA is thus epistemic**. This does not mean that there is no computation in a CA. A CA computes actually because of our inability to see in the input arrangement and the local rule what will happen after a few time steps. CA computation hinges on the weakness of our mind. However this is not degrading at all and in fact is common to all sorts of computation. If one would see the same thing in $\sqrt{27225}/5$, and in 33, then there would be no need for calculators. But it is meaningless to consider this computation *in abstracto*, *in absentia* of an observer. Concretely, this means that a CA computes if from a chaotic input configura-

[2]In this context, I should say that Gordon, who argued that this idea of a designer/observer implied the idea of god, [70], misunderstood our argument. The necessity of the observer is only there to establish the emergent behavior, not for the behavior to occur. Though an insect is not conscious of death, it still dies. As for the designer argument, though maybe not clear at the time, I would say today that it just subsumes the idea that local behavior is quite objectively established, whereas global behavior is subjective, a question of perspective.

tion, it stabilizes in a global state that appears *visually* to be ordered, understandable. A problem-solving CA is the fortunate meeting of a good look with a good local rule. This couple CA/beholder opens the path for new considerations. For instance the result is not anymore a final configuration but rather the global temporal dynamic of the CA, such as illustrated in figure 7.1(b). This last point is interesting in two ways. On the one hand, it discards the need to look after n time steps, which solves one of the usual paradox. Solving the density task without a global counter, but requiring it to get the final result is rather paradoxical. On the other hand, now considering cycle and the global dynamic provides a much richer pool of potentially interesting CAs. This view of problem-solving CAs could have great influence on how one may find such CAs. For instance, by evolution, whereas today how the CA should behave globally is decided a priori, in the future an artificial observer could actually judge the fitness of the CA, the result being then the pair observer/CA.

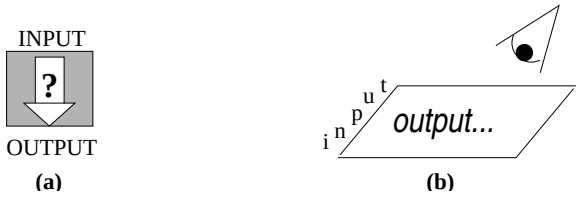


Figure 7.1 We need to change our look on CA computation to grab its full potentialities.

If the desired global behavior is fixed *a priori*, as is the case in the synchronization task, finding the good local behavior to produce is still not a trivial problem by far. Evolution is the natural way to proceeds as it is inherently a system that maps a genotype, the local rule, to a phenotype, the global behavior. However, as we have shown in the case of asynchronous CA, it is often useful to remember what is the information contained in a CA. In effect, it is often taken for granted that the global state of a CA is the spatial arrangement of the states of each cell. This has led many researchers to conclude that the behavior of a CA is an artifact of the global clock. This conclusion is interesting in two respects. First, it reflects the fact that one views the computation of the CA to be the visual result. The spatial arrangement of the states of each cell is nothing else than what is viewed. Second, it deceptively implies that if a two-state CA solves a task working synchronously, one should have expected to find another two-state CA solving the task asynchronously; the set of possible global states, the set of possible spatial arrangements of the states of each cell, being a priori the same in both cases. As the study conducted in this thesis with respect to asynchronous CA has shown, this is unwarranted. The information contained in a CA is really the states and the spatial arrangement, but also the time step. If we take all this information into account then in synchronous and asynchronous modes the set of possible global states are totally different. This appears clearly in the evolution of asynchronous CA, while the evolution of binary CA gave little results, the evolution of slightly redundant CA gave surprisingly good results. As shown with the perfect method of correction of asynchronous CA, three times the original number of states in the asynchronous case is enough to maintain the same level of information as in the synchronous case. Hence, the interesting question, when trying to

find a good local rule, is what are the necessary number of states to solve the task in the given environment, a necessary number of states which should inevitably be augmented in an asynchronous or faulty CA.

These were the main conclusions I wanted to draw from this work, as they are in my opinion good answers to the original queries. Nevertheless, this work brought many other contributions, more or less important or factual, that I expose below.

Thesis Contributions

This section highlights the contributions of this work, organized in chronological order.

- *A novel developmental cellular system* was presented. The results obtained demonstrated that qualities of robustness to faults and fully asynchronous mode could be obtained with such systems. It demonstrated also that development was a way to encode a more general solution within a simpler genome. However, the mitigated results on more complex tasks showed that such properties were not to be found systematically in this system. The main contribution is thus to have *paved the path for future developments of ontogenetic cellular systems.*

- *A scaling method for non-uniform CAs.* Non-uniform CAs have proven in the past more powerful than uniform ones, at no extra cost in terms of "hardware", while being faster to evolve. The principal critique concerning evolution of non-uniform CAs were their non-adaptability to different sizes than the one they were evolved for. Empirically, I demonstrated that the method proposed allowed *adaption to all sizes* (above a minimum size).

- *A definition of emergent global behavior in the scope of non-uniform CAs.* The discussion about the validity of this scaling method by a dialectical reversal led me to propose a definition of emergent global behavior, restricted to the scope of computation by non-uniform CAs. The main conclusions were that such a behavior must, obviously, result from local interactions and be global, but moreover it should be total, in the sense that *the result is the state of the system as a whole*, that all elements in the grid were inputs and outputs, and that the result holds for any system size.

- *Proofs concerning the density task.* In this thesis I proved a series of results about the density task in its various forms. To wit, the proof of *the impossibility to find a non-uniform CA to solve perfectly the density task in its original form.* This proof was important in two ways. First, it relativized the power of non-uniform CAs compared to uniform CAs. Second, and more importantly, it showed that the density task as originally defined was utterly impossible. It was also demonstrated in this thesis that this impossibility was very relative as *a different definition* on the form but not in the essence of the task allowed me to find *a CA solving perfectly* in one of the simplest class of CAs. This led me to prove necessary conditions to solve the density task, mainly that *a density classifying CA should be density conserving.* A last proof showed that a newly proposed task was as impossible as the density task.

- *A proposition of what is computation in CAs: Visual efficiency.* Please refer to the first part of this chapter for a detailed exposition, but essentially I argued that problem-solving CAs were *the fortunate meeting of a good CA with the correct perspective from the beholder*, and thus that they relied on being visually efficient.

- *An analysis of the Cellular Programming Algorithm.* In this analysis, I showed a number of features of each task, demonstrating the traps in which this algorithm may fall. Most notably, common to two of the three specific tasks studied, *the algorithm revealed a tendency to uniform the grid, thus losing diversity.* This was confirmed by the behavior of the algorithm on the control task, tending to show that the algorithm always creates large blocks. *These conclusions may help in the design of future fine grained parallel evolutionary algorithms.*

- *Different methods for Asynchronous Cellular Automata* were presented. Essentially, it was shown that *a simple perfect method* existed that fully exploited the inherent parallelism of CA, decreasing the relative lateness of the automata as the probability of synchronization faults augments. Though this method is perfect, it is not the best in all respects. That's why I then presented an *imperfect method* particular to the synchronization task which performs well at a much lower cost in terms of necessary states per cell, and with no lateness. Finally, *I evolved redundant CAs* and found low-cost, timely efficient solutions this way.

- *Reevaluation of what is the global state of a CA.* Please see the first part of this chapter for a detailed exposition, but, briefly, I concluded from the preceding study of asynchronous CAs that the information contained in a CA was *the state and the space but also the time*, and that redundant CAs were absolutely necessary if one was to find good asynchronous CA through evolution.

- *A low-cost fault-tolerant CA.* The previous conclusion led naturally to observe that extra states were necessary to correct faulty CAs. Noticing that usual CA constructs like loops leave many neighborhoods unused (the equivalent of many extra states), I derived *a simple method, based on error-correcting code, to design fault-tolerant constructs.*

Future work

This thesis, like any PhD thesis, calls for more work, numerous ideas having sprouted during the realization (and the writing). For instance, the present results of the developmental model calls for the design of a new system taking into account the conclusions drawn from the current model. This is a long term goal as I believe that any truly ontogenetic system should rely on a cellular approach and that we need first a better understanding of cellularity.

The main work to come, the one that is the most urgent in my view, is the artificial observer, and this in many ways. First, there are many studies to conduct to determine what is visually efficiency, e.g., how to measure irregularity vs. regularity. Second, we need to study the evolution of a CA when the fitness criteria are not given anymore objectively,

strictly fixed criteria, but rather by this artificial observer. Thirdly, another way to go, or a joint way to go, is to co-evolve an observer and a CA, thereby resulting in a coupled pair observer/CA. This latter kind of study may provide interesting results on the observer point of view. Nevertheless, these are all mid term projects, and first, one should try out these ideas on the density task in two dimensions, to test if it is possible to find a solution which would run in $O(\sqrt{n})$ time.

The results on the fault-tolerant loop should be completed, both out of curiosity, to know in how many states one could code such a loop, but also more generally. In effect, one should try to establish an algorithmic way to develop such structures, concentrating on the question of the number of neighborhoods available. It would be interesting to explore the development of an algorithm based on artificial evolutionary techniques to find, given a series of shapes in a quiescent sea in time, a suite of neighborhoods at a Hamming distance greater than three.

Finally, it would be interesting to develop a new evolutionary algorithm for non-uniform CAs that could balance the tendency to create blocks. A tendency which obviously we do not want to eliminate completely, as it is the basis of our scaling algorithm, but that is not good in itself, preventing any solution containing a large number of blocks to be found.

As one may see, the future paths are very much centered around the evolutionary techniques. This is due to two facts. First, we are still left with no mathematical way to map local to global behavior. Second, as highlighted several times throughout this thesis, artificial evolution matches inherently this structure global/local. However, as shown, such development should be based around studies of the cellular system in itself to avoid blind evolution.

Bibliography

[1] Andrew Adamatzky. Simulation of inflorescence growth in cellular automata. *Chaos, Solitons and Fractals*, 7(7):1065–1094, 1996.

[2] Andrew Adamatzky. *Computing in Nonlinear Media & Automata Collectives*. Institute of Physics Publishing, 2001.

[3] Leonard M. Adleman. Molecular computation of solutions to combinatorial problem. *Science*, 266:1021–1024, November 1994.

[4] Pankaj Agarwal. *The cell programming language*. Courant Institute of Mathematical Sciences, Computer Science Dept, New-York University, 1993. Technical Report TR630.

[5] Lee Altenberg. The schema theorem and price's theorem. In Darrel Whitley and Michael Vose, editors, *Foundations Of Genetic Algorithms 3*, San Francisco, CA, 1995. Morgann Kaufmann Publishers. completed version of August 15, 2001.

[6] Serafino Amoroso and Gerald Cooper. Tessellation structures for reproduction of arbitrary patterns. *Journal of Computer and System Science*, 5:455–464, 1971.

[7] Serafino Amoroso and Y. N. Patt. Decision procedures for surjectivity and injectivity of parallel maps for tessellation structures. *Journal of Computer and System Science*, 10:77–82, 1975.

[8] David Andre, Forrest H. Bennet III, and John R. Koza. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. In John R. Koza, David E. Goldberg, David B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the first conference*, pages 3–11, Cambridge, MA, 1996. The MIT Press.

[9] David Andre and John R. Koza. Parallel genetic programming: A scalable implementation using the transputer network architecture. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming 2*, Cambridge, MA, 1996. The MIT Press.

[10] Peter J. Angeline. Subtree crossover causes bloat. In J.R:. Koza, K. Deb, M. Dorigo, D.B. Fogel, M.H. Gazon, and R.L. Riolo, editors, *Genetic Programming 1998: Proceedings of the third annual conference*. Morgan Kaufmann, 1998.

[11] Peter J. Angeline and J. B. Pollack. Co-evolving high-level representations. In C. G. Langton, editor, *Artificial Life III*, pages 55–71, Reading, MA, 1994. Addison-Wesley.

[12] Michael A. Arbib. Simple self-reproducing universal automata. *Information and Control*, 9(2):177–189, 1966.

[13] Robert Axelrod. Advancing the art of simulation in the social sciences. In R. Conte, Hegselmann, and P. Terna, editors, *Simulating Social Phenomena*, pages 21–40, Berlin, 1997. Springer.

[14] Nils A. Baas and Claus Emmeche. On emergence and explanation. *Intellectica*, 25:67–83, 1997.

[15] Jonathan B. L. Bard. A model for generating aspects of zebra and other mammalian coat patterns. *Journal of Theoretical Biology*, 93, 1981.

[16] Vladimir Belitsky and Pablo A. Ferrari. Ballistic annihilation and deterministic surface growth. *Journal of Statistical Physics*, 80(3/4):517–543, 1995.

[17] Miguel Benasayag. *Le mythe de l'individu.* La Découverte, Paris, 1998.

[18] Elwin R. Berkelamp, John H. Conway, and R. K. Guy. *Winning ways for your mathematical plays.* Academic Press, 1982.

[19] Hugues Bersini and Vincent Detours. Asynchrony induces stability in cellular automata based models. In R.A. Brooks and P. Maes, editors, *Proceedings of the Artificial Life IV conference*, pages 382–387, Cambridge, MA, 1994. MIT Press.

[20] Tobias Blickle and Lothar Thiele. Subtree crossover causes bloat. In J. Hopf, editor, *Genetic Algorithm within the Framework of Evolutionary Computation (KI-94 worksjop, Saarbrücken)*, pages 33–38. Max-Planck Institut für Informatik, 1994.

[21] Nino Boccara and Henry Fukś. Number-conserving cellular automaton rules. *Fundamenta Informaticae*, To Appear, 2001/2002?

[22] Lashon B. Booker, David B. Fogel, Darell Whitley, Peter J. Angeline, and A. E. Eiben. Ch. 33: Recombination. In Thomas Baeck, Fogel David B, and Zbigniew Michalewicz, editors, *Evolutionary Computation 1*, pages 256–307, Bristol, UK and Philadelphia, USA, 2000. Institute of Physics Publishing.

[23] Arthur W. Burks, editor. *Essays on Cellular Automata.* University of Illinois Press, Urbana, IL., 1970.

[24] Erik Cantú-Paz. A summary of research on parallel genetic algorithms. Technical Report 95007, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL, July 1995.

[25] Erik Cantú-Paz and David E. Goldberg. Modeling idealized bounding cases of parallel genetic algorithms. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 353–361, San Francisco, 1997. Morgan Kaufmann Publishers.

[26] Mathieu S. Capcarrere, Moshe Sipper, and Marco Tomassini. Two-state, r=1 cellular automaton that classifies density. *Physical Review Letters*, 77(24):4969–4971, December 1996.

[27] Mathieu S. Capcarrere, Andrea Tettamanzi, Marco Tomassini, and Moshe Sipper. Statistical study of a class of cellular evolutionary algorithms. *Evolutionary Computation*, 7(3):255–274, 1998.

[28] Bastien Chopard and Michel Droz. *Cellular Automata Modeling of Physical Systems.* Cambridge University Press, 1998.

[29] Hui-Sien Chou and James A. Reggia. Problem solving during artificial selection of self-replicating loops. *Physica D*, 115(3):293–312, 1998.

[30] Cioran. *Oeuvres.* Quarto. Gallimard, Paris, 1995(1952).

[31] Edgar F. Codd. *Cellular Automata.* Academic Press, New-York, NY, 1968.

[32] James P. Cohoon, S. U. Hedge, Worthy N. Martin, and Dana S. Richards. Punctuated equilibria: A parallel genetic algorithm. In John J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, page 148. Lawrence Erlbaum Associates, 1987.

[33] Robert Cori, Yves Métivier, and Wieslaw Zielonka. Asynchronous mappings and asynchronous cellular automata. *Information and Computation*, 106:159–202, 1993.

[34] Nichael J. Cramer. A representation of the adaptive generation of simple sequential program. In John J. Grefenstette, editor, *Proceedings of the 1st Internatinal Conference on Genetic Algorithms (Pittsburgh, PA, July 1985)*, Hillsdale, NJ, 1985. Erlbaum.

[35] James P. Crutchfield and Melanie Mitchell. The evolution of emergent computation. *Proceedings of the National Academy of Science*, 23(92):10742, 1995.

[36] Karel Culik II, Lyman P. Hurd, and Sheng Yu. Computation theoretic aspects of cellular automata. *Physica D*, 45:357–378, 1990.

[37] Karel Culik II, Lyman P. Hurd, and Sheng Yu. Formal languages and global cellular automata. *Physica D*, 45:396–403, 1990.

[38] Karel Culik II and Sheng Yu. Undecidability of ca classification schemes. *Complex Systems*, 2:177–190, 1988.

[39] Rajarshi Das, James P. Crutchfield, Melanie Mitchell, and James E. Hanson. Evolving globally synchronized cellular automata. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 336–343, San Francisco, CA, 1995. Morgan Kaufmann.

[40] Frank Dellaert. *Toward a Biologically Defensible Model of Development.* Dept of Computer Engineering and science, Case Western Reserve University, 1995. PhD Thesis.

[41] Marianne Delorme and Jacques Mazoyer. Cellular automata as language recognizers. In M. Delorme and J. Mazoyer, editors, *Cellular Automata: A parallel model.* Kluwer Acadamecic Publishers, 1999.

[42] D. Z. Du, D. F. Hsu, and F. K. Hwang. The hamiltonian properties of consecutive-d digraphs. *Math. Comput. Modelling*, 17(11):61–63, 1993.

[43] R. Durikovic, K. Kaneda, and H. Yamashita. Animation of biological organ growth based on l-systems. *Computer graphics forum: the international journal of the Eurographics Association*, 17(3):C1–C13, 1998.

[44] Charles Dyer. One-way cellular automata. *Information and Control*, 44:54–69, 1980.

[45] Claus Emmeche, Simo Køppe, and Frederik Stjernfelt. Explaining emergence: Towards an ontology of levels. *Journal for General Philosophy of Science*, 28:83–119, 1997.

[46] G. Bard Ermentrout and Leah Edestein-Keshet. Cellular automata approaches to biological modeling. *Journal of Theoretical Biology*, 160:97–133, 1993.

[47] Jörg Esser and Michael Schreckenberg. Microscopic simulation of urban traffic based on cellular automata. *International Journal of Modern Physics C*, 8(5):1025–1036, October 1997.

[48] Henri Féraud, Pierre et Maria Sire, Joë Bousquet, Claire-Charles Géniaux, Albert Béguin, and Louis prat. Soirée languedocienne: Entretien dans la cité. In Joë Bousquet, Jean Ballard, René Nelli, P.M. Sire, and Henri Féraud, editors, *Le Génie d'Oc et l'homme méditerranéen*, pages 390–405. Les Cahiers du Sud, 1943. (retranscrit par Jean Ballard).

[49] Robert Fisch. Cyclic cellular automata and related processes. *Physica D*, 45:19–25, 1990.

[50] E. J. Fittkau and H. Klinge. On biomass and trophic structure of the central amazonian rain forest ecosystem. *Biotropica*, 5:2–14, 1973.

[51] Kurt Fleischer. *A Multiple Mechanism Developmental Model for Defining Self-Organing Structures*. California Institute of Technology, 1995. PhD Thesis.

[52] Kurt Fleischer and A. H. Barr. A simulation testbed for the study of multicellular development: The multiple mechanisms of morphogenesis. In C. G. Langton, editor, *Artificial life III*, volume XVII of *SFI Studies in the Sciences of Complexity*. Addison-Wesley, 1994.

[53] Gary B. Fogel and David B. Fogel. Continuous evolutionary programming: analysis and experiments. *Cybernetic systems*, 26:79–90, 1995.

[54] Lawrence Fogel. Autonomous automata. *Industrial Reasearch*, 4:14–19, 1962.

[55] Lawrence J. Fogel, Alvin J. Owens, and Michael J. Walsh. *Artificial intelligence through Simulated Evolution*. John-Wiley, New-York, N.Y., 1966.

[56] Richard Forsyth. BEAGLE A Darwinian approach to pattern recognition. *Kybernetes*, 10:159–166, 1981.

[57] E. Franji, M. Dascalu, and M. Stanescu. Cellular automata architecture for the artificial insect eye. In *Proceedings of NC 1998: International ICSC/IFAC Symposium on Neural Computation*, pages 664–668. Academic Press, Zurich, Switzerland, 1998.

[58] Dan R. Frantz. *Non-linearities in Gnetic Adaptive Search*. University of Michigan: Ann Arbor, 1972. PhD Thesis.

[59] A. S. Fraser. Simulation of genetic systems by automatic digital computers. *Australian Journal of Biological Sciences*, 10:484–499, 1957.

[60] R. M. Friedberg. A learning machine: part i. *IBM Journal*, 2:2–13, 1958.

[61] Henry Fúks. Exact results for deterministic cellular automata traffic models. *Physical Review E*, 60(1):197–202, 1999.

[62] Chikara Furusawa and Kunihiko Kaneko. Emergence of multicellular organisms with dynamic differentiation and spatial pattern. *Artificial Life*, 4:79–93, 1998.

[63] Peter Gács. Self-correcting two-dimensionnal arrays. In Silvio Micali, editor, *Randomness in computation*, volume 5 of *Advances in Computing Research*, pages 223–326, Greenwich, Conn, 1989. JAI Press.

[64] Peter Gács. Reliable cellular automata with self-organization. In *Proceedings of the 38th IEEE Symposium on the Foundation of Computer Science*, pages 90–99, 1997.

[65] Peter Gács. Reliable cellular automata with self-organization. *Journal of Statistical Physics*, 103(1/2):45–268, 2001.

[66] Peter Gacs, G. L. Kurdyumov, and L. A. Levin. One-dimensional uniform arrays that wash out finite islands. *Problemy Peredachi Informatsii*, 14:92–98, 1978.

[67] Martin Gardner. The fantastic combinations of john conway's new solitaire game 'life'. *Scientific American*, 223(4):120–123, July 1970.

[68] Hugo de Garis. Artificial embryology and cellular differentaition. In Peter Bentley, editor, *Evolutionary Design by Computers*, pages 281–295, San Francisco, CA, 1999. Morgan Kaufmann.

[69] H. Gerola and P. Seiden. Stochastic star formation and spiral structures of galaxy. *Astrophysical Journal*, 223:129, 1978.

[70] R. Gordon. The emergence of emergence: A critique of "design, observation, surprise!". *Rivista di Biologia-Biology Forum*, 93(2):346–356, 2000.

[71] Lawrence F. Gray. A reader's guide to gacs's "positive rates" paper. *Journal of Statistical Physics*, 103(1/2):1–44, 2001.

[72] J. M. Greenberg, B. D. Hassard, and S. P. Hastings. Pattern formation and periodic structures in systems modelled by reaction-diffusion equations. *Bulletin of The American Mathematical Society*, 84:1296, 1978.

[73] John J. Grefenstette and James Edward Baker. How genetic algorithms work: a critical look at implicit parallelism. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 20–27. Morgan Kaufmann, 1989.

[74] Frederic Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD Thesis, LIP-Ecole Normale Supèrieure de LYON, 1994.

[75] Howard A. Gutowitz and Christopher Langton. Methods for designing "interesting" cellular automata. *CNLS News Letter*, 1988. may be found at `http://www.santafe.edu/~hag/interesting/interesting.html`.

[76] Howard A. Gutowitz and Christopher Langton. Mean field theory of the edge of chaos. In F. Morán, A. Moreno, J. J. Merelo, and P. Chacon, editors, *Advances in Artificial Life, Proceedings of the 3rd European Conference on Artificial Life*, volume 929 of *LNAI*, pages 52–64. Springer-Verlag, 1995.

[77] James E. Hanson and James P. Crutchfield. Computational mechanics of cellular automata. *Physica D*, 103:169–189, 1997.

[78] Masaretu Harao and Shoici Noguchi. Fault tolerant cellular automata. *Journal of computer and system sciences*, 11:171–185, 1975.

[79] H. Hartman and Gérard Y. Vichniac. Inhomogeneous cellular automata (inca). In E. Bienenstock et al., editor, *Disordered Systems and Biological Organization*, volume F 20, pages 53–57. Springer-Verlag, Berlin, 1986.

[80] G. A. Hedlund. Endomorphism and automorphism of shift dynamical systems. *Mathematical Systems Theory*, 3:51–59, 1969.

[81] G. A. Hedlund, K. I. Appel, and Welch L. R. All onto functions of span less than or equal to five. *Communications Research Division*, JULY 1963. working paper.

[82] G. T. Herman. On universal computer-constructors. *Information Processing Letters*, 2:61–64, 1973.

[83] John Holland. Universal species: A basis for studies in adaptation. *Automata Theory*, pages 218–230, 1966.

[84] John H. Holland. *Nonlinear environment permitting efficient adaptation.* Academic Press, New-York, N.Y., 1967.

[85] John H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* MIT Press/Bradford book editions, 1992(1975).

[86] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory Languages and Computation.* Addison-Wesley, Redwood City, CA, 1979.

[87] Wim Hordijk. The structure of the synchonizing-ca landscape. Technical Report 96-10-078, Santa Fe Institute, Santa Fe, NM (USA), 1996.

[88] Wim Hordijk. *Dynamics, Emergent Computation and Evolution in Cellular Automata.* Computer Science Dept, University of New Mexico, Albuquerque, NM (USA), Dec. 1999.

[89] Peter D. Hortensius, Robert D. McLeod, and Howard C. Card. Parallel random number generation for VLSI systems using cellular automata. *IEEE Transactions on Computers*, 38(10):1466–1473, October 1989.

[90] Peter T. Hraber, Terry Jones, and Stephanie Forrest. The ecology of echo. *Artificial Life Journal*, 3:165–190, 1997.

[91] Oscar Ibarra. Computational complexity of cellular automata: an overview. In M. Delorme and J. Mazoyer, editors, *Cellular Automata: A parallel model.* Kluwer Acadamecic Publishers, 1999.

[92] T. E. Ingerson and R. L. Buvel. Structures in asynchronous cellular automata. *Physica D*, 10:59–68, 1984.

[93] Yasusi Kanada. Asynchronous 1d cellular automata and the effects of fluctuation and randomness. In R.A. Brooks and P. Maes, editors, *A-Life IV: Proceedings of the Fourth Conference on Artificial Life*, page Poster, Cambridge, MA, 1994. MIT Press.

[94] Emmanuel Kant. *Idée d'une histoire universelle au point de vue cosmopolitique.* Bordas, Paris, 1988. Translation J.-M. Muglioni. Notes: 1) The translation of this sentence in this edition is slightly different from the one quoted here. 2) Of course, this sentence is taken in the opposite of its original meaning, Kant believing in a teleological nature.

[95] James Kennedy and Rusell C. Eberhart (with Yuhui Shi). *Swarm Intelligence.* Morgan Kaufmann Publishers, 2001.

[96] Kenneth E. Kinnear, editor. *Advances in Genetic Programming.* MIT Press, Cambridge, MA., 1994.

[97] Hiroaki Kitano. A simple model of neurogenesis and cell differentiation based on evolutionary large-scale chaos. *Artificial Life*, 2(1):79–99, 1995.

[98] Hiroaki Kitano. Building complex system using a developmental process: An engineering approach. In *Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware (ICES98).* Springer-Verlag, 1998.

[99] John R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the 11th Internatinal Joint Conference on Artificial Intelligenge*, San Mateo, CA, 1989. Morgan Kaufmann.

[100] John R. Koza. *Genetic Programming: On The Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, 1992.

[101] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge, MA, 1994.

[102] John R. Koza and David Andre. Classifying protein segments as transmembrane domains using architecture-altering operations in genetic programming. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming 2*, pages 155–176, Cambridge, MA, 1996. The MIT Press.

[103] Dietrich Kuske. Emptiness is decidable for asynchronous cellular machines. In C. Palamidessi, editor, *CONCUR 2000*, Lecture Notes in Computer Science, LNCS 1877, pages 536–551, Berlin, 2000.

[104] Richard Laing. Artificial organisms and autonomous cell rules. *Journal of Cybernetics*, 2(1):38–49, 1972.

[105] Richard Laing. Automaton introspection. *Journal of Computer and System Sciences*, 13:172–183, 1976.

[106] Mark Land and Richard K. Belew. No perfect two-state cellular automata for density classification exists. *Physical Review Letters*, 74(25):5148–5150, June 1995.

[107] William B. Langdon. Size fair and homologous tree genetic programming crossovers. In W. Banzhaf, j: Daida, A.E. Eiben, M.H: Garzon, V. Honavar, M. Jakiela, and R.E. smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)*, volume 2, pages 1092–1097. Morgan Kaufmann, 1999.

[108] William B. Langdon. Quadratic bloat in genetic programming. In Darrel Whitley, David Goldberg, Erick Cantú-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors, *Genetic and Evolutionary Computing COnference'00: The proceedings of*, pages 451–458, San Francisco, CA, July 2000. Morgan Kaufmann.

[109] William B. Langdon and Riccardo Poli. Fitness causes bloat. In R. Chawdhry, P.K.and Roy and R.K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22, London, 1997. Springer-Verlag.

[110] J. S. Langer. Instabilities and pattern formation in crystals growth. *Review of Modern Physics*, 52:1, 1980.

[111] Christopher G. Langton. Computation at the edge of chaos: Phase transitions and emergent computation. *Physica D*, 42:12–37, 1984.

[112] Christopher G. Langton. Self-reproduction in cellular automata. *Physica D*, 10:135–144, 1984.

[113] Christopher G. Langton. Studying artificial life with cellular automata. *Physica D*, 22:120–149, 1986.

[114] Dong Wook Lee and Kwee Bo Sim. Ontogenesis of artificial neural networks based on l-system and genetic algorithms. In *Proceedings of the 5th International Conference on Soft Computing and Information/Intelligent Systems. Methodologies for the Conception, Design and Application of Soft Computing*, pages 817–820, Singapore, 1998. World Scientific Publishing Comp.

[115] K. M. Lee, Hao Xu, and H. F. Chau. Parity problem with a cellular automaton solution. *Physical Review E*, 64:026702, July 2001. The page number refers to the official internet numbering scheme of the American Physical Society.

[116] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer-Verlag, New-York, 1993.

[117] Wentian Li. Phenomenology of non-local cellular automata. *Journal of Statistical Physics*, 68:829–882, 1992.

[118] Wentian Li and Norman Packard. The structure of elementary cellular automata rule space. *Complex Systems*, 4:281–297, 1990.

[119] Aristid Lindenmayer. Mathematical models for cellular interaction in development,part i and ii. *Journal of Theoretical Biology*, 18, 1968.

[120] K. Lindgren and M. G. Nordahl. Universal computation in simple one-dimensional cellular automata. *Complex Systems*, 4:299–318, 1990.

[121] Shida Liu and Ziguo Zhang. Simulation of snowflake by cellular automata. *Scientia-Atmospherica-Sinica*, 13(2):193–198, 1989.

[122] Andrea Loraschi, Andrea Tettamanzi, Marco Tomassini, and P. Verda. Distributed genetic algorithms with an application to portfolio selection problems. In *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 384–387. Springer-Verlag, New-York, 1995.

[123] Sean Luke. *Issues in Scaling Genetic Programming: Breeding Strategies, tree Generation, and Code Bloat*. University of Maryland, 2000. PhD Thesis.

[124] Bernard Manderick and Piet Spiessens. Fine-grained parallel genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, page 428. Morgan Kaufmann, 1989.

[125] Daniel Mange, Moshe Sipper, Andre Stauffer, and Gianluca Tempesti. Towards robust integrated circuits: The embryonics approach. *Proceedings of the IEEE*, 88(4):516–541, April 2000.

[126] Karl Marx. *Philosophie*. Number 244 in Folio. Gallimard, 2000.

[127] Jacques Mazoyer and N. Reimen. A linear speed-up theorem for cellular automata. *Theoretical Computer Science*, 50(2):183–238, 1992.

[128] Barry McMullin. John von neumann and the evolutionary growth of complexity: Looking backwards, looking forwards... In Mark A. Bedau, John S. McCaskill, Norman H. Packard, and Steen Rasmussen, editors, *Artificial Life VII: Proceedings of the seventh international conference*, pages 467–476, Cambridge, MA., 2000. MIT Press.

[129] Nicholas Freitag McPhee and Justin Darwin Miller. Accurate replication in genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the sixth international conference (ICGA'95)*, pages 303–309. Morgan Kaufmann, 1995.

[130] Hans Meinhardt. *Models of Biological Pattern Formation*. Academic Press, london, 1982.

[131] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.

[132] Melanie Mitchell, James P. Crutchfield, and Peter T. Hraber. Evolving cellular automata to perform computations: Mechanisms and impediments. *Physica D*, 75:361–391, 1994.

[133] Melanie Mitchell, Peter T. Hraber, and James P. Crutchfield. Revisiting the edge of chaos: Evolving cellular automata to perform computations. *Complex Systems*, 7:89–130, 1993.

[134] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[135] Heinz Mühlenbein. Evolution in time and space–the parallel genetic algorithm. In Gregory J. E. Rawlins, editor, *Foundations Of Genetic Algorithms I*. Morgann Kaufmann Publishers, 1991.

[136] James D. Murray. On pattern formation mechanisms for lepidoptera wing patterns and mammalian coat markings. *Philosophical Transactions of the Royal Society (B)*, 295, 1981.

[137] Kai Nagel and Hans J. Hermann. Deterministic model for traffic jams. *Physica A*, 199:254–269, 1993.

[138] Kai Nagel, Dietrich E. Wolf, Peter Wagner, and Patrice Simon. Two-lane traffic rules for cellular automata: A systematic approach. *Physical Review E*, 58(2):1425–1437, August 1998.

[139] John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Illinois, 1966. Edited and completed by A. W. Burks.

[140] Hidenosuke Nishio and Youichi Kobuchi. Fault tolerant cellular spaces. *Journal of computer and system sciences*, 11:150–170, 1975.

[141] Peter Nordin. A compiling genetic programming system that directly manpulates the machine code. In K. E Kinnear Jr, editor, *Advances in Genetic Programming*, Cambridge, MA, 1994. MIT Press.

[142] Peter Nordin and Wolfgang Banzhaff. Complexity, compression adn evolution. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the sixth international conference (ICGA'95)*, pages 310–317. Morgan Kaufmann, 1995.

[143] Martin A. Nowak, Sebastian Bonhoeffer, and Robert M. May. Spatial games and the maintenance of cooperation. *Proceedings of the National Academic of Sciences USA*, 91:4877–4881, May 1994.

[144] G. M. Odell, G. Oster, P. Albrech, and B. Burnside. The mechanical basis of morphogenesis. *Developmental Biology*, 85, 1981.

[145] G. M. B. Oliveira, P. P. B. De Oliveira, and N. Omar. Searching for one-dimensional cellular automata, in the absence of *a priori* information. In Jozef Kelemen and Petr Sosík, editors, *Advances in Artificial Life: Proceedings of the sixth European Conference on Artificial Life (ECAL'01)*, Lecture Notes in Artificial Intelligence 2159, pages 262–271. Springer-Verlag, 2001.

[146] M. Oussaidene, Bastien Chopard, Olivier Pictet, and Marco Tomassini. Parallel genetic programming and its application to trading model induction. *Parallel Computing*, 23:1183–1198, 1997.

[147] Norman H. Packard. Adaptation toward the edge of chaos. In J. A. S. Kelso, A. J. Mandell, and M. F. Shlesinger, editors, *Dynamic Patterns in Complex Systems*, pages 293–301. World Scientific, Singapore, 1988.

[148] Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

[149] Blaise Pascal. *Pensées*. Folio/Gallimard, 1977. Edition de Michel le Guern.

[150] Jean-Yves Perrier, Moshe Sipper, and Jacques Zhand. Toward a viable, self-reproducing universal computer. *Physica D*, 97:335–352, 1996.

[151] Umberto Pesavento. An implementation of von neumann's self-reproducing machine. *Artificial Life*, 2(4):337–354, 1995.

[152] Enrico Petraglio, Jean-Marc Henry, and Gianluca Tempesti. Arithmetic operations on self-replicating cellular automata. In D. Floreano, J.-D. Nicoud, and F. Mondada, editors, *Advances in Artificial Life, Proceedings of the 5th European Conference on Artificial Life*, pages 447–456. SpringerVerlag, 1999.

[153] Giovanni Pighizzini. Asynchronous automata versus asynchronous cellular automata. *Theoretical Computer Science*, 132:179–207, 1994.

[154] Plotin. *Ennéades*. Les Belles Lettres, Paris, 1970.

[155] Riccardo Poli. Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover. *Genetic Programming and Evolvable Machines*, 2(2):123–163, 2001.

[156] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New-York, N.Y., 1990.

[157] François Rabelais. *Les Cinq Livres – Gargantua/Pantagruel/Le Tiers Livre/Le Quart Livre/Le Cinquième Livre*. Classiques Modernes/La Pochothèque. Le Livre de Poche/Librairie Générale Française, 1994. Edition critique de Jean Céard, Gérard Defaux, et Michel Simonin.

[158] Nicholas Radcliffe. Equivalence class analysis of genetic algorithms. *Complex Systems*, 5(2):183–205, 1991.

[159] Ingo Rechenberg. *Cybernetic Solution Path of an experiemntal Problem*. Royal Aircraft Establishment Library Transslation, 1965. 1122.

[160] Ingo Rechenberg. *Evolutionstrategis: Optimierung technisher Systeme nach Prinzipien der biologischen Evolution*. Fromman-Holzboog Verlag, Stuttgart, 1973. Evolution strategy: the optimization of technical systems according to the principles of biological evolution.

[161] James A. Reggia, Hui-Sien Chou, and Jason D. Lohn. Cellular automata models of self-replicating systems. In Marshall C. Yovits, editor, *Advances in Computers*, volume 47. Academic Press, 1998.

[162] James A. Reggia, Hui-Sien Chou, and Jason D. Lohn. Self-replicating structures : Evolution, emergence and computation. *Artificial Life*, 4:283–302, 1998.

[163] D. Richardson. Tessellation with local transformations. *Journal of Computer and System Science*, 6:373–388, 1972.

[164] Edmund Ronald, Moshe Sipper, and Mathieu S. Capcarrère. A test of emergence, design, observation, surprise! *Artificial Life*, 5(3):225–239, 1999.

[165] Justinian P. Rosca and Dana H. Ballard. Discovery of subroutines in genetic programming. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming 2*, pages 177–202, Cambridge, MA, 1996. The MIT Press.

[166] Günter Rudolph and Joachim Sprave. A cellular genetic algorithm with self-adjusting acceptance threshold. In *First IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, pages 365–372, London, 1995. IEE.

[167] Steven J. Ruuth and Barry Merriman. Convolution-thresholding methods for interface motion. *Journal of Computational Physics*, 169(2):678–707, May 2001.

[168] Palash Sarkar. A brief history of cellular automata. *ACM Computing Surveys*, 32(1):80–107, March 2000.

[169] V. V. Savchenko, A. G. Basnakian, and A. A. Pasko. Computer simulation and analysis of a growing mammalian cell colony. *Lectures in Mathematics in the Life Sciences*, 26:111–120, 1999.

[170] Hiroaki Sayama. *Constructing evolutionary systems on a simple deterministic cellular automata space.* Dept of Information Science, University of Tokyo, December 1998. PhD Thesis.

[171] Birgitt Schönfisch and André de Roos. Synchronous and asynchronous updating in cellular automata. *BioSystems*, 51:123–143, 1999.

[172] Hans-Peter Schwefel and G. Rudolph. Contemporary evolution strategies. In F. Morana et al, editor, *Advances in Artificial Life*, pages 893–907, Berlin, 1995. Springer-Verlag.

[173] Claude E. Shannon. *A mathematical theory of communication.* University of Illinois Press, 1949.

[174] Oliver Sharpe. Continuing beyond nfl: dissecting real world problems. In W. Banzhaf, J. Daida, A.E. Eiben, M.H: Garzon, V. Honavar, M. Jakiela, and R.E. smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)*, volume 2, pages 1074–1381. Morgan Kaufmann, 1999.

[175] Hans B. Sieburg, J. Allen McCutchan, Oliver K. Clay, Lisa Cabalerro, and James J. Ostlund. Simulation of hiv infection in artificial immune systems. *Physica D*, 45:208–227, 1990.

[176] Oliver K. Sieburg, Hans B. Clay. The cellular device machine development system for modeling biology on the computer. *Complex Systems*, 5(6):575–601, 1991.

[177] Karl Sims. Evolving 3d morphology and behavior by competition. In R.A. Brooks and P. Maes, editors, *A-Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, Cambridge, MA, 1994. MIT Press.

[178] Moshe Sipper. Studying artificial life using a simple, general cellular model. *Artificial Life Journal*, 2(1):1–35, 1995.

[179] Moshe Sipper. Co-evolving non-uniform cellular automata to perform computations. *Physica D*, 92:193–208, 1996.

[180] Moshe Sipper. *Evolution of Parallel Cellular Machines: The Cellular Programming Approach.* Springer-Verlag, Heidelberg, 1997.

[181] Moshe Sipper. The evolution of parallel cellular machines: Toward evolware. *BioSystems*, 42:29–43, 1997.

[182] Moshe Sipper. Computing with cellular automata: Three cases for nonuniformity. *Physical Review E*, 57(3), March 1998.

[183] Moshe Sipper. Fifty years of research on self-replication: An overview. *Artificial Life*, 4:237–257, 1998.

[184] Moshe Sipper, Mathieu S. Capcarrère, and Edmund Ronald. A simple cellular automaton that solves the density and ordering problems. *International Journal of Modern Physics C*, 9(7):899–902, October 1998.

[185] Moshe Sipper and Ethan Ruppin. Co-evolving architectures for cellular machines. *Physica D*, 99:428–441, 1997.

[186] Moshe Sipper and Marco Tomassini. Co-evolving parallel random number generators. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 950–959. Springer-Verlag, Heidelberg, 1996.

[187] Moshe Sipper and Marco Tomassini. Generating parallel random number generators by cellular programming. *International Journal of Modern Physics C*, 7(2):181–190, 1996.

[188] Moshe Sipper, Marco Tomassini, and Olivier Beuret. Studying probabilistic faults in evolved non-uniform cellular automata. *International Journal of Modern Physics C*, 7(6):923–939, 1996.

[189] S.F. Smith. *A Learning System Based on Genetic Adaptive Algorithms.* Computer Science Dept, University of Pittsburg, 1980. PhD Dissertation.

[190] Alvy Ray Smith III. Cellular automata complexity trade-offs. *Information and Control*, 18:223–253, 1971.

[191] William Somerset Maughan. *The moon and six pence.* Dover Thrift, 2000.

[192] Terence Soule and J.A. Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *1998 IEEE Conference on Evolutionary Computation*, pages 781–786, New-York, N.Y., USA, 1998. IEEE Press.

[193] Lee Spector. Simultaneous evolution of programs and their control structures. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming 2*, pages 137–154, Cambridge, MA, 1996. The MIT Press.

[194] Lee Spector and Killian Stoffel. Automatic generation of adaptive programs. In P. Maes, M. Mataric, J.-A. Meyer, J. Pollack, and S.W. Wilson, editors, *From Animals to Animat 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (SAB'96)*, pages 476–483, Cambridge, CA, 1996. MIT Press.

[195] W. Richard Stark. Dynamics for fundamental problem of biological information processing. *International Journal of Artificial Intelligence Tools*, 4(4):471–488, 1995.

[196] T. Starkweather, Darrell Whitley, and K. Mathias. Optimization using distributed genetic algorithms. In H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature*, volume 496 of *Lecture Notes in Computer Science*, page 176, Heidelberg, 1991. Springer-Verlag.

[197] Luc Steels. The artificial intelligence roots of artificial life. *Artificial Life*, 1(1/2):75–110, 1994.

[198] Kenneth Steiglitz, R. K. Squier, and M. H. Jakubow. Programmable parallel arithmetic in cellular automata using a particle model. *Complex Systems*, 8:311–323, 1994.

[199] Klaus Sutner. Classifying circular cellular automata. *Physica D*, 45:386–395, 1990.

[200] Klaus Sutner. sigma-automata and chebyshev-polynomials. *Theoretical Computer Science*, 230(1-2):39–73, January 2000.

[201] Walter Alden Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs.* Dept of Elect. Eng. Syst.: University of Southern California, 1994. PhD Thesis.

[202] Timothy Taylor. *Nidus Design Document.* University of Edinburgh, 1998. Departmental Working Paper No.269. Department of Artificial Intelligence.

[203] Timothy Taylor. *From Artificial Evolution to Artificial Life.* University of Edinburgh, 1999. PhD Thesis.

[204] Gianluca Tempesti. A new self-reproducing cellular automaton capable of construction and computation. In F. Morán, A. Moreno, J. J. Merelo, and P. Chacon, editors, *Advances in Artificial Life: Proc. 3rd Eur. Conf. on Artificial Life (ECAL95)*, volume 929 of *LNAI*, pages 555–563. Springer-Verlag, 1995.

[205] Andrea Tettamanzi and Marco Tomassini. Evolutionary algorithms and their applications. In D. Mange and M. Tomassini, editors, *Bio-Inspired Computing Machines: Toward Novel Computational Architectures.* Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998. (to appear).

[206] Andrea Tettamanzi and Marco Tomassini. *Soft Computing : Integrating Evolutionary, Neural, and Fuzzy System.* Springer-Verlag, 2001.

[207] Tommaso Toffoli. Integration of the phase difference relation in asynchronous sequential networks. In Giorgio Aussiello and Corrado Böhm, editors, *Automata, Languages and Programming, fifth(international) colloquium; Udine, July 17-21, 1978*, pages 457–473, Berlin, 1978.

[208] Tommaso toffoli. Cellular automata as an alternative to (rather than an approximation of) differential equations in modeling physics. *Physica D*, 10:117–127, 1984.

[209] Tommaso Toffoli and Norman H. Margolus. *Cellular Automata Machines, A new Environment for modeling.* MIT Press, Cambridge, MA, 1987.

[210] Tommaso Toffoli and Norman H. Margolus. Invertible cellular automata: A review. *Physica D*, 45:229–253, 1990.

[211] Marco Tomassini. The parallel genetic cellular automata: Application to global function optimization. In R. F. Albrecht, C. R. Reeves, and N. C. Steele, editors, *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 385–391. Springer-Verlag, 1993.

[212] Marco Tomassini, Moshe Sipper, and Mathieu Perrenoud. On the generation of high-quality random numbers by two-dimensional cellular automata. *IEEE Transactions on Computers*, 49(10):1146–1151, 2000.

[213] Alan Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society (B)*, 237, 1952.

[214] Stanislav Ulam. Random processes and transformations. In *Proceedings of the International congress of Mathematics, 1950*, volume 2, 1952.

[215] Tatsuo Unemi. A simple evolvable development system in euclidean space. *Lectures in Mathematics in the Life Sciences*, 26:103–110, 1999.

[216] R. R. Varshamov. Estimate of the number of signals in error correcting codes. *Dokl. Akad. Nauk. SSSR*, 117:739–741, 1957.

[217] Gérard Y. Vichniac. Simulating physics with cellular automata. *Physica D*, 10:96–116, 1984.

[218] Gérard Y Vichniac. Boolean derivatives on cellular automata. *Physica D*, 45:63–74, 1990.

[219] Darell Whitley. The genitor algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 239–255. Morgan Kaufmann, 1989.

[220] Darrel Whitley. Cellular genetic algorithms. In Stephanie Forrest, editor, *Proceedings of the Fith International Conference on Artificial Neural Networks and Genetic Algorithms*, page 658, San Mateo, CA, 1993. Morgan Kaufmann.

[221] Oscar Wilde. *The picture of Dorian Gray and other writings.* A Bantam Classic. Bantam Books, New-York, 1982(1891).

[222] V. william Porto. Evolutionary programming. In Thomas Baeck, Fogel David B, and Zbigniew Michalewicz, editors, *Evolutionary Computation 1*, pages 89–102, Bristol, UK and Philadelphia, USA, 2000. Institute of Physics Publishing.

[223] Stephen Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55(3):601–644, july 1983.

[224] Stephen Wolfram. Computation theory of cellular automata. *Communications in mathematical physics*, 96:15–57, 1984.

[225] Stephen Wolfram. Universality and complexity in cellular automata. *Physica D*, 10:1–35, 1984.

[226] Stephen Wolfram. Approaches to complexity engineering. *Physica D*, 22:385–399, 1986.

[227] Stephen Wolfram. *Cellular Automata and Complexity.* Addison-Wesley, Reading, MA, 1994.

[228] Stephen Wolfram. *A new kind of science.* Wolfram media, inc., 2001?

[229] David H. Wolpert and William G. Macready. No free lunch theorems for search. *Santa Fe Institute Tech. Rep.*, SFI-TR-95-02-010, 1995.

[230] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

[231] Andrew Wuensche. Classifying cellular automata automatically: Finding gliders, filtering, and relating space-time patterns, attractor basins, and the z paramater. *Complexity*, 4(3):47–66, 1999.

[232] Tomoyuki Yamamoto and Kunihiko Kaneko. Tile automaton: A model for an architecture of a living system. *Artificial Life*, 5:37–76, 1999.

[233] Tomohiro Yasuda, Hideo Bannai, Shuichi Onami, Satoru Miyano, and Hiroaki Kitano. Towards automatic construction of cell-lineage of *c.elegans* from nomarski dic microscope images. In K. Asai, S. Miyano, and T. Takagi, editors, *Genome Informatics 1999*, Tokyo, 1999. Universal Academy Press.

[234] Wieslaw Zielonka. Notes on finite asynchronous automata. *Informatique théorique et Applications/Theoretical Infomatics and Applications*, 21(2):99–135, 1987.

[235] Wieslaw Zielonka. Safe executions of recognizable trace languages by asynchronous automata. In Albert R. Meyer and Michael A. Taitslin, editors, *Logic at Botik'89*, Lecture Notes in Computer Science, LNCS 363, pages 278–289, Berlin, 1989. Springer-Verlag.

[236] Konrad Zuse. *Rechnender Raum.* Vieweg, Braunschweig, 1969. Translated as Calculating Space, Tech. Trans., AZT-70-164-DEMIT, MIT Project MAC (1970).

Appendix A

Some Cell Programs Obtained Using Phuon

This annex presents, for information, the two cell programs studied in chapter 3. The food foraging program was found by an earlier version of the cellular language where the `Split` command took only one argument, the direction in which the new cell is created, and the `Nop` command had a constant argument `1`. The instructions are numbered in the form `X.xx`, where `X` is either `S, B,` or `A`, indicating the type, Statement, Boolean and Arithmetic, and `xx` is simply the total number of instructions in its type.

A.1 Food Foraging

```
S.1: Nop(1);
S.2: IfB.1:((A.1: readEnv) > (A.2: readN)){
_S.3: Split (A.3: readEnv);
_S.4: Nop(1);
_S.5: Write_Own(A.4: readEnv);
_S.6: IfB.2:((A.5: readEnv) > (A.6: readN)){
__S.7: Split (A.7: readEnv);
__S.8: Split (A.8: readEnv);
__S.9: Nop(1);
__S.10: Write_Own(A.9: readEnv);
__S.11: IfB.3:((A.10: readEnv) > (A.11: readN)){
___S.12: Split (A.12: readEnv);
___S.13: Split (A.13: readEnv);
___S.14: Nop(1);
___S.15: Write_Own(A.14: readEnv);
___S.16: IfB.4:((A.15: readEnv) > (A.16: readN)){
____S.17: Split (A.17: readEnv);
____S.18: Nop(1);
___}else {
____S.19: Nop(1);
____S.20: Nop(1);
____S.21: DIE!;
___}
___S.22: Nop(1);
___S.23: DIE!;
__}else {
___S.24: Nop(1);
___S.25: Nop(1);
___S.26: Nop(1);
__}
__S.27: Nop(1);
__S.28: DIE!;
_}else {
__S.29: Nop(1);
__S.30: Nop(1);
__S.31: Nop(1);
_}
_S.32: Nop(1);
_S.33: DIE!;
}else {
_S.34: IfB.5:((A.18: readEnv) > (A.19: readN)){
__S.35: Split (A.20: readS);
__S.36: Split (A.21: read0);
__S.37: Nop(1);
_}else {
```

```
__S.38: Nop(1);
__S.39: DIE!;
_}
_S.40: Nop(1);
_S.41: Nop(1);
}
S.42: Nop(1);
S.43: Nop(1);
```

A.2 Controlled Growth

```
S.1: Repeat (A.1: read0) {
__S.2: Write_Env (A.2: readE) ;
__S.3: Write_West (A.3: readS) ;
__S.4: Repeat A.4: (_0){
____S.5: Repeat (A.5: readEnv) {
______S.6: DIE!;
____}
____S.7: IfB.1: ( (A.6: readS)  <  (A.7: read0) ) {
______S.8: Split( Arg1: A.8: (_7), Arg2: A.9: (#2));
______S.9: Write_Own (A.10: readN) ;
______S.10: DIE!;
____}else {
______S.11: Write_Own (A.11: readN) ;
______S.12: Split( Arg1: A.12: ( (A.13: readW)  +  A.14: (#6)) , Arg2: A.15: (#8));
______S.13: DIE!;
____}
____S.14: Write_Own (A.16: readN) ;
____S.15: Repeat (A.17: readEnv) {
______S.16: DIE!;
____}
____S.17: IfB.2: ( (A.18: readS)  <  (A.19: readW) ) {
______S.18: DIE!;
____}else {
______S.19: Write_North (A.20: readE) ;
______S.20: Write_Own (A.21: read0) ;
______S.21: DIE!;
____}
____S.22: Write_Own (A.22: readN) ;
____S.23: Repeat (A.23: read0) {
______S.24: Write_Env (A.24: readE) ;
______S.25: Write_West (A.25: readS) ;
______S.26: Repeat A.26: (#0){
________S.27: Repeat (A.27: readEnv) {
__________S.28: DIE!;
________}
________S.29: IfB.3: ( (A.28: readN)  >  (A.29: read0) ) {
__________S.30: Split( Arg1: A.30: (#7), Arg2: A.31: (#2));
__________S.31: Split( Arg1: A.32: (#0), Arg2: (A.33: readB) );
__________S.32: DIE!;
________}else {
__________S.33: Split( Arg1: A.34: (#0), Arg2: (A.35: readN) );
__________S.34: DIE!;
________}
________S.35: Write_Own (A.36: readN) ;
________S.36: Repeat (A.37: readEnv) {
__________S.37: DIE!;
________}
________S.38: IfB.4: ( (A.38: readN)  >  (A.39: read0) ) {
__________S.39: Write_Own (A.40: readN) ;
__________S.40: Split( Arg1: A.41: ( (A.42: read0)  +  A.43: (#6)) , Arg2: A.44: (#8));
__________S.41: DIE!;
________}else {
__________S.42: Write_North (A.45: readE) ;
__________S.43: Write_Env A.46: (#1);
__________S.44: Write_Own (A.47: readEnv) ;
__________S.45: Split( Arg1: A.48: (#7), Arg2: A.49: (#8));
__________S.46: Write_Own (A.50: readN) ;
__________S.47: Split( Arg1: A.51: ( (A.52: readW)  +  (A.53: readN) ) , Arg2: A.54: (#8));
__________S.48: DIE!;
________}
________S.49: Write_Own (A.55: readN) ;
________S.50: DIE!;
______}
______S.51: IfB.5: ( (A.56: readB)  >  (A.57: read0) ) {
________S.52: Write_West (A.58: readW) ;
________S.53: Repeat A.59: (#0){
__________S.54: Write_North (A.60: readN) ;
__________S.55: Write_South (A.61: read0) ;
__________S.56: Write_North (A.62: read0) ;
__________S.57: Write_Env A.63: (#7);
__________S.58: Write_Own (A.64: readEnv) ;
__________S.59: Split( Arg1: A.65: (#7), Arg2: A.66: (#8));
__________S.60: Write_Own (A.67: readN) ;
__________S.61: Split( Arg1: A.68: ( (A.69: readW)  +  A.70: (#6)) , Arg2: A.71: (#8));
__________S.62: DIE!;
________}
________S.63: IfB.6: ( (A.72: readS)  <  (A.73: readW) ) {
__________S.64: Split( Arg1: A.74: (#0), Arg2: (A.75: readB) );
```

```
__________S.65: DIE!;
________}
________S.66: NOP( A.76: (#6));
________S.67: DIE!;
______}
______S.68: DIE!;
____}
____S.69: Write_North (A.77: readN) ;
____S.70: Write_South (A.78: read0) ;
____S.71: Write_North (A.79: read0) ;
____S.72: Write_Env A.80: (#1);
____S.73: Write_Own (A.81: readEnv) ;
____S.74: Split( Arg1: A.82: (#7), Arg2: A.83: (#8));
____S.75: IfB.7: ( (A.84: readB)  <  (A.85: read0) ) {
______S.76: Split( Arg1: A.86: (#7), Arg2: A.87: (#2));
______S.77: Write_Own (A.88: readN) ;
______S.78: DIE!;
____}else {
______S.79: Split( Arg1: A.89: (#0), Arg2: (A.90: readB) );
______S.80: DIE!;
____}
____S.81: Write_Own (A.91: readN) ;
____S.82: Split( Arg1: A.92: ( (A.93: readN)  +  A.94: (#6)) , Arg2: A.95: (#8));
____S.83: IfB.8: ( (A.96: readS)  <  (A.97: read0) ) {
______S.84: Write_Own A.98: (#0);
______S.85: DIE!;
____}else {
______S.86: Write_Env A.99: (#7);
______S.87: Write_Own (A.100: readEnv) ;
______S.88: Split( Arg1: A.101: (#7), Arg2: A.102: (#8));
______S.89: Write_Own (A.103: readN) ;
______S.90: Split( Arg1: A.104: ( (A.105: readW)  +  A.106: (#6)) , Arg2: A.107: (#8));
______S.91: DIE!;
____}
____S.92: Write_Own (A.108: read0) ;
____S.93: Write_West (A.109: readW) ;
____S.94: Repeat (A.110: readW) {
______S.95: DIE!;
____}
____S.96: IfB.9: ( (A.111: readB)  >  (A.112: read0) ) {
______S.97: Split( Arg1: A.113: (#0), Arg2: (A.114: readB) );
______S.98: DIE!;
____}
____S.99: NOP(A.115: (#6));
____S.100: DIE!;
__}
__S.101: IfB.10: ( (A.116: readS)  <  (A.117: read0) ) {
____S.102: Write_West (A.118: read0) ;
____S.103: Repeat A.119: (#0){
______S.104: DIE!;
____}
____S.105: IfB.11: ( (A.120: readS)  <  (A.121: readW) ) {
______S.106: Split( Arg1: A.122: (#0), Arg2: (A.123: readN) );
______S.107: Repeat (A.124: readEnv) {
________S.108: Repeat (A.125: read0) {
__________S.109: Write_Env (A.126: readE) ;
__________S.110: Write_West (A.127: readS) ;
__________S.111: Repeat A.128: (#0){
____________S.112: Write_North (A.129: readN) ;
____________S.113: Write_South (A.130: read0) ;
____________S.114: Write_North (A.131: read0) ;
____________S.115: Write_Env A.132: (#1);
____________S.116: Write_Own (A.133: readEnv) ;
____________S.117: Split( Arg1: A.134: (#7), Arg2: A.135: (#8));
____________S.118: IfB.12: ( (A.136: readB)  <  (A.137: read0) ) {
______________S.119: Split( Arg1: A.138: (#7), Arg2: A.139: (#2));
______________S.120: Write_Own (A.140: readN) ;
______________S.121: DIE!;
____________}else {
______________S.122: Split( Arg1: A.141: (#0), Arg2: (A.142: readB) );
______________S.123: DIE!;
____________}
____________S.124: Write_Own (A.143: readN) ;
____________S.125: Repeat (A.144: readEnv) {
______________S.126: DIE!;
____________}
____________S.127: IfB.13: ( (A.145: readS)  <  (A.146: read0) ) {
______________S.128: Write_Own A.147: (#0);
______________S.129: DIE!;
____________}else {
______________S.130: Write_North (A.148: readE) ;
______________S.131: Write_Own (A.149: read0) ;
______________S.132: DIE!;
____________}
____________S.133: Write_Own (A.150: read0) ;
____________S.134: Write_West (A.151: readW) ;
____________S.135: Repeat (A.152: readW) {
______________S.136: DIE!;
____________}
____________S.137: IfB.14: ( (A.153: readS)  <  (A.154: readW) ) {
______________S.138: Split( Arg1: A.155: (#0), Arg2: (A.156: readB) );
______________S.139: DIE!;
```

```
__________}
__________S.140: Write_North (A.157: readN) ;
__________S.141: Write_South (A.158: read0) ;
__________S.142: Write_North (A.159: read0) ;
__________S.143: Write_Env A.160: (#1);
__________S.144: Write_Own (A.161: readEnv) ;
__________S.145: Split( Arg1: A.162: (#7), Arg2: A.163: (#8));
__________S.146: IfB.15: ( (A.164: readB)  <  (A.165: read0) ) {
______________S.147: Split( Arg1: A.166: (#7), Arg2: A.167: (#2));
______________S.148: Write_Own (A.168: readN) ;
______________S.149: DIE!;
__________}else {
______________S.150: Split( Arg1: A.169: (#0), Arg2: (A.170: readB) );
______________S.151: DIE!;
__________}
__________S.152: Write_Own (A.171: readN) ;
__________S.153: Repeat (A.172: readEnv) {
______________S.154: DIE!;
__________}
__________S.155: IfB.16: ( (A.173: readS)  <  (A.174: read0) ) {
______________S.156: Write_Own A.175: (#0);
______________S.157: DIE!;
__________}else {
______________S.158: Write_North (A.176: readE) ;
______________S.159: Write_Own (A.177: read0) ;
______________S.160: DIE!;
__________}
__________S.161: Write_Own (A.178: read0) ;
__________S.162: Write_West (A.179: readW) ;
__________S.163: Repeat (A.180: readW) {
______________S.164: DIE!;
__________}
__________S.165: IfB.17: ( (A.181: readB)  >  (A.182: read0) ) {
______________S.166: Split( Arg1: A.183: (#0), Arg2: (A.184: readB) );
______________S.167: DIE!;
__________}
__________S.168: NOP( A.185: (#6));
__________S.169: DIE!;
__________}
__________S.170: IfB.18: ( (A.186: readS)  <  (A.187: read0) ) {
__________S.171: Write_West (A.188: read0) ;
__________S.172: Repeat A.189: (#0){
______________S.173: DIE!;
__________}
__________S.174: IfB.19: ( (A.190: readS)  <  (A.191: readW) ) {
______________S.175: Split( Arg1: A.192: (#0), Arg2: (A.193: readN) );
______________S.176: Repeat (A.194: readEnv) {
________________S.177: Repeat (A.195: read0) {
__________________S.178: Write_Env (A.196: readE) ;
__________________S.179: Write_West (A.197: readS) ;
__________________S.180: Repeat A.198: (#0){
____________________S.181: Write_North (A.199: readN) ;
____________________S.182: Write_South (A.200: read0) ;
____________________S.183: Write_North (A.201: read0) ;
____________________S.184: Write_Env A.202: (#1);
____________________S.185: Write_Own (A.203: readEnv) ;
____________________S.186: Split( Arg1: A.204: (#7), Arg2: A.205: (#8));
____________________S.187: IfB.20: ( (A.206: readB)  <  (A.207: read0) ) {
______________________S.188: Split( Arg1: A.208: (#7), Arg2: A.209: (#2));
______________________S.189: Write_Own (A.210: readN) ;
______________________S.190: DIE!;
____________________}else {
______________________S.191: Split( Arg1: A.211: (#0), Arg2: (A.212: readB) );
______________________S.192: DIE!;
____________________}
____________________S.193: Write_Own (A.213: readN) ;
____________________S.194: Repeat (A.214: readEnv) {
______________________S.195: DIE!;
____________________}
____________________S.196: IfB.21: ( (A.215: readS)  <  (A.216: read0) ) {
______________________S.197: Write_Own A.217: (#0);
______________________S.198: DIE!;
____________________}else {
______________________S.199: Write_North (A.218: readE) ;
______________________S.200: Write_Own (A.219: read0) ;
______________________S.201: DIE!;
____________________}
____________________S.202: Write_Own (A.220: read0) ;
____________________S.203: Write_West (A.221: readW) ;
____________________S.204: Repeat (A.222: readW) {
______________________S.205: DIE!;
____________________}
____________________S.206: IfB.22: ( (A.223: readB)  >  (A.224: read0) ) {
______________________S.207: Split( Arg1: A.225: (#0), Arg2: (A.226: readB) );
______________________S.208: DIE!;
____________________}
____________________S.209: Write_North (A.227: readN) ;
____________________S.210: Write_South (A.228: read0) ;
____________________S.211: Write_North (A.229: read0) ;
____________________S.212: Write_Env A.230: (#1);
____________________S.213: Write_Own (A.231: readEnv) ;
____________________S.214: Split( Arg1: A.232: (#7), Arg2: A.233: (#8));
```

```
__________S.215: IfB.23: ( (A.234: readB)  >  (A.235: read0) ) {
__________S.216: Split( Arg1: A.236: (#7), Arg2: A.237: (#2));
__________S.217: Write_Own (A.238: readN) ;
__________S.218: DIE!;
__________}else {
__________S.219: Split( Arg1: A.239: (#0), Arg2: (A.240: readB) );
__________S.220: DIE!;
__________}
__________S.221: Write_Own (A.241: readN) ;
__________S.222: Repeat A.242: (#7){
__________S.223: DIE!;
__________}
__________S.224: IfB.24: ( (A.243: readS)  <  (A.244: read0) ) {
__________S.225: Write_Own A.245: (#0);
__________S.226: DIE!;
__________}else {
__________S.227: Write_North (A.246: readE) ;
__________S.228: Write_Own (A.247: read0) ;
__________S.229: DIE!;
__________}
__________S.230: Write_Own (A.248: read0) ;
__________S.231: Write_West (A.249: readW) ;
__________S.232: Repeat (A.250: readW) {
__________S.233: DIE!;
__________}
__________S.234: IfB.25: ( (A.251: readB)  >  (A.252: read0) ) {
__________S.235: Split( Arg1: A.253: (#0), Arg2: (A.254: readB) );
__________S.236: DIE!;
__________}
__________S.237: NOP(A.255: (#6));
__________S.238: DIE!;
________}
________S.239: IfB.26: ( (A.256: readS)  <  (A.257: read0) ) {
__________S.240: Write_West (A.258: read0) ;
__________S.241: Repeat A.259: (#0){
__________S.242: DIE!;
__________}
__________S.243: IfB.27: ( (A.260: readS)  <  (A.261: readW) ) {
__________S.244: Write_West (A.262: readS) ;
__________S.245: Repeat A.263: (#0){
__________S.246: Repeat (A.264: readEnv) {
__________S.247: DIE!;
__________}
__________S.248: IfB.28: ( (A.265: readS)  <  (A.266: read0) ) {
__________S.249: Split( Arg1: A.267: (#7), Arg2: A.268: (#2));
__________S.250: Write_Own (A.269: readN) ;
__________S.251: DIE!;
__________}else {
__________S.252: Write_Own (A.270: readN) ;
__________S.253: Split( Arg1: A.271: ( (A.272: readW)  +  A.273: (#6)) , Arg2: A.274: (#8));
__________S.254: DIE!;
__________}
__________S.255: Write_Own (A.275: readN) ;
__________S.256: Repeat (A.276: readEnv) {
__________S.257: DIE!;
__________}
__________S.258: IfB.29: ( (A.277: readS)  <  (A.278: readW) ) {
__________S.259: DIE!;
__________}else {
__________S.260: Write_North (A.279: readE) ;
__________S.261: Write_Own (A.280: read0) ;
__________S.262: DIE!;
__________}
__________S.263: Write_Own (A.281: readN) ;
__________S.264: Repeat (A.282: read0) {
__________S.265: Write_Env (A.283: readE) ;
__________S.266: Write_West (A.284: readS) ;
__________S.267: Repeat A.285: (#0){
__________S.268: Repeat (A.286: readEnv) {
__________S.269: DIE!;
__________}
__________S.270: IfB.30: ( (A.287: readB)  >  (A.288: read0) ) {
__________S.271: Split( Arg1: A.289: (#7), Arg2: A.290: (#2));
__________S.272: Split( Arg1: A.291: (#0), Arg2: (A.292: readB) );
__________S.273: DIE!;
__________}else {
__________S.274: Split( Arg1: A.293: (#0), Arg2: (A.294: readN) );
__________S.275: DIE!;
__________}
__________S.276: Write_Own (A.295: readN) ;
__________S.277: Repeat (A.296: readEnv) {
__________S.278: DIE!;
__________}
__________S.279: IfB.31: ( (A.297: readS)  <  (A.298: readW) ) {
__________S.280: Write_Own (A.299: readN) ;
__________S.281: Split( Arg1: A.300: ( (A.301: read0)  +  A.302: (#6)) , Arg2: A.303: (#8));
__________S.282: DIE!;
__________}else {
__________S.283: Write_North (A.304: readE) ;
__________S.284: Write_Env A.305: (#1);
__________S.285: Write_Own (A.306: readEnv) ;
__________S.286: Split( Arg1: A.307: (#7), Arg2: A.308: (#8));
```

```
_____S.287: Write_Own (A.309: readN) ;
_____S.288: Split( Arg1: A.310: ( (A.311: readW)  +  (A.312: readN) ) , Arg2: A.313: (#8));
_____S.289: DIE!;
____}
____S.290: Write_Own (A.314: readN) ;
____S.291: DIE!;
___}
___S.292: IfB.32: ( (A.315: readB)  <  (A.316: read0) ) {
____S.293: Write_West (A.317: readW) ;
____S.294: Repeat A.318: (#0){
_____S.295: Write_North (A.319: readN) ;
_____S.296: Write_South (A.320: read0) ;
_____S.297: Write_North (A.321: read0) ;
_____S.298: Write_Env A.322: (#7);
_____S.299: Write_Own (A.323: readEnv) ;
_____S.300: Split( Arg1: A.324: (#7), Arg2: A.325: (#8));
_____S.301: Write_Own (A.326: readN) ;
_____S.302: Split( Arg1: A.327: ( (A.328: readW)  +  A.329: (#6)) , Arg2: A.330: (#8));
_____S.303: DIE!;
____}
____S.304: IfB.33: ( (A.331: readB)  >  (A.332: read0) ) {
_____S.305: Split( Arg1: A.333: (#0), Arg2: (A.334: readB) );
_____S.306: DIE!;
____}
____S.307: NOP(A.335: (#6));
____S.308: DIE!;
___}
___S.309: DIE!;
__}
__S.310: Write_North (A.336: readN) ;
__S.311: Write_South (A.337: read0) ;
__S.312: Write_North (A.338: read0) ;
__S.313: Write_Env A.339: (#1);
__S.314: Write_Own (A.340: readEnv) ;
__S.315: Split( Arg1: A.341: (#7), Arg2: A.342: (#8));
__S.316: IfB.34: ( (A.343: readB)  <  (A.344: read0) ) {
___S.317: Split( Arg1: A.345: (#7), Arg2: A.346: (#2));
___S.318: Write_Own (A.347: readN) ;
___S.319: DIE!;
__}else {
___S.320: IfB.35: ( (A.348: readS)  <  (A.349: read0) ) {
____S.321: Split( Arg1: A.350: (#0), Arg2: (A.351: readB) );
____S.322: DIE!;
___}
___S.323: NOP(A.352: (#6));
___S.324: DIE!;
__}
__S.325: Write_Own (A.353: readN) ;
__S.326: Repeat A.354: (#2){
___S.327: DIE!;
__}
__S.328: Split( Arg1: A.355: (#0), Arg2: (A.356: readB) );
__S.329: DIE!;
_}
_S.330: IfB.36: ( (A.357: readS)  <  (A.358: read0) ) {
__S.331: Write_West (A.359: read0) ;
__S.332: Repeat A.360: (#0){
___S.333: DIE!;
__}
__S.334: IfB.37: ( (A.361: readS)  <  (A.362: readW) ) {
___S.335: Split( Arg1: A.363: (#0), Arg2: (A.364: readN) );
___S.336: Repeat (A.365: readEnv) {
___S.337: Repeat (A.366: read0) {
____S.338: Write_Env (A.367: readE) ;
____S.339: IfB.38: ( (A.368: readS)  <  (A.369: read0) ) {
_____S.340: Write_West (A.370: read0) ;
_____S.341: Repeat A.371: (#0){
______S.342: DIE!;
_____}
_____S.343: IfB.39: ( (A.372: readS)  <  (A.373: read0) ) {
______S.344: Split( Arg1: A.374: (#0), Arg2: (A.375: readN) );
______S.345: Write_West (A.376: read0) ;
______S.346: Repeat A.377: (#0){
_______S.347: DIE!;
______}
______S.348: IfB.40: ( (A.378: readB)  <  (A.379: read0) ) {
_______S.349: Split( Arg1: A.380: (#0), Arg2: (A.381: readN) );
_______S.350: Write_South (A.382: readW) ;
_______S.351: Write_North (A.383: read0) ;
_______S.352: Write_Env A.384: (#1);
_______S.353: Write_Own (A.385: readEnv) ;
_______S.354: Split( Arg1: A.386: (#7), Arg2: A.387: (#8));
_______S.355: Write_Own (A.388: readN) ;
_______S.356: Split( Arg1: A.389: ( (A.390: readW)  +  A.391: (#6)) , Arg2: A.392: (#8));
_______S.357: DIE!;
______}
______S.358: NOP(A.393: (#6));
______S.359: DIE!;
_____}
_____S.360: NOP(A.394: (#6));
_____S.361: DIE!;
____}
```

```
_____________________________S.362: DIE!;
_____________________________}
____________________________S.363: Write_North (A.395: readN) ;
____________________________S.364: Write_South (A.396: read0) ;
____________________________S.365: Write_North (A.397: read0) ;
____________________________S.366: Write_Env A.398: (#7);
____________________________S.367: Write_Own (A.399: readEnv) ;
____________________________S.368: Split( Arg1: A.400: (#7), Arg2: A.401: (#8));
____________________________S.369: Write_Own (A.402: readN) ;
____________________________S.370: Split( Arg1: A.403: ( (A.404: readW)  +  A.405: (#6)) , Arg2: A.406: (#8));
____________________________S.371: DIE!;
___________________________}
___________________________S.372: Write_Own (A.407: readN) ;
___________________________S.373: Split( Arg1: A.408: ( A.409: (#7) +  A.410: (#6)) , Arg2: A.411: (#8));
___________________________S.374: Write_Own (A.412: readN) ;
___________________________S.375: Split( Arg1: A.413: ( (A.414: readW)  +  A.415: (#6)) , Arg2: (A.416: readB) );
___________________________S.376: DIE!;
__________________________}
__________________________S.377: NOP(A.417: (#6));
__________________________S.378: DIE!;
_________________________}
_____________________S.379: DIE!;
____________________}
____________________S.380: DIE!;
___________________}
___________________S.381: DIE!;
_____________________}
_____________S.382: Write_North (A.418: readN) ;
_____________S.383: Repeat A.419: (#0){
_____________S.384: Repeat (A.420: readEnv) {
____________________S.385: DIE!;
___________________}
___________________S.386: IfB.41: ( (A.421: readN)  >  (A.422: read0) ) {
____________________S.387: Split( Arg1: A.423: (#7), Arg2: A.424: (#2));
____________________S.388: Split( Arg1: A.425: (#0), Arg2: (A.426: readB) );
____________________S.389: DIE!;
___________________}else {
____________________S.390: Split( Arg1: A.427: (#0), Arg2: (A.428: readN) );
____________________S.391: DIE!;
___________________}
___________________S.392: Write_Own (A.429: readN) ;
___________________S.393: Repeat (A.430: readEnv) {
____________________S.394: DIE!;
___________________}
___________________S.395: IfB.42: ( (A.431: readN)  >  (A.432: read0) ) {
____________________S.396: Write_Own (A.433: readN) ;
____________________S.397: Split( Arg1: A.434: ( (A.435: read0)  +  A.436: (#6)) , Arg2: A.437: (#8));
____________________S.398: DIE!;
___________________}else {
____________________S.399: Write_North (A.438: readE) ;
____________________S.400: Write_Env A.439: (#1);
____________________S.401: Write_Own (A.440: readEnv) ;
____________________S.402: Split( Arg1: A.441: (#7), Arg2: A.442: (#8));
____________________S.403: Write_Own (A.443: readN) ;
____________________S.404: Split( Arg1: A.444: ( (A.445: readW)  +  (A.446: readN) ) , Arg2: A.447: (#8));
____________________S.405: DIE!;
___________________}
___________________S.406: Write_Own (A.448: readN) ;
___________________S.407: DIE!;
_____________________}
_____________S.408: IfB.43: ( (A.449: readB)  >  (A.450: read0) ) {
___________________S.409: Write_West (A.451: readW) ;
___________________S.410: Repeat A.452: (#0){
____________________S.411: Write_North A.453: ( (A.454: readW)  +  A.455: (#6)) ;
____________________S.412: Write_South (A.456: read0) ;
____________________S.413: Write_North (A.457: read0) ;
____________________S.414: Write_Env A.458: (#7);
____________________S.415: Write_Own (A.459: readEnv) ;
____________________S.416: Split( Arg1: A.460: (#7), Arg2: A.461: (#8));
____________________S.417: Write_Own (A.462: readN) ;
____________________S.418: Split( Arg1: A.463: ( (A.464: readW)  +  A.465: (#6)) , Arg2: A.466: (#8));
____________________S.419: DIE!;
___________________}
___________________S.420: IfB.44: ( (A.467: readS)  <  (A.468: readW) ) {
____________________S.421: Split( Arg1: A.469: (#0), Arg2: (A.470: readB) );
____________________S.422: DIE!;
___________________}
___________________S.423: NOP(A.471: (#6));
___________________S.424: DIE!;
_____________________}
_____________S.425: DIE!;
_____________}
_____________S.426: Write_Own (A.472: readN) ;
_____________S.427: Split( Arg1: A.473: ( A.474: (#7) +  A.475: (#6)) , Arg2: A.476: (#8));
_____________S.428: Write_Own (A.477: readN) ;
_____________S.429: Split( Arg1: A.478: ( (A.479: readW)  +  A.480: (#6)) , Arg2: (A.481: readB) );
_____________S.430: DIE!;
____________}
____________S.431: NOP(A.482: (#6));
____________S.432: DIE!;
___________}
___________S.433: DIE!;
```

```
________}
________S.434: Write_North (A.483: readN) ;
________S.435: Write_South (A.484: read0) ;
________S.436: Write_North (A.485: read0) ;
________S.437: Write_Env A.486: (#7);
________S.438: Write_Own (A.487: readEnv) ;
________S.439: Split( Arg1: A.488: (#7), Arg2: A.489: (#8));
________S.440: Write_Own (A.490: readN) ;
________S.441: Split( Arg1: A.491: ( (A.492: readW)  +  A.493: (#6)) , Arg2: A.494: (#8));
________S.442: DIE!;
______}
______S.443: Write_Own (A.495: readN) ;
______S.444: Split( Arg1: A.496: ( A.497: (#7) +  A.498: (#6)) , Arg2: A.499: (#8));
______S.445: Write_Own (A.500: readN) ;
______S.446: Split( Arg1: A.501: ( (A.502: readW)  +  A.503: (#6)) , Arg2: (A.504: readB) );
______S.447: DIE!;
____}
____S.448: NOP(A.505: (#6));
____S.449: DIE!;
__}
__S.450: DIE!;
}
S.451: Write_North (A.506: readN) ;
S.452: Write_South (A.507: readW) ;
S.453: Write_North (A.508: read0) ;
S.454: Write_Env A.509: (#1);
S.455: Write_Own (A.510: readEnv) ;
S.456: Split( Arg1: A.511: (#7), Arg2: A.512: (#8));
S.457: Write_Own (A.513: readN) ;
S.458: Split( Arg1: A.514: ( (A.515: readW)  +  A.516: (#6)) , Arg2: A.517: (#8));
S.459: DIE!;
```

Appendix B

Curriculum Vitae

Mathieu Capcarrère was born on the 14^{th} of March 1974 in Montpellier. He gained in 1992 his French baccalaureate in mathematics and physics with honors and moved to the *Mathématiques Supèrieures* class. In 1993, he joined the Imperial College of Science Technology (that was to become soon afterward the Imperial College of Science, Technology and Medicine), and gained in 1997 a first-honours Master of Engineering degree in computing science, specializing in Artificial Intelligence. Following, unknowingly, the same path many followed before, he found in Artificial Life an answer to the original queries of Artificial Intelligence, dropping the "logic" straitjacket. During this London stay, he spent six months in 1996 at the Logic Systems Laboratory in Lausanne, working with Drs Tomassini and Sipper. In 1997, he rejoined them, and started a PhD, in july 1998, whose results is the book you hold. Mathieu Capcarrère is author or co-author of a dozen papers listed below:

Journal Papers:

1. M. Capcarrere, M. Sipper, *Necessary Conditions for Density Classification by Cellular Automata*, In Physical Review E, 6403 (3): 6113-7, Part 2, Sep 2001.

2. M. Capcarrere, A. Tettamanzi, M. Tomassini, M. Sipper, *A Statistical Study of a Class of Cellular Evolutionary Algorithms*, Evolutionary Computation, 7(3):255-274, 1999.

3. E. Ronald, M. Sipper, M. S. Capcarrère, *Design, Observation, Surprise! A Test of Emergence*, Artificial Life, 5(3): 225-239, 1999.

4. M. Sipper, M. S. Capcarrère, E. Ronald, *A simple Cellular automaton that solves the density and ordering problems.* International Journal of Modern Physics C, 9(7):899-902, October 1998.

5. M. Sipper, M. Tomassini, M. Capcarrere, *Designing Cellular Automata Using a Parallel Evolutionnary Algorithm*, In AIHENP'96 proceedings, World Scientific Publishing Company. This paper was reprinted in "Nuclear instruments & Methods In Physics research A", 389:278-283, 1997, Elsevier Publisher.

6. M. Capcarrere, M. Tomassini, M. Sipper, *A r=1, two-state Cellular Automata that Classifies density*, In Physical Review Letters, 77(24):4969-4971, 1996.

Conference Papers

1. M.S.Capcarrere, *Evolution of Asynchronous Cellular Automata* The seventh international conference on Parallel Problem Solving From Nature, PPSN'02 Proceedings. (Full paper to Appear)

2. M.S.Capcarrere, *Emergent computation in CA: A matter of visual efficiency.* Fifth International Conference on Complex Systems: ICCS 2002 Proceedings. (Full paper to Appear)

3. M.S.Capcarrere, *Evolution of Asynchronous Cellular Automata: Finding the Good Compromise.* Genetic and Evolutionary Computing COnference 2002: GECCO'02, New-York, NY, USA. Proceedings published by Morgan Kaufmann to appear. (poster: 1 page summary).

4. D. Bünzli and M.S.Capcarrere, *Fault-tolerant structures: Towards robust self-replication in a probabilistic environment*, Advances in Artificial Life, proceedings of ECAL'01, Josef Kelemen and Petr Sosík Eds, Lecture Notes in Artificial Intelligence (LNAI 2159), Springer-Verlag, 2001.

5. J. Conus, D.Bongard and M.S.Capcarrere, *The evolution of computer viruses: a natural A-Life task*, International Confernce in Artificial Life (A-Life VII), 2000, (Poster).

6. E. Ronald, M. Sipper, M. S. Capcarrère, *Testing for Emergence in Artificial Life*, in Advances in Artificial Life, 5th European Conference, ECAL'99 proceedings, Floreano, D. and Nicoud, J.-D. and Mondada, F., (Eds.), Lecture Notes in Artificial Intelligence (LNAI 1674), Springer-Verlag, 1999.

7. M. Capcarrere, A. Tettamanzi, M. Tomassini, M. Sipper, *Studying Parallel Evolutionary Algorithms: The Cellular Programming Case*, A. E. Eiben, T. Bäck, M. Schoenauer, H.-P. Schwefel (Eds), Parallel Problem Solving from Nature - PPSN V, proceedings of. Lecture Notes in Computer Science series (LNCS), no 1498, Springer-Verlag, 1998.

8. M. Sipper, M. Tomassini, M. Capcarrere, *Evolving Asynchronous and scaleable Cellular Automata*, International Conf. on Artificial neural Network and Genetic Algorithms, ICANNGA'97 proceedings, Smith et al Eds, Springer-Verlag, 1998.

Special

1. M.Capcarrere, J. Cunningham, L. Kamara, J. Pitt and M. Rigg, *A Testbed for Animating Multi-Agent System*, ESPRIT Basic Research Project 6471, Deliverable DV.2-2P. 1995.

Member of the Program Committee

- ACRI 02: Fifth International Conference on Cellular Automata for Research and Industry. Program Chair: Bastien Chopard.
- ECAL 99: European Conference on Artificial Life, Program Chair: Dario Floreano, Francesco Mondada and Jean-Daniel Nicoud, Conference Secretary: Joseba Urzelai.
- ICES 98: International Conference on Evolvable Systems, Program Chair: Daniel Mange and Moshe Sipper, Conference Secretary: Andres Perez-Uribe.

Occasional reviewer for the following journals

- Physica D, Elsevier Press.
- IEEE Transactions on Evolutionary Computation.
- Evolutionary Computation Journal, Elsevier Press.