Università degli Studi di Milano-Bicocca
Facoltà di Scienze Matematiche, Fisiche e Naturali
Dottorato di Ricerca in Informatica – Ciclo XXIV

The Time-Space Trade-Off in Membrane Computing

Tesi di Dottorato di
Antonio Enrico Porreca

Supervisor: Prof. Claudio Zandron
Tutor: Prof.ssa Lucia Pomello
Coordinator: Prof.ssa Stefania Bandini

Acknowledgements

First of all, I would like to thank my supervisor Claudio Zandron, for all the trust he put in me and his support during my research.

Then, the research group I've been working in, the "Milano Team" as they call us, including Alberto Leporati, Giancarlo Mauri, and the aforementioned Claudio Zandron, together with the "other half" of the team: Paolo Cazzaniga, Dario Pescini and Daniela Besozzi, whom I haven't had the pleasure to work with due to different research interests.

I also wish to thank the Sevilla team, where I've spent six months during my Ph.D.; in particular, I'd like to thank Mario J. Pérez-Jiménez, Miguel Ángel Gutiérrez Naranjo, Agustin Riscos-Núñez, Manuel García-Quismondo, Miguel Ángel Martínez-del-Amor, Ignacio Pérez-Hurtado, Luis Valencia Cabrera, Luis Felipe Macías-Ramos, and last but not least Ana Ruiz. You really made me feel at home.

Another fine lad I've had the pleasure to work with while in Sevilla is Niall Murphy, a colleague and a friend. The two months we've shared in the lab have probably been the most productive of my entire life. He and Satoko Yoshimura, whom I also wish to thank, have been a wonderful company.

Of course, I want to thank my labmates in Milano: Stefano Beretta, Mauro Castelli, Luca Manzoni and the new entry Carlo Maj, together with all the other Ph.D. students, and in particular Lorenza Manenti and Luca Panziera.

And finally, as usual, I want to thank Alice, because she's been here all the time.

Contents

1 Introduction **7**
1.1 Motivation 8
1.2 Contributions 9

2 P systems **11**
2.1 Membrane structures 11
2.2 The contents of regions 12
2.3 Computation rules 14
2.3.1 Object evolution rules 14
2.3.2 Communication rules 15
2.3.3 Dissolution rules 16
2.3.4 Elementary division rules 16
2.3.5 Nonelementary division rules 17
2.4 A formal definition 17
2.4.1 Configurations and computations 18
2.5 Recogniser P systems 19
2.5.1 Uniformity conditions 20
2.6 Time complexity of P systems 22

3 Computing with P systems **25**
3.1 P systems are universal 25
3.1.1 Simulating register machines 26
3.2 Solving **NP**-complete problems in polynomial time 28
3.2.1 Encoding of Boolean formulae 29
3.2.2 Solution to 3SAT 30

4 Space complexity of P systems **37**
4.1 Preliminaries 37
4.2 The notion of space complexity 38
4.2.1 Basic results 40
4.3 Solving problems in **PSPACE** 41
4.3.1 Simulating Turing machines 41
4.4 Simulating P systems via Turing machines 44

4.4.1 Simulation algorithm 44
4.4.2 Analysis of the algorithm 46
4.4.3 Complexity-theoretic implications 50

5 Counting by trading space for time 53
5.1 On the nature of membrane division 53
5.2 Counting problems 54
5.3 Solving THRESHOLD-3SAT 56
5.4 Solving the **PP** problems 59

6 P systems as oracles 61
6.1 Oracle Turing machines 61
6.2 Simulating an oracle machine 62
6.2.1 Informal description of the simulation 63
6.2.2 Technical details 65
6.2.3 Main result 67

7 Conclusions 69
7.1 Open problems 70

Chapter 1

Introduction

The theory of computation investigates the nature and properties of algorithmic procedures. This field emerged in the 20s and 30s of the 20th century from the work on the philosophy and the foundations of mathematics. Of great importance and inspiration was David Hilbert's ambitious program to "dispose of the foundational questions in mathematics once and for all" [41], that lead to fundamental results in logic such as Gödel's incompleteness theorems [8], and ultimately to the birth of recursion theory (nowadays mostly referred to as computability theory) and computer science itself.

The formal notion of computability that is almost universally adopted today is due to Alan Turing, who introduced in his ground-breaking paper *On computable numbers, with an application to the Entscheidungsproblem* [38] a simple, elegant and convincing mathematical formalisation of the notion of computation, as it is carried out by a human executor equipped with enough scratch paper. Turing's work showed that, as long as we accept his notion of computation, there exist well-formed mathematical questions whose answer cannot be computed. In particular, one of the main challenges of Hilbert's program, the *Entscheidungsproblem* (finding a decision procedure for the validity of statements in first-order logic) was proved to be unsolvable.

This formalisation, that rapidly became known as *Turing machine*, is still the reference model for computing devices in theoretical computer science, as it also enjoys the property of being a good model of actual electronic computers; this is also due to the fact that it was itself an inspiration for the design of automatic computing machinery [7].

With the development of computers as a technology, being able to solve a particular problem proved not to be satisfying: *fast*, *efficient* solutions are needed. This led to the development of computational complexity theory, pioneered [10] by Hartmanis and Stearns in the paper *On the computational complexity of algorithms* [13], that also gives the name to the field. Identifying the notion of "efficient" with "polynomial-time computable" is due [10] to

Edmonds [9], while the central question of complexity theory, whether $\mathbf{P} = \mathbf{NP}$, arose from the work of Cook [6] and Karp [15]. This question has shaped the whole development of the field, and still remains open today.

However, the theory of computation is not entirely about Turing machines. Several authors sought to draw inspiration from the way nature "computes" in order to define alternative, unconventional computing models, or, from the opposite point of view, to interpret natural phenomena as computation [1]. For instance, artificial neural networks [19] are inspired by the functioning of neurons in the brain, and genetic algorithms seek to solve computationally hard problems by simulating the processes of mutation, mating and natural selection. A clear example of biological inspiration is given by DNA computing, which provided an actual *in vitro* implementation of an algorithm for the Hamilton path problem [2] (the theory was initiated a few years earlier, particularly by Head [14]).

Inspired by the work on DNA computing [33], Păun introduced in 1998 [31] the notion of *membrane systems*, initially called *super-cells*, and nowadays usually *P systems*. Here the computing device is an abstraction of biological cells. Unlike in neural networks, we do not deal with cells as atomic objects: as the name suggests, the focus is on the membranes that define a cell and its internal compartments, which work together by performing different individual functions. The chemical environment of the various compartments are described in terms of *multisets* of symbols (i.e., sets with multiplicity). Another defining feature of P systems (as they were initially defined) is *maximal parallelism*: as many operations as possible are carried out simultaneously, and no part of the systems remains inactive if it can carry out some part of the computation.

Although P systems have also been investigated from the perspectives of bioinformatics and systems biology, where they might be used as models of biological phenomena in computer simulations, most of the research in membrane computing has been carried out from a language-theoretic (including the original paper [31]), computability-theoretic and complexity-theoretic standpoint. This thesis aims to make a contribution in this latter area.

1.1 Motivation

The goal of this work is to pursue further investigation regarding the computational complexity theory of P systems, particularly for the variant called *P systems with active membranes* [32], which has a richer background from this point of view. This kind of device presents an interesting trade-off, whereby we may be able to solve classically hard problems in polynomial time at the expense of space: an exponential number of membranes can be created by *membrane division* in polynomial time, and they can then com-

pute in parallel. This allows us, for instance, to explore the whole candidate solution space of an **NP**-complete problem in polynomial time. While this time-space trade-off has often been exploited, no formal analysis has been performed before.

The motivation for this work, besides the study of membrane systems *per se*, is twofold.

As is common for research in complexity theory, we aim to gain further insight into the nature of computation, with the aim of ultimately attacking the central open problems of the field, by studying alternative models of computation. The hope here is that by comparing the alternative model with the traditional ones, and analysing how the notion of *hardness* translates in the new framework, we may obtain results that can be then taken back into the usual setting, possibly advancing the state of the art of general complexity theory.

Another, less pragmatic reason why we feel that it is worth pursuing research in complexity theory (whether or not involving membrane systems or natural computing devices) is because we think that this discipline has a relevance in philosophical discussion [1], as much as computability theory had in the past. The fact that nature-inspired devices are involved, instead of the original human-based notion of computation, can also hopefully contribute to a better understanding of what "computing" means.

1.2 Contributions

The contributions this thesis makes can be summarised as follows.

First of all, we have introduced a space complexity measure for P systems, in order to be able to *prove* results about the space-time trade-off, which has always been acknowledged but never formally investigated before.

Space complexity classes for P systems have also been defined and analysed. The overall result here is a proof that P systems with active membranes operating in polynomial space solve *exactly* the same problems solved by Turing machines under the same restriction.

Then, P systems working in exponential space (using elementary membrane division) have been considered. The previously known algorithms to solve **NP**-complete problems in polynomial time have been improved, showing that this class of devices can also solve the "counting" problems in the complexity class **PP**.

This result has then been used, together with an efficient simulation of Turing machines (the first proposed one, to the knowledge of the present author), in order to simulate Turing machines with oracles. This proves that elementary membrane division allows us to solve all problems in the complexity class $\mathbf{P}^{\mathbf{PP}}$ in polynomial time.

Chapter 2

P systems

The variant of P systems we discuss in this thesis, called *P systems with active membranes*, was introduced by Păun in 1999 [32]. The very subtitle of that paper, *Attacking NP-complete problems*, shows how the focus on the complexity-theoretic properties of these devices has been important since the beginning. We begin our discussion of this topic by recalling the structural definition of P systems with active membranes, defining how they can be used to solve decision problems, and how we can measure their efficiency in terms of time complexity; this will allow us to define an array of complexity classes in order to classify decision problems.

2.1 Membrane structures

The main feature of all variants of P systems is their *membrane structure*. Several variants have been proposed in the literature [34], but P systems with active membranes retain the original *cell-like* shape [31]: a hierarchy of membranes nested to an arbitrary depth. There is a clear bijective mapping between the set of membranes and the *regions* they define, delimited by the membrane itself from the outside, and any membrane immediately included in it from the inside. The outermost membrane (sometimes called the *skin*) separates the actual P systems from the surrounding *environment*, which is a further implied region also playing a role in the computation. Membranes not containing further membranes inside them are called *elementary*; the others are called *nonelementary*.

The abstract shape of a membrane structure in any given instant is mathematically represented by a *rooted, unordered tree*. Here the membranes (equivalently, the regions they delimit) correspond to vertices, the outermost membrane being associated with the root of the tree, and an edge between two vertices indicates that the corresponding membranes are located one immediately inside the other; the "parent" and "child" roles are implied by the distance from the root. The leaves of the tree correspond to elementary

membranes, while the internal nodes correspond to the nonelementary ones. The tree is unordered (i.e., there is no distinguishable "first" or "leftmost" child) because we do not keep track of any spatial relation between the membranes other than simple containment[1].

Membranes are not only set apart by their position in the membrane structure, but also by the role they supposedly perform. This is represented in P systems by attaching a *label* taken from a finite set Λ to each membrane (formally, to the vertex representing it in the corresponding unordered tree). As we shall see later, different labels correspond to different sets of rules that can be applied to the membranes or their contents. Initially, each membrane must be given a different label (although two distinct labels might be associated to the same set of rules), while the process of membrane division may create multiple membranes sharing the same label.

Finally, the original description of P systems with active membranes introduces a notion of local state of the membrane itself: this is an *electrical charge*, which can be positive, negative or neutral and, unlike its label, may change during the computation. Formally, the charges are also attached to the nodes of the unordered tree.

A membrane structure is traditionally described in a "linear" notation by means of a string of balanced brackets. Given the unordered rooted tree corresponding to the membrane structure, fix an arbitrary ordering of the children of each node. The resulting ordered tree, ignoring the node labels, is then uniquely interpreted a string of balanced brackets belonging to the language defined by the following context-free grammar, beginning with the nonterminal S_1:

$$S_1 \to [S_2] \qquad\qquad S_2 \to [S_2] \mid S_2 S_2 \mid \epsilon.$$

These are all the non-empty strings of balanced brackets having a single outermost pair. The nesting of brackets is induced by the shape of the tree (and obviously corresponds to the containment relation of the original membrane structure). Finally, each bracket is subscripted by the label of the corresponding membrane, and superscripted by its charge. An example of membrane structure, represented in a pictorial form, as a tree, and in bracket notation can be found in Figure 2.1.

2.2 The contents of regions

A single molecule can be represented abstractly as a symbol taken from an alphabet[2] Γ. However, subsets of Γ are not an adequate description of a whole chemical environment, as they do not carry any information related to the

[1] There exist other variants of P systems, such as the *spatial* ones, where the positions of membranes and objects do actually matter [5].

[2] All alphabets in this thesis are assumed to be finite.

$$[[\,]^{+}_{h_1}\ [[\,]^{0}_{h_3}\ [\,]^{-}_{h_4}]^{0}_{h_2}]^{0}_{h_0}$$

Figure 2.1: A membrane structure and two formalisms representing it.

(absolute or relative) quantities of molecules of the same species. Since the concentrations of chemicals in a region do actually matter from a metabolic standpoint, we use *multisets* to describe them.

A multiset w over a set Γ is simply a function $w\colon \Gamma \to \mathbb{N}$, mapping each object to its multiplicity. Generalisations of the usual set-theoretic operations can be defined on the sets of multisets over Γ; for instance, the union of multisets w_1 and w_2 is the multiset where, for all $a \in \Gamma$, the object a has multiplicity $w_1(a) + w_2(a)$. From an algebraic point of view, the set of multisets over Γ, together with the operation of union, is the *free commutative monoid on* Γ. Compare this to the set $\Gamma^\star$ of strings, which is the *free monoid on* Γ: here the operation (i.e., string concatenation) is *not* commutative. When describing the chemical environment of a region, we are interested in the number of molecules, but not in any particular ordering of them.

Another equivalent characterisation can be given in terms of equivalence classes of strings. For each string $u \in \Gamma^\star$ and each symbol $a \in \Gamma$, let $|u|_a$ be the number of occurrences of a in u. Now let u and v be strings, and let the binary relation $\simeq$ be defined by

$$u \simeq v \qquad \text{if and only if} \qquad |u|_a = |v|_a \text{ for all } a \in \Gamma.$$

It is trivial to prove that $\simeq$ is an equivalence relation, and in particular a congruence relation with respect to the operation of string concatenation. Then, the set of equivalence classes $\Gamma^\star/\simeq$ (together with the operation induced by concatenation) is isomorphic to the set of multisets over Γ (together with the operation of multiset union). This provides us with a simple way to denote multisets in writing. Let w be a multiset, and let u be any string such that $w(a) = |u|_a$ for all $a \in \Gamma$; then, we shall identify w with $[u]_{/\simeq}$ (the equivalence class of u modulo $\simeq$) and, furthermore, we shall drop the equivalence class notation, simply writing u for w, as the formal meaning of the symbol is implied (correspondingly, multiset union will be

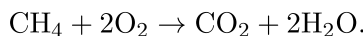
written as string concatenation).

2.3 Computation rules

The elements we have described until now, i.e., the membrane structure (including labels and charges) and the multisets contained in its regions, define the instantaneous configuration of P systems. A configuration evolves by means of computation rules inspired by the biology of cells.

2.3.1 Object evolution rules

A first kind of computation rule describes simple chemical reactions taking place inside a particular region. Consider, as an example, the following formula for the burning of methane as it may be described in a chemistry textbook:



$$\mathrm{CH_4} + 2\mathrm{O_2} \to \mathrm{CO_2} + 2\mathrm{H_2O}.$$

This notation describes how a molecule of methane (CH_4) reacts with two molecules of oxygen (O_2) yielding a molecule of carbon dioxide (CO_2) and two molecules of water (H_2O).[3] The molecules on the left-hand side of the arrow are called *reactants* or *reagents*, while those on the right-hand side are called *products*.

This kind of reaction corresponds quite closely with a limited form of *multiset rewriting*, where a certain submultiset is replaced by another. The notation is also quite similar, as for multisets we shall write $u \to v$ to indicate that multiset u is replaced by v (we use juxtaposition instead of the addition symbol used in chemistry). The general notion of multiset rewriting is more abstract, and it usually disregards the law of conservation of mass.[4]

A general chemical reaction viewed as a multiset rewriting rules possesses a powerful feature from a language-theoretic perspective: it is *context-sensitive*. For instance, methane may only burn if oxygen is also present, and we require at least *two* molecules of oxygen for each molecule of methane. The notion of multiset rewriting employed in the original model of P system [31] is indeed context-sensitive (or *cooperative*, which is the term normally used in membrane computing). P systems with active membranes, however, possess other features that make them extremely powerful from a computational perspective: as a consequence, we limit our rewriting rules to *context-free* (or *non-cooperative*) ones.

[3] The molecules involved in this reaction are, in turn, made of several atoms; however, in this discussion we consider compound objects such as H_2O as indivisible.

[4] This is an area where our model diverges from a cell as it is described in biology, although the theory could also be developed by taking conservation laws into account (probably producing results different from those described in this thesis).

The first kind of computation rules used by P systems with active membranes, called *object evolution rules* or rules of type (a), are thus denoted as follows:

$$[a \to w]_h^\alpha.$$

This rule can be applied inside each membrane labelled by $h \in \Lambda$, assuming that membrane has a particular charge $\alpha \in \{+, 0, -\}$ in the current configuration; furthermore, the membrane must contain at least a copy of object $a \in \Gamma$. When the rule is applied, a copy of object a is removed and replaced by the multiset w on the right-hand side.

The rule may not be applied if the current charge of the membrane is different from α: this implies that the rules that can be applied to membranes having label h can be changed by adjusting their charges. This task can be accomplished by applying other kinds of rules.

2.3.2 Communication rules

The main role of cell membranes is that of delimiting and separating regions where different functions are performed. However, these distinct regions must also cooperate: for instance, the products of a chemical reaction happened in membrane h_1 might be stored inside membrane h_2. The cell membranes are thus *selectively permeable*, allowing specific molecules to move between regions. When modelling this kind process, usually referred to as *communication*, we assume that only one unit of substance may move through a membrane during each time unit.

First consider the case when a unit of substance is absorbed by a membrane. This kind of rule is called *send-in communication rule*, or type (b) rule, and is formally written

$$a\ [\ ]_h^\alpha \to [b]_h^\beta.$$

This rule can be applied to a membrane having label $h \in \Lambda$ and charge $\alpha \in \{+, 0, -\}$; the region located outside the membrane must also contain at least an instance of object $a \in \Gamma$. The effects of this rule are the following: the instance of a is removed from the outside region, and a copy of object $b \in \Gamma$ appears inside the membrane; furthermore, the electrical charge of the membrane becomes $\beta \in \{+, 0, -\}$. As we can see, the object a may be rewritten into b as it is brought in, also causing a change in the state of the membrane in the process (note that we may also have $a = b$ or $\alpha = \beta$). As a restriction, we assume that no object can be brought into the outermost membrane of the P system from the external environment.

The converse kind of rule describes the process by which a membrane expels a unit of substance; these are called *send-out communication rules* or type (c) rules:

$$[a]_h^\alpha \to [\ ]_h^\beta\ b$$

This rule can be applied to a membrane labelled by $h \in \Lambda$, having charge $\alpha \in \{+, 0, -\}$ and containing a copy of object a. By applying this rule, we remove the copy of a from the membrane, change its charge to $\beta \in \{+, 0, -\}$, and place an instance of object $b \in \Gamma$ into the region immediately outside the membrane.

2.3.3 Dissolution rules

A membrane does not necessarily last as long as the whole cell itself. When its role is completed, it may *dissolve*, releasing its contents (including any membrane located inside it) to the outside, while the rest of the cell continues performing its tasks.

This kind of process is modelled in P systems by *dissolution rules*, also called type (d) rules. The notation is the following one:

$$[a]_h^\alpha \to b.$$

This rule may be applied to a membrane having label $h \in \Lambda$, charge $\alpha \in \{+, 0, -\}$ and containing at least one instance of object a. When the rule is applied, the membrane is deleted from the system: as a consequence, all its children membranes (if any) become children of its parent membrane; furthermore, the objects contained in the dissolving membrane are moved to the outside regions, while the copy of a involved in the rule is rewritten as $b \in \Gamma$. We assume that the outermost membrane of the P system cannot be dissolved.

2.3.4 Elementary division rules

We have already seen how the membrane structure of a cell needs not remain unchanged during its entire lifespan, as membranes may dissolve. However, new membranes can also be created. The quintessential example is given by the process of mitosis, whereby a whole copy of the original cell is produced by division. The process of division also occurs *internally* in cells (that is, without involving the outermost cell membrane), as mitochondria also divide by binary fission [16]. In general, division allows the creation of new "processing units" when the need arises.

Division is modelled in P systems with active membranes in two different ways, depending on whether the dividing membrane is elementary (i.e., it does not contain further membranes) or not. Let us start by considering the former, simpler case. The corresponding *elementary division rules*, or type (e) rules, have the following form:

$$[a]_h^\alpha \to [b]_h^\beta \; [c]_h^\gamma.$$

A rule of this kind can be applied to a membrane labelled by $h \in \Lambda$, having charge $\alpha \in \{+, 0, -\}$ and containing at least one instance of object $a \in \Gamma$. Its

effect is as follows: a whole new copy of the membrane is created and placed into the same region as the original one; the contents are also copied, with the exception of the instance of a mentioned above, which is replaced by an instance of $b \in \Gamma$ in one of the two resulting membranes, and by $c \in \Gamma$ in the other one. The charges of the two membranes are also set to $\beta \in \{+, 0, -\}$ and $\gamma \in \{+, 0, -\}$ respectively. This kind of rule cannot be applied to the outermost membrane.[5]

2.3.5 Nonelementary division rules

The original variant of P systems with active membranes [32] that we employ here involves a different form of division rules for nonelementary membranes. In particular, we do not replicate the *whole* internal structure; instead, the children membranes are *separated* according to their charge, and only the neutral ones are replicated.

Nonelementary division rules, or type (f) rules, have the following form:

$$[[\,]_{h_1}^{+} \cdots [\,]_{h_i}^{+} [\,]_{k_1}^{-} \cdots [\,]_{k_j}^{-}]_h^{\alpha} \to [[\,]_{h_1}^{\delta} \cdots [\,]_{h_i}^{\delta}]_h^{\beta} \; [[\,]_{k_1}^{\epsilon} \cdots [\,]_{k_j}^{\epsilon}]_h^{\gamma}.$$

This kind of rule can be applied to a membrane having label $h \in \Lambda$, charge $\alpha \in \{+, 0, -\}$. This membrane must contain positively charged children membranes labelled by $h_1, \ldots, h_i \in \Lambda$ and negatively charged children membranes labelled by $k_1, \ldots, k_j \in \Lambda$; children membranes with other labels may be present, but they must all be neutrally charged. When the rule is applied, the outermost membrane h is divided; the charge of the two copies are set to $\beta \in \{+, 0, -\}$ and $\gamma \in \{+, 0, -\}$ respectively. The internal membranes labelled by $h_1, \ldots, h_i$ remain inside the copy having charge β, and their charge is also changed to δ; the membranes labelled by $k_1, \ldots, k_j$ are moved inside the other copy of h, having charge γ, and their own charge is set to $\epsilon \in \{+, 0, -\}$. Any neutral charge membrane inside the original one is replicated (together with its internal structure) and placed inside both copies of h. Finally, the multiset of objects located inside the original membrane h is also replicated in both copies.

2.4 A formal definition

We are now able to give a precise definition of our model of computation of interest.

Definition 1. A *P system with active membranes* of initial degree $d \geq 1$ is a structure $\Pi = (\Gamma, \Lambda, \mu, w_{h_1}, \ldots, w_{h_d}, R)$ where

[5] Although division rules are mainly inspired by mitosis, here we are interested in single-cell P systems. Multi-cell P systems have also been described, e.g., those modelling tissues [17].

- Γ is the alphabet;
- Λ is the finite set of labels;
- μ is the membrane structure, a rooted undirected tree of d nodes labelled in a one-to-one way by elements of Λ;
- $w_{h_1}, \ldots, w_{h_d}$, with $h_1, \ldots, h_d \in \Lambda$, are multisets over Γ;
- R is a finite set of rules.

For each $h \in \Lambda$, the multiset w_h describes the objects initially located in the region delimited by the membrane labelled by h.

2.4.1 Configurations and computations

An instantaneous *configuration* of Π is given by the rooted unordered tree describing its current membrane structure, augmented as follows: each node is labelled by a triple (h, α, w) where

- $h \in \Lambda$ is the label of the corresponding membrane;
- $\alpha \in \{+, 0, -\}$ is the charge of the membrane;
- w is the multiset over Γ contained in the membrane.

In particular, in the initial configuration $\mathcal{C}_0$ each node is labelled by $(h, 0, w_h)$, i.e., each membrane is initially assumed to be neutrally charged.

A computation step changes the current configuration by applying the rules in a *maximally parallel way*, according to the following principles.

- Each membrane can be subject to at most one communication, dissolution or division rule per step. Any number of object evolution rules can be simultaneously applied inside the same membrane (except for the restrictions on objects described below).
- Each object can be subject to at most one rule per step.
- A maximal combination of rules must be applied during each step. Each object that appears on the left-hand side of an applicable object evolution, communication, dissolution, or elementary division rule must be subject to exactly one of them. We say that a rule is "applicable" if the label and the current charge of the membrane containing the object correspond to those appearing on the left-hand side of the rule. The only objects that do not evolve are those associated to no applicable rule (or to no rule at all).

- Each membrane that appears on the left-hand side of an applicable (i.e., having the correct labels and charges on the left-hand side) communication, dissolution, or division rule must be subject to exactly one of them. The only membranes that do not evolve are those associated to no applicable rule (or to no rule at all).

- When several maximally parallel combinations of rules can be applied at the same time, a nondeterministic choice between them is made. As a consequence, in the general case multiple possible configurations can be reached by performing a computation step.

- After a legitimate maximally parallel combination of rules has been chosen (i.e., every membrane and object that can evolve has been assigned to a rule), the rules are applied in a logical bottom-up (depth-first) way, from the elementary membranes towards the outermost one. Inside each membrane in turn, first all object evolution rules are applied, then the remaining communication, dissolution or division rule.

A *halting computation* of Π is a sequence of configurations $(\mathcal{C}_0, \ldots, \mathcal{C}_k)$ starting from the initial one such that every $\mathcal{C}_{i+1}$ is reachable from $\mathcal{C}_i$ by performing a single computation step, and no rule at all can be applied in $\mathcal{C}_k$. If Π never reaches a halting configuration, the result is an infinite, non-halting computation $(\mathcal{C}_i : i \in \mathbb{N})$.

2.5 Recogniser P systems

In this thesis we intend to investigate the use of P systems in solving decision problems, or, equivalently, to decide the membership of strings in languages. To achieve this goal, we need to define recogniser P systems.

Definition 2. A *recogniser P system* Π is a P system with the following properties:

- all computations of Π are halting;

- the alphabet Γ of Π contains two distinguished objects YES and NO;

- exactly one object between YES and NO is sent out of the outermost membrane during each computation, and only in the last computation step.

A computation $\vec{\mathcal{C}}$ of Π is said to be *accepting* if the object YES is output; it is said to be *rejecting* if the output is NO.

While P systems are defined as nondeterministic computation devices, we can impose some restriction on the amount of nondeterminism.

Definition 3. A recogniser P system is said to be *confluent* iff all its computations agree on the result (acceptance or rejection). When this is not necessarily the case, we call the P system *nonconfluent*, and its overall result is acceptance iff there exists an accepting computation, and rejection otherwise (as for nondeterministic Turing machines).

When solving decision problems, we use families of P systems instead of single ones, as it is done with other *nonuniform* computation models such as Boolean circuits [40]. From this point on, we assume that all the inputs we intend to process are given as strings over an alphabet Σ (not necessarily related to the alphabet of the P systems).

Definition 4. A *family of recogniser P systems* is a infinite set $\mathbf{\Pi} = \{\Pi_x : x \in \Sigma^\star\}$ consisting of a recogniser P systems for each possible input string (P systems corresponding to different strings need not be distinct).

A family $\mathbf{\Pi}$ is said to *recognise* or *decide* the language $L \subseteq \Sigma^\star$ (or solve the associated decision problem), in symbols $L(\mathbf{\Pi}) = L$, iff for each $x \in L$ the P system Π_x accepts, and for each $x \notin L$ the P systems Π_x rejects.

2.5.1 Uniformity conditions

An arbitrary family of recogniser P system (like a nonuniform family of circuits [23]) is not a realistic computation model, as the following example shows.

Example 5. Let Π_{YES} be a P system consisting of a single membrane h containing the object YES and having $[\text{YES}]_h^0 \to [\,]_h^0\ \text{YES}$ as its only rule; also let Π_{NO} be the P system consisting of a single membrane h containing NO and having $[\text{NO}]_h^0 \to [\,]_h^0\ \text{NO}$ as its only rule.

Now let $L \subseteq \Sigma^\star$ be an arbitrary language. Finally, for each string $x \in \Sigma^\star$ let Π_x be Π_{YES} if $x \in L$, and Π_{NO} otherwise. Then, the family $\mathbf{\Pi} = \{\Pi_x : x \in \Sigma^\star\}$ decides L *in one computation step*. But L can be an arbitrary language, including an *undecidable* one. The problem here is that we have not done any actual computation at all (except for trivially sending out a fixed object), and all the complexity is hidden into the mapping $x \mapsto \Pi_x$, which is noncomputable whenever L also is.

In order to avoid this kind of problem, we proceed as for Boolean circuits, by providing a *uniformity condition* [21], that is, by forcing the mapping $x \to \Pi_x$ to be a computable function and, in particular, an efficiently computable one. This will ensure that the family of P systems does actually contribute to the solution to the problem.

The first variant proposed in the literature is due to Obtułowicz [22], although we employ the formalisation due Pérez-Jiménez et al. [24] here.

Definition 6. A family $\mathbf{\Pi} = \{\Pi_x : x \in \Sigma^\star\}$ is said to be *(polynomial-time) semiuniform* iff the mapping $x \mapsto \Pi_x$ can be computed in polynomial time by a deterministic Turing machine M.

Any encoding of Π_x is allowed as an output of M, as long as all membranes, objects and rules are listed one by one; this is also called a *permissible encoding* [21]. We enforce this restriction in order to mimic a hypothetical "realistic" process of construction of the P systems, which presumably requires placing a constant amount of membranes and objects per computation step, and requires actual physical space proportional to their number (see also Chapter 4 for further details). As an example, writing down the membrane structure as a string of balanced brackets, the multisets as strings (i.e., in unary notation), and the rules as we did in Section 2.3 constitutes a permissible encoding. If we were allowed to use a binary notation for the multisets, as in a^3b^2c for $aaabbc$, or expressions such as "the complete binary tree with 32 levels" for the membrane structure, then an exponentially large (or even larger) P system could be created in polynomial time; this is not consistent with our idea of a "realistic" construction.

In practise, computing with a semiuniform family of P systems is done as follows: given the Turing machine M of Definition 6, for each string $x \in \Sigma^\star$ we give x as input to M, thus obtaining (the encoding of) the P system $M(x) = \Pi_x$. Then, the P system Π_x computes and provides us with the answer to the question "Is $x \in L$?" by sending out one of the answer objects YES or NO.

The usual uniformity conditions for Boolean circuits [40], however, do not map every string to a (potentially) distinct circuit. Instead, there is a single circuit C_n for all strings of the same length n, and the actual string is then given as input to the circuit of the correct size. The following definition follows the same spirit, albeit with an extra step required by the encoding of strings as multisets.

Definition 7. A family $\mathbf{\Pi} = \{\Pi_x : x \in \Sigma^\star\}$ is said to be *(polynomial-time) uniform* iff the mapping $x \mapsto \Pi_x$ can be computed as follows: there exist two polynomial-time Turing machines F (for "family") and E (for "encoding") such that

- the machine F, on input 1^n, where $n = |x|$ (i.e., the length of x in unary notation), constructs a P system Π_n with a distinguished input membrane;

- the machine E, on input x, outputs a multiset w_x (an encoding of the specific string x);

- finally, Π_x is simply Π_n but with w_x adjoined to the multiset placed inside its input membrane.

In this thesis, we shall only use this kind of uniformity condition. As required, the P system Π_n of Definition 7 is common for all strings of length n. This differs from (and is possibly more restricted than) another notion of uniformity commonly employed in the literature [24], in which the machine F maps each input x to a P system $\Pi_{s(x)}$, where $s\colon \Sigma^\star \to \mathbb{N}$ is a "size function" for the input. In our definition, $s(x)$ is always the length function $|x|$. Even more restrictive notions of uniformity can be obtained by replacing the polynomial-time Turing machines E and F by weaker devices. For instance, the original notion of semiuniformity actually used logarithmic space machines; Murphy and Woods [21] define complexity classes for P systems parametric with respect to the chosen uniformity conditions, and usually employ weak $\mathbf{AC}^0$ circuits as a concrete choice. This is useful when comparing the computing power of P systems with "small" complexity classes included in $\mathbf{P}$, such as $\mathbf{AC}^0$, $\mathbf{L}$, and $\mathbf{P}$ itself. On the other hand, in this thesis we stick to polynomial-time uniformity, as we are interested in complexity classes larger than $\mathbf{P}$ such as $\mathbf{PSPACE}$, $\mathbf{PP}$ and the polynomial hierarchy $\mathbf{PH}$; however, most of our uniform constructions can probably be done in logarithmic space (and some possibly even by $\mathbf{AC}^0$ circuits).

2.6 Time complexity of P systems

Having shown how to solve decision problems by means of uniform families of P systems, we are also interested in measuring the efficiency of these computing devices, and how it compares to standard computing devices such as Turing machines with various time and space restrictions [24].

We begin by defining precisely the time complexity of a P system and a family of P systems.

Definition 8. A recogniser P system is said to work in k steps if all its computations have length at most k.

A family $\mathbf{\Pi} = \{\Pi_x : x \in \Sigma^\star\}$ of recogniser P systems is said to work in time $f\colon \mathbb{N} \to \mathbb{N}$ if, for each $n \in \mathbb{N}$ and each $x \in \Sigma^n$, the P system Π_x works in $f(n)$ steps.

It is now easy to define complexity classes for P systems [24] similarly to those defined in terms of Turing machines [23].

Definition 9. Let $f\colon \mathbb{N} \to \mathbb{N}$. The class of problems solvable by uniform families of confluent P systems with active membranes in time f is denoted by $\mathbf{MC}_{\mathcal{AM}}(f)$. The corresponding class for uniform families of nonconfluent P systems with active membranes is $\mathbf{NMC}_{\mathcal{AM}}(f)$.

As usual in complexity theory, one of the most interesting classes is defined by those problems that can be solved in a time-efficient way, i.e., in polynomial time.

Definition 10. The class of problems solvable in polynomial time by uniform families of confluent P systems with active membranes is denoted by $\mathbf{PMC}_{\mathcal{AM}}$ (and by $\mathbf{NPMC}_{\mathcal{AM}}$ in the nonconfluent case).

When some types of rules are avoided, we will use another notation instead of $\mathcal{AM}$ when discussing the related complexity classes. In particular, $\mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$ will denote the class of problems solvable in polynomial time by uniform families of confluent *P systems with restricted elementary active membranes*, where dissolution and nonelementary division rules are not used.

Chapter 3

Computing with P systems

Can P systems be used in order to compute "interesting" functions? The answer to this question is strikingly positive, as it turns out they are as powerful as Turing machines. This fact was already mentioned in the original paper [32]; here we prove this result by simulating another class of simple, computationally universal devices, namely *register machines*. Furthermore, we also show how P systems with membrane division can be used to solve **NP**-complete problems in polynomial time by exploiting the time-space trade-off that gives this thesis its title.

3.1 P systems are universal

Turing machines are a very simple and convincing mathematical formalism to describe how computations can be carried out by a human executor provided with a pencil and enough scratch paper. Although they were an important inspiration for the design of electronic computers [7], these turned out to be constructed with a very different internal structure. In particular, digital computers normally have a random access memory, instead of a linear one like Turing machines.

Register machines, also called *counter machines* or *program machines* [20], are a model of computation which more accurately describes the functioning of modern electronic computers. Register machines consist of a constant, usually small number of registers able to contain arbitrary large natural numbers[1]; a fixed number of instructions (labelled by consecutive natural numbers) describes the way a register machine operates, and a program

[1] Actual digital computers, instead, have a large number of "registers" (memory locations) able to contain constant-sized integers. In a computability-theoretic context, "large" translates to "potentially infinite", and being able to randomly address a potentially infinite memory space would require arbitrarily large registers in order to store pointers to memory locations. *Random access machines* [23], a more sophisticated model of computation we will make use of later, are defined this way.

counter keeps track of the label of the instruction to be executed. The instructions themselves are of three very simple kinds.

- Increment the value of a fixed register r and proceed with a specified instruction i; we denote this kind of instruction by $\textsc{inc}(r), i$.
- Test the value of a fixed register r: if it is nonzero, then it is decremented by one and the execution proceeds with a first specified instruction i; otherwise, the execution jumps to an alternate specified instruction j. This is denoted by $\textsc{dec}(r), i, j$.
- Finally, halt the computation; we denote this instruction by HALT.

Some of the registers of the machine, say the first m, are conventionally chosen (on a machine-by-machine basis) to contain the input, while n of them are devoted to the output. All non-input registers are assumed to be initially null. The register machine starts computing from its first instruction, having label 1, and keeps executing instruction until a halting one is reached. This way, a register machine R can be thought of as computing a partial function $f \colon \mathbb{N}^m \rightharpoonup \mathbb{N}^n$; as usual, we assume that f is undefined on a particular m-tuple if and only if R does not halt when given that input.

Example 11. Assume that a and b are the input registers of machine R, and that c is the output register. Also, let z be an auxiliary register. Then, the following instructions compute the sum of a and b:

$1\colon \textsc{dec}(a), 2, 3$
$2\colon \textsc{inc}(c), 1$
$3\colon \textsc{dec}(b), 4, 5$
$4\colon \textsc{inc}(c), 3$
$5\colon$ HALT

The first two instructions 1–2 form a loop that decrements a as long as it is possible, while simultaneously incrementing c. When a becomes null, the executions proceeds to the loop of instructions 3–4, which performs an analogous task on register b. When b is also null, the execution jumps to the halting instruction 5.

Register machines are computationally equivalent to Turing machines; mutual simulations are described by [20]. Nonetheless, they are (in a way) even simpler than Turing machines: this is precisely why we show how to simulate them [34] in order to prove the computational universality of P systems.

3.1.1 Simulating register machines

In order to simulate register machines via P systems, we exploit one of the defining features of the latter devices: namely, that the contents of a

region are described by a *multiset* of objects, where the multiplicity of each occurring object does actually matter.

Let R be a register machine having k registers $r_1, \dots, r_k$. We represent each register r by a membrane, also labelled by r; these are enclosed by an outermost membrane s. Hence the P system Π_R, whose task is simulating R, has the following initial membrane structure:

$$\mu_R = \big[[\,]^0_{r_1} \cdots [\,]^0_{r_k}\big]^0_s.$$

The value of each register r is represented by the multiplicity of an object a inside the corresponding membrane r. Our simulation will also employ another object p_i, initially located inside the outermost membrane s. This object represents the program counter of R, and its subscript i denotes the label of the instruction currently being simulated.

If $r_1, \dots, r_\ell$ are the input registers of R, and the input values are $\vec{n} = (n_1, \dots, n_\ell)$, the P system Π_R simulating the computation of R on input $\vec{n}$ has the following initial configuration:

$$\mathcal{C}_0 = \big[[a^{n_1}]^0_{r_1} \cdots [a^{n_\ell}]^0_{r_\ell} [\,]^0_{r_{\ell+1}} \cdots [\,]^0_{r_k} p_1\big]^0_s.$$

The evolution rules of Π_R are defined from the instructions of R. Consider an increment instruction $i\colon \textsc{inc}(r), j$. This requires the creation of a new instance of a inside membrane r, and replacing the object p_i by p_j inside the outermost membrane. The former task is accomplished by the object p_i itself. First of all, we move it to membrane r by using a communication rule:

$$p_i\ [\,]^0_r \to [p_i]^0_r.$$

When it is inside membrane r, it produces a copy of a, while simultaneously "priming" itself in order to avoid entering an infinite loop:

$$[p_i \to a p'_i]^0_r.$$

Finally, the object p'_i is sent back out as p_j:

$$[p'_i]^0_r \to [\,]^0_r\ p_j.$$

It is easy to see that, by applying these three rules, our "simulation invariant" (membrane r contains as many a's as the value of register r, and the subscript of p_j corresponds to the label of the instruction to simulate) is restored.

Now consider a decrement instruction $i\colon \textsc{dec}(r), j, k$. In this case, we also send the object p_i to membrane r, but this time we change its charge to negative:

$$p_i\ [\,]^0_r \to [p_i]^-_r.$$

When the charge of r is negative, one copy of a (if any appears) is sent out from the membrane, while restoring its neutral charge:

$$[a]_r^- \to [\;]_r^0\; a.$$

Thus, the charge of r remains negative if and only if no instance of a is found inside it, due to the maximal application of rules: the program counter object can use this fact to choose the next value of its subscript. However, in order to avoid interference with the rule $[a]_r^- \to [\;]_r^0\; a$, the object p_i has to "wait" one step first; we use an evolution rules to implement this:

$$[p_i \to p_i']_r^- .$$

Now p_i' can be sent out from r as p_j or p_k, depending on the charge of the membrane (which is set back to neutral in both cases):

$$[p_i']_r^0 \to [\;]_r^0\; p_j \qquad\qquad [p_i']_r^- \to [\;]_r^0\; p_k.$$

After this last step, the "simulation invariant" is restored, and the next instruction of R can be simulated.

As for the third and final kind of instruction, i: HALT, the P system Π_R does not need any rule: this way, when the subscript of the program counter object reaches i, the computation of Π_R also halts.

This shows that register machines can be simulated by P systems using only a subset of the available rules, namely object evolution and both kinds of communication rules. The simulation is also quite fast, as each instruction executed by a register machine requires at most three steps in a P system.

Theorem 12. *Let R be a register machines having input registers $r_1, \ldots, r_\ell$ and output registers $r_{\ell+1}, \ldots, r_m$. Then there exists a P system Π_R, using only object evolution and communication rules, with the following properties: when its membranes $r_1, \ldots, r_\ell$ are initialised by putting $n_1, \ldots, n_\ell$ copies of object a inside them, the P system computes the same function as R on input $(n_1, \ldots, n_\ell)$; that is, the number of instances of a found inside membranes $r_{\ell+1}, \ldots, r_m$ of Π_R when it halts correspond to the values of the output registers of R, assuming R halts on that input. Whenever R does not halt, neither does Π_R. Furthermore, the time required by Π_R is the same as R but for a small multiplicative constant.* □

Corollary 13. *P systems are computationally universal, even if only object evolution and communication rules are allowed.* □

3.2 Solving NP-complete problems in polynomial time

In order to provide an example illustrating the way we can exploit membrane division and maximal parallelism of P systems to solve computationally hard problems in polynomial time, we consider the 3SAT decision problem [23].

We are given a Boolean formula φ in *ternary conjunctive normal form* (3CNF), that is, a formula consisting of a conjunction of clauses, each one consisting in turn of a disjunction of exactly three literals; each literal is either a variable, or a negated variable. We also assume that each variable occurs at most once per clause. Our task is deciding whether φ admits at least one satisfying assignment.

The 3SAT problem is one of the standard **NP**-complete problems [15], and as such it is generally believed not to be solvable in polynomial time by a deterministic Turing machine. On the other hand, the ability to make nondeterministic choices allows us to solve 3SAT efficiently. Here we replace nondeterminism with membrane division, which (together with maximal parallelism) allows us to test simultaneously all possible assignments.

3.2.1 Encoding of Boolean formulae

We start by defining a simple encoding of formulae as binary strings, under which formulae having the same number of variables have equally long representations. This will help us design a uniform family of P system solving the problems.

Given a set of m variables, we can choose three of them in $\binom{m}{3}$ ways; the three variables can be negated in 2^3 possible ways. Hence, the number of possible clauses (as defined above) is $8\binom{m}{3}$. Now fix an arbitrary total ordering on the set of clauses; any ordering will do, as long as, for all $i \in \{1, \ldots 8\binom{m}{3}\}$, we are able to identify the i-th clause in polynomial time with respect to m. An appropriate ordering is given, for instance, by the algorithm in Figure 3.1 which, for $m = 4$, produces the following output:

$$
\begin{array}{rrrr}
x_1 \vee x_2 \vee x_3 & x_1 \vee x_2 \vee \neg x_3 & x_1 \vee \neg x_2 \vee x_3 & x_1 \vee \neg x_2 \vee \neg x_3 \\
\neg x_1 \vee x_2 \vee x_3 & \neg x_1 \vee x_2 \vee \neg x_3 & \neg x_1 \vee \neg x_2 \vee x_3 & \neg x_1 \vee \neg x_2 \vee \neg x_3 \\
x_1 \vee x_2 \vee x_4 & x_1 \vee x_2 \vee \neg x_4 & x_1 \vee \neg x_2 \vee x_4 & x_1 \vee \neg x_2 \vee \neg x_4 \\
\neg x_1 \vee x_2 \vee x_4 & \neg x_1 \vee x_2 \vee \neg x_4 & \neg x_1 \vee \neg x_2 \vee x_4 & \neg x_1 \vee \neg x_2 \vee \neg x_4 \\
x_1 \vee x_3 \vee x_4 & x_1 \vee x_3 \vee \neg x_4 & x_1 \vee \neg x_3 \vee x_4 & x_1 \vee \neg x_3 \vee \neg x_4 \\
\neg x_1 \vee x_3 \vee x_4 & \neg x_1 \vee x_3 \vee \neg x_4 & \neg x_1 \vee \neg x_3 \vee x_4 & \neg x_1 \vee \neg x_3 \vee \neg x_4 \\
x_2 \vee x_3 \vee x_4 & x_2 \vee x_3 \vee \neg x_4 & x_2 \vee \neg x_3 \vee x_4 & x_2 \vee \neg x_3 \vee \neg x_4 \\
\neg x_2 \vee x_3 \vee x_4 & \neg x_2 \vee x_3 \vee \neg x_4 & \neg x_2 \vee \neg x_3 \vee x_4 & \neg x_2 \vee \neg x_3 \vee \neg x_4
\end{array}
$$

Once an ordering has been fixed, a formula φ of m variables can be encoded as a string $\ulcorner \varphi \urcorner$ of $n = 8\binom{m}{3}$ bits, where the i-th bit is set if and only if the i-th clause under that ordering appears in φ. Furthermore, all and only the strings of length $8\binom{m}{3}$ are well-formed, and it easy to check that an instance is well-formed, as well as recover m from n, simply by finding the

```
PRINT-CLAUSES(m)
    IF m > 3 THEN
        PRINT-CLAUSES(m − 1)
    END
    FOR i ← 1 TO m − 2 DO
        FOR j ← i + 1 TO m − 1 DO
            PRINT "x_i ∨ x_j ∨ x_m"
            PRINT "x_i ∨ x_j ∨ ¬x_m"
            PRINT "x_i ∨ ¬x_j ∨ x_m"
            PRINT "x_i ∨ ¬x_j ∨ ¬x_m"
            PRINT "¬x_i ∨ x_j ∨ x_m"
            PRINT "¬x_i ∨ x_j ∨ ¬x_m"
            PRINT "¬x_i ∨ ¬x_j ∨ x_m"
            PRINT "¬x_i ∨ ¬x_j ∨ ¬x_m"
        END
    END
END
```

Figure 3.1: A recursive, polynomial time algorithm enumerating all ternary clauses over m variables.

unique positive root of the polynomial

$$p(m) = 8\binom{m}{3} - n = \frac{4}{3}m^3 - 4m^2 + \frac{8}{3}m - n.$$

This can be accomplished, for instance, by trying all values of m in the range $(0, n]$: this procedure runs in polynomial time with respect to the length of the encoding $\ulcorner\varphi\urcorner$, and if no m such that $p(m) = 0$ is found, then the input can be safely rejected as malformed.

3.2.2 Solution to 3SAT

The algorithm for solving 3SAT via P systems we are going to describe is based on the first polynomial-time solution to the SAT problem that only uses elementary membrane division (instead of both elementary and nonelementary division) due to Zandron et al. [42]. The following is a high-level description of the procedure.

Algorithm 14. Solving 3SAT on input φ, a 3CNF Boolean formula of m variables.

A *Generate* 2^m membranes using elementary division, each one containing objects representing a different truth assignment to the variables of φ.

B *Evaluate* φ under the 2^m assignments, in parallel, and send out from each membrane an object s_n whenever the assignment contained in that membrane satisfies φ.

C *Output* YES if at least one instance of s_n has been sent out, and NO otherwise.

Assume we are given a well-formed encoding $\ulcorner\varphi\urcorner$ of φ having length n. The machine F of Definition 7, given 1^n as input, construct a P system with input membrane Π_n able to process all instances of size n. Its initial configuration is the following one:

$$\mathcal{C}_0 = \big[[\text{x}_1 \cdots \text{x}_m \ \text{w}_m]^0_{\text{E}} \ \text{o}_t \ \text{no}_{t+2}\big]^0_{\text{S}}$$

where $t = m + 3n + 3$ (as we shall see later, this number is related to the duration of Phases A and B described above). The membrane labelled by E is the input membrane of Π_n.

The "encoding" machine E of Definition 7 takes as input the whole string $\ulcorner\varphi\urcorner$ and translates the string into a multiset encoding of φ as follows:

$$E(\ulcorner\varphi\urcorner) = \{\text{c}_i : \text{the } i\text{-th clause does } \textit{not} \text{ appear in } \varphi, \text{ for } 1 \le i \le 8\tbinom{m}{3}\}$$

that is, a multiset describing the clauses missing in φ. This multiset is placed inside membrane E of Π_n, and then the actual computation may begin. The details of the three phases of Algorithm 14, including the rules output by F, are described below.

Phase A (Generate) The variable-objects $\text{x}_1, \dots, \text{x}_m$ are exploited in order to divide the membrane containing them m times. Each division also causes the value of one of the variables to be fixed, by rewriting it as a T object on one side, and as a F object on the other. The corresponding division rules are

$$[\text{x}_i]^0_{\text{E}} \to [\text{t}_i]^0_{\text{E}} \ [\text{f}_i]^0_{\text{E}} \qquad \text{for } 1 \le i \le m.$$

These rules are applied for m steps, after which 2^m copies of membrane E exist, and each of them contains a multiset representing a different truth assignment to φ: the occurrence of t_i (resp., f_i) indicates that the variable x_i is set to true (resp., false). Notice that one variable-object is chosen nondeterministically during each of these m steps, but the final result is independent of the sequence of choices made. All the input objects c_j are replicated among all copies of E.

While the membranes labelled by E divide, the subscript of object w_i is used to count down to 0 by using the following object evolution rules:

$$[\text{w}_i \to \text{w}_{i-1}]^0_{\text{E}} \qquad \text{for } 1 \le i \le m.$$

At the end of this phase, we have 2^m membranes labelled by E, and each of them contains either object t_i or f_i for all $i \in \{1, \dots, m\}$; a different truth assignment can be found in each copy of membrane E. The duration of Phase A is m computation steps.

Phase B (Evaluate) After the last membrane division in Phase A has occurred, each copy of membrane E contains an instance of w_0, which is sent out in order to change the charge of those membranes to +. This is accomplished by using the following communication rules:

$$[\mathrm{w}_0]_{\mathrm{E}}^{0} \to [\,]_{\mathrm{E}}^{+}\ \mathrm{w}_0.$$

When the E membranes are positively charged, each object T_i and F_i is rewritten into a set of objects representing all clauses (whether they actually appear in φ or not) that are satisfied by setting the corresponding variable to that truth value:

$[\mathrm{T}_i \to \mathrm{C}_{i_1} \cdots \mathrm{C}_{i_\ell}]_{\mathrm{E}}^{+}$ for $1 \le i \le m$, where clause i_j contains the literal x_i
$[\mathrm{F}_i \to \mathrm{C}_{i_1} \cdots \mathrm{C}_{i_\ell}]_{\mathrm{E}}^{+}$ for $1 \le i \le m$, where clause i_j contains the literal $\neg x_i$.

Here we have $\ell = 4\binom{m-1}{2} = 2m^2 - 6m + 4$, as this is the number of clauses over m variables that contain a particular literal. Notice that the set of clauses satisfied by setting each variable to a particular truth value can be computed in polynomial time by enumerating all clauses and checking which literals they contain (e.g., by using a simple variant of the algorithm of Figure 3.1); this set only depends on the length n, and not on the actual input formula φ

In the same computation step, while the clause-objects C_j are introduced, the objects w_0 are brought back into the membranes labelled by E as s_0. Each instance of E gets exactly one copy of s_0, as only one communication rule per step may be applied to a membrane, and only one step is needed, as the communication rules are applied in a maximally parallel way. The corresponding communication rule is

$$\mathrm{w}_0\ [\,]_{\mathrm{E}}^{+} \to [\mathrm{s}_0]_{\mathrm{E}}^{+}.$$

Now each copy of membrane E contains the following objects:

- One or more instances of C_i, $1 \le i \le m$, for each possible clause i over m variables (not necessarily appearing in φ) that is satisfied by the truth assignment corresponding to that particular membrane.
- At least one copy of C_i for each clause i that does *not* appear in φ, whether it is satisfied by the truth assignment or not: a missing clause is always considered to be satisfied. These are the objects that have been given as input to Π_n.
- An instance of object s_0.

The occurrence of object s_i (for some $1 \le i \le n$) inside a copy of membrane E denotes the fact that the first i clauses of m variables have been found

to be satisfied by the truth assignment corresponding to that membrane (possibly because they are altogether missing from φ).

From this point on, whenever a membrane E is positive, a copy of object c_1 (if present) is sent out from E as a "junk" object # in order to switch the charge to negative:

$$[c_1]_E^+ \to [\,]_E^- \; \#. \tag{3.1}$$

When the membranes labelled by E are negative, all objects $c_2, \dots, c_n$ are temporarily "primed":

$$[c_i \to c_i']_E^- \qquad \text{for } 2 \le i \le n.$$

and, simultaneously, all remaining copies of object c_1 are discarded:

$$[c_1 \to \#]_E^-.$$

Object s_i is also sent out:

$$[s_i]_E^- \to [\,]_E^- \; s_i \qquad \text{for } 0 \le i \le n-1.$$

During the next computation step, each object c_i' becomes c_{i-1}: this way, checking whether all clauses are satisfied (or missing) only amounts to repeatedly checking whether object c_1 occurs.

$$[c_i' \to c_{i-1}]_E^- \qquad \text{for } 2 \le i \le n.$$

At the same time, the object s_i is brought back in as s_{i+1} (as long as $i < m$), denoting the fact that a further clause has been found to be satisfied, and simultaneously restoring the positive charge of membrane E:

$$s_i \; [\,]_E^- \to [s_{i+1}]_E^+.$$

Now the rules starting at (3.1) can be executed again to check for the next clause. Whenever a clause does actually appear in φ, but it is not satisfied by the assignment corresponding to a particular membrane E, the object c_1 will be missing from the membrane at some point; that membrane will stop computing altogether, and the subscript of the corresponding s_i object will never reach n. On the other hand, if a membrane E initially contains at least an instance of all possible clause-objects $c_1, \dots, c_n$, the object s_n will eventually be found inside it.

The total duration of Phase B is $3n + 2$ steps.

Phase C (Output) At the beginning of the last phase, the objects s_n are sent out (without changing the charge of the membrane E they are located in) by using the rule

$$[s_n]_E^+ \to [\,]_E^+ \; s_n.$$

Now, the outermost membrane s contains at least one copy of s_n if and only if the formula φ admits a satisfying assignment.

We have not described yet the behaviour of the objects o_t and no_t located inside membrane s. During all the computation until now they implement timers counting down to zero, according to the following object evolution rules:

$$[\mathrm{o}_i \to \mathrm{o}_{i-1}]_{\mathrm{s}}^{0} \qquad \text{for } 1 \le i \le t$$
$$[\mathrm{no}_i \to \mathrm{no}_{i-1}]_{\mathrm{s}}^{0} \qquad \text{for } 1 \le i \le t+2.$$

The subscript of o_i reaches 0 exactly during the last step of Phase B. In the next step, while the objects s_n are sent out, it is in turn sent out from s while changing its charge to positive:

$$[\mathrm{o}_0]_{\mathrm{s}}^{0} \to [\;]_{\mathrm{s}}^{+}\ \#.$$

If any object s_n is found inside the outermost membrane, it is sent out during the next step as YES, changing the charge of the membrane to negative and halting the computation in a successful way:

$$[\mathrm{s}_n]_{\mathrm{s}}^{+} \to [\;]_{\mathrm{s}}^{-}\ \mathrm{YES}.$$

On the other hand, if not s_n is found, then membrane s is still positively charged after the subscript of no_i has reached 0 (i.e., after step $t+2$). In this case, that object is sent out as NO, halting the computation by rejecting:

$$[\mathrm{no}_0]_{\mathrm{s}}^{+} \to [\;]_{\mathrm{s}}^{-}\ \mathrm{NO}.$$

The duration of Phase C is either 2 or 3 steps, depending on whether φ is satisfiable or not.

Theorem 15. *The problem* 3SAT *can be solved in polynomial time by a uniform family of confluent P systems with active membranes (without using dissolution or nonelementary division rules). In symbols,* 3SAT $\in \mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$.

Proof. Each P system Π_n, once it has been provided with the encoding $E(\ulcorner\varphi\urcorner)$ as input multiset, decides whether φ has a satisfying assignment in $m+(3n+2)+2$ or $m+(3n+2)+3$ computation steps, which is polynomial (actually, linear) time with respect to the input size. The behaviour of each P system is confluent, as the only nondeterministic choices (made in Phase A) always produce the same configuration at the beginning of Phase B, and the rest of the computation is deterministic.

Furthermore, the whole family $\mathbf{\Pi} = \{\Pi_x : x \in \Sigma^\star\}$ solving 3SAT is uniform. Indeed, the initial configuration of Π_n (without input multiset) and the rules described above only depend on the input length n, and can be computed in polynomial time by a deterministic Turing machine F. The multiset encoding the input formula φ can also computed in polynomial time (given $\ulcorner\varphi\urcorner$) by another deterministic Turing machine E. $\square$

Notice that we have not stated yet that this result implies $\mathbf{NP} \subseteq \mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$, as the latter class is not currently known to be closed under polynomial-time reductions. We will deal with this technical detail in Chapter 5.

Chapter 4

Space complexity of P systems

A large part of the literature on the complexity-theoretic properties of P systems with active membranes investigate polynomial-time solutions to classically intractable problems. In most cases this requires a trade-off: polynomial time solutions are achieved by creating exponentially many membranes (by membrane division) and exploiting maximal parallelism in order to let them compute simultaneously, as for the solution of 3SAT described in the previous chapter. In order to better characterise this trade-off and *prove* actual theorems on the subject, we start by formalising the notion of *space complexity* for P systems [25]. Although our focus here is on P systems with active membranes, most definitions can be carried over to other models, with the due adjustments. We then prove that P systems with active membranes operating in polynomial space have exactly the same computing power as Turing machines working in polynomial space.

4.1 Preliminaries

P systems are, in principle, defined an abstraction of biological cells, although some features deviate from actual biology for reasons of generality or elegance. As a consequence, our aim is to define a notion of space complexity that keeps a "scientific" spirit as far as it is possible, while being at the same time simple enough to admit a mathematical treatment. An analogy is given by the usual space complexity measure for Turing machines [23], that amounts to counting the number of tape cells; while very simple, this notion of space complexity is clearly related to the amount of paper needed by the human computer postulated by Turing himself [38], who carries out the computation by hand.

In our case, we wish our space complexity measure to be related to the physical space occupied by a cell. First of all, consider the molecules

forming the cell that are not part of a membrane, which are represented by symbol-objects in P systems. Here we adopt a very simple molecular model, where the volume of the molecule is proportional to the volume of its constituent atoms. We also make a further simplification: since we are mostly dealing with polynomial-time uniform families of P systems, at most a polynomial number of different molecular species may be involved at one time; assuming we can afford to "choose" the smallest molecules available for our computation, the largest one is going to be at most polynomially larger than the smallest one.[1] Disregarding these polynomial variations in size, we postulate that *all molecules* (i.e., objects) *have unit size*.

The volume required by a membrane usually depends on the volume of its contents. An approximation can be given as follows. Assume that the membrane has spherical shape and it is filled by n molecules of unit size; its interior radius is then $r = \sqrt[3]{\frac{3n}{4\pi}} = \Theta(\sqrt[3]{n})$ and its surface $A = 4\pi r^2 = \Theta(n^{2/3})$. The volume required by the membrane itself is then approximately A multiplied by its thickness. Assuming a constant thickness for all membranes, this volume is still $\Theta(n^{2/3})$, which is $o(n)$; this means that, asymptotically, the total volume of a membrane *and* the molecules it contains, taken together, is mostly due to the molecules. As we are dealing with the space required asymptotically by P systems, we might be tempted to completely ignore the volume of the membranes themselves and only focus on their contents. On the other hand, we do not want to disregard the case of an *empty* membrane, whose volume is not null. As a compromise, we shall adopt the following principle: *each membrane also has unit size.*

The resulting notion of space complexity for the instantaneous configuration of a P system is then immediate.

Definition 16. Let $\mathcal{C}$ be the instantaneous configuration of a P system with active membranes. The *size of* $\mathcal{C}$, in symbols $|\mathcal{C}|$, is given by the sum of the number of membranes constituting the membrane structure and the total number of objects it contains.[2]

4.2 The notion of space complexity

Having defined the size of an instantaneous configuration of a P system, we need to establish how much space is required by a whole computation.

Definition 17. Let $\vec{\mathcal{C}} = (\mathcal{C}_0, \ldots, \mathcal{C}_k)$ be a halting computation of a P system.

[1] Where these simplifications turn out to be unrealistic, more sophisticated measurements may be employed. This kind of analysis is beyond the scope of this work.

[2] Different notions of space might be more adequate when P systems are not considered as "physical objects" but, for instance, as models of cells in a computer simulation of some biological phenomenon. One such alternative measure might be simply given by "the number of bits required in order to store the current configuration".

The *space required by* $\vec{\mathcal{C}}$, in symbols $|\vec{\mathcal{C}}|$, is given by

$$|\vec{\mathcal{C}}| = \max\{|\mathcal{C}_0|, \dots, |\mathcal{C}_k|\}.$$

If $\vec{\mathcal{C}} = (\mathcal{C}_i : i \in \mathbb{N})$ is an *infinite* computation, by extension the space required by it is given by

$$|\vec{\mathcal{C}}| = \sup\{|\mathcal{C}_i| : i \in \mathbb{N}\}.$$

Non-halting computations require an infinite amount of space, i.e., $|\vec{\mathcal{C}}| = \infty$, when the sequence $|\mathcal{C}_0|, |\mathcal{C}_1|, \dots$ is unbounded.

When discussing the overall space requirements $|\Pi|$ of a P system Π, in general multiple computations must be taken into account. Clearly, if only one computation $\vec{\mathcal{C}}$ is possible (the P system is strictly deterministic) then a sensible definition is simply $|\Pi| = |\vec{\mathcal{C}}|$. In the confluent case, when we have multiple computations leading to the same result, we opt for a worst-case analysis, assuming that any given computation may occur. In this thesis, we make the same choice even when nonconfluent P systems are considered. Alternative definitions, based for instance on the space of the computation that minimises the requirements, are of course possible; whether the use of these alternative notions changes the actual results in a significant way is left as an open problem.

Definition 18. Let Π be a P system with active membranes. The *space required by* Π is defined as

$$|\Pi| = \sup\{|\vec{\mathcal{C}}| : \vec{\mathcal{C}} \text{ is a computation of } \Pi\}.$$

The case $|\Pi| = \infty$ may occur when Π admits non-halting computations requiring infinite space, or an infinite set of halting computations requiring unbounded space.

Finally, we can define the space requirements of a *family* of P systems, a parameter that will allow us to classify decision problems in terms of their space complexity with respect to membrane computing solutions.

Definition 19. Let $\mathbf{\Pi} = \{\Pi_x : x \in \Sigma^\star\}$ be a family of recogniser P systems with active membranes, and let $f\colon \mathbb{N} \to \mathbb{N}$. We say that $\mathbf{\Pi}$ *operates within space* f iff $|\Pi_x| \leq f(|x|)$ holds for all $x \in \Sigma^\star$.

We can now define complexity classes in terms of families of recogniser P systems.

Definition 20. Let $f\colon \mathbb{N} \to \mathbb{N}$. The class of problems solvable by uniform families of confluent recogniser P systems with active membranes operating within space f is denoted by $\mathbf{MCSPACE}_{\mathcal{AM}}(f)$. The corresponding class for families of nonconfluent P systems is denoted by $\mathbf{NMCSPACE}_{\mathcal{AM}}(f)$.

Definition 21. The class of problems solvable in *polynomial space* by uniform families of confluent (resp., nonconfluent) recogniser P systems with active membranes is denoted by $\mathbf{PMCSPACE}_{\mathcal{AM}}$ (resp., $\mathbf{NPMCSPACE}_{\mathcal{AM}}$).

The class of problems solvable in *exponential space* by uniform families of confluent (resp., nonconfluent) recogniser P systems with active membranes is denoted by $\mathbf{EXPMCSPACE}_{\mathcal{AM}}$ (resp., $\mathbf{NEXPMCSPACE}_{\mathcal{AM}}$).

4.2.1 Basic results

A few properties of the space complexity classes for P systems follow immediately from their definitions [25]. We have the obvious set-theoretic ones, such as

$$\mathbf{PMCSPACE}_{\mathcal{AM}} \subseteq \mathbf{EXPMCSPACE}_{\mathcal{AM}}$$
$$\mathbf{NPMCSPACE}_{\mathcal{AM}} \subseteq \mathbf{NEXPMCSPACE}_{\mathcal{AM}}$$

and, given $f, g \colon \mathbb{N} \to \mathbb{N}$, if g is pointwise larger than f then

$$\mathbf{MCSPACE}_{\mathcal{AM}}(f) \subseteq \mathbf{MCSPACE}_{\mathcal{AM}}(g)$$
$$\mathbf{NMCSPACE}_{\mathcal{AM}}(f) \subseteq \mathbf{NMCSPACE}_{\mathcal{AM}}(g).$$

Other properties are due to the fact that confluence is actually a special case of nonconfluence; hence, for all $f \colon \mathbb{N} \to \mathbb{N}$ we have

$$\mathbf{MCSPACE}_{\mathcal{AM}}(f) \subseteq \mathbf{NMCSPACE}_{\mathcal{AM}}(f)$$

and, as a consequence,

$$\mathbf{PMCSPACE}_{\mathcal{AM}} \subseteq \mathbf{NPMCSPACE}_{\mathcal{AM}}$$
$$\mathbf{EXPMCSPACE}_{\mathcal{AM}} \subseteq \mathbf{NEXPMCSPACE}_{\mathcal{AM}}.$$

However, due to the nature of P systems (in particular with respect to object evolution and division rules) we do not have immediate results relating time complexity classes, such as $\mathbf{PMC}_{\mathcal{AM}}$, to space complexity classes, such as $\mathbf{PMCSPACE}_{\mathcal{AM}}$; compare this to Turing machines, where the inclusion $\mathbf{P} \subseteq \mathbf{PSPACE}$ trivially follows from the definitions of those classes [23]. Closure under polynomial time reductions also seems to be nontrivial (and it is actually not yet known to hold for our definition of uniformity[3]).

It is, instead, very easy to prove that the confluent space complexity classes are closed under complement.

Proposition 22. *For all* $f \colon \mathbb{N} \to \mathbb{N}$ *we have*

$$\mathbf{MCSPACE}_{\mathcal{AM}}(f) = \mathbf{coMCSPACE}_{\mathcal{AM}}(f)$$

[3]Closure under polynomial-time reductions for semi-uniform complexity classes, or for the more general notion of uniformity, are instead easy to prove [24].

As a consequence,

$$\mathbf{PMCSPACE}_{\mathcal{AM}} = \mathbf{coPMCSPACE}_{\mathcal{AM}}$$
$$\mathbf{EXPMCSPACE}_{\mathcal{AM}} = \mathbf{coEXPMCSPACE}_{\mathcal{AM}}.$$

Proof. Let $L \in \mathbf{MCSPACE}_{\mathcal{AM}}(f)$ be decided by a polynomial-space uniform family of P systems $\mathbf{\Pi} = \{\Pi_x : x \in \Sigma^\star\}$. For each $x \in \Sigma^\star$, define $\overline{\Pi}_x$ to be exactly like Π_x, except that every time the object YES is mentioned in its definition (both as an object in a multiset and in a rule) it is replaced by NO, and similarly every instance of NO is replaced by YES. Clearly, $\overline{\Pi}_x$ behaves exactly like Π_x (including its time and space requirements) but it rejects whenever Π_x accepted, and accepts whenever Π_x rejected. Hence, $\overline{\mathbf{\Pi}} = \{\overline{\Pi}_x : x \in \Sigma^\star\}$ decides $\overline{L}$, and it is also uniform (we just need to exchange YES and NO during the construction phase).

Since L was arbitrary, we have $\mathbf{coMCSPACE}_{\mathcal{AM}}(f) \subseteq \mathbf{MCSPACE}_{\mathcal{AM}}(f)$, and the reverse inclusion is proved in a completely analogous way. □

It is also true that $\mathbf{coNPMCSPACE}_{\mathcal{AM}} = \mathbf{NPMCSPACE}_{\mathcal{AM}}$, although this result requires more work and will be proved later. The same proof as that of Proposition 22 applies to time complexity classes for P systems.

4.3 Solving problems in PSPACE

In this section we prove that polynomial-space Turing machines can be simulated efficiently (i.e., within the same time and space bounds) by P systems with active membranes [39, 27]. Furthermore, we only need communication rules to implement the simulation. As a consequence, the inclusion $\mathbf{PSPACE} \subseteq \mathbf{PMCSPACE}_{\mathcal{AM}}$ holds (see also [28] for a different approach to the same topic).

4.3.1 Simulating Turing machines

Let M be a deterministic Turing machine working in time $t(n)$ and *polynomial* space $p(n)$. Here we assume that such Turing machine is completely described by a partial transition function

$$\delta \colon Q \times \Gamma \rightharpoonup Q \times \Gamma \times \{\triangleleft, \triangleright\}$$

where Q is the set of states of M, including distinguished states q_0 (the initial state), q_{YES} (the accepting state), and q_{NO} (the rejecting state); $\Gamma = \{0, 1, \sqcup\}$ is the tape alphabet ($\sqcup$ denotes a blank cell), and $\triangleleft$ and $\triangleright$ denote movements to the left and right respectively. We also assume, without loss of generality, that δ is undefined on (and only on) the accepting and rejecting states.

Consider another deterministic Turing machine F_M that, on input 1^n, produces a P system with input $\Pi_{M,n}$ having the following initial configuration:

$$\big[\text{H}\ [\ ]_0^0 [\ ]_1^0 [\ ]_2^0 \cdots [\ ]_{p(n)}^0 \big]_{\text{S}}^0.$$

The outermost membrane S is the input membrane. Each membrane having numerical label $0, \dots, p(n)$ represents a cell tape of M. The symbol contained in that cell in a given configuration of M is represented by the charge of the corresponding membrane: a neutral charge represents a $\sqcup$, a positive charge a 1, and a negative charge a 0. In the following discussion, we denote both the tape symbols of M and the corresponding charges by lower case Greek letters α, β.

The P system $\Pi_{M,n}$ is meant to simulate M on all inputs of length n. The precise input $x \in \{0,1\}^\star$ is encoded as a multiset by another Turing machine E_M simply by subscripting the i-the symbol of x (counting from 0) by its position i. For instance,

$$E_M(11001) = 1_0 1_1 0_2 0_3 1_4.$$

This encoding can be clearly computed in polynomial time with respect to n.

Once the input multiset $E_M(x)$ has been placed into the input membrane S of $F_M(1^n) = \Pi_{M,n}$, the resulting P system $P_{M,x}$ computes according to the following algorithm.

Algorithm 23. Simulating the Turing machine M on input x.

A *Initialise* the charges of membranes $0, \dots, n-1$ according to the input x, thus reconstructing the initial configuration of M.

B *Simulate* the computation of M on x step-by-step until the simulated machine halts.

C *Output* YES if M accepted, and NO if it rejected.

The three phases of Algorithm 23 are detailed below.

Phase A (Initialise) During the first step, the input objects encoding the string x are used in order to initialise the charges of membranes $0, \dots, n-1$. This is accomplished by using the following communication rules:

$$\left.\begin{array}{l} 1_i\ [\ ]_i^0 \to [\#]_i^+ \\ 0_i\ [\ ]_i^0 \to [\#]_i^- \end{array}\right\} \text{for } 0 \le i \le n-1.$$

At the same time, the object H in S is rewritten into H_{0,q_0}. In general, the presence of object $\text{H}_{q,i}$ indicates that the simulated Turing machine is currently in state q, and its tape head is positioned on the i-th cell.

After this one step of Phase A, the initial configuration of M on input x has been recreated in $\Pi_{M,x}$.

Phase B (Simulate) The instruction provided by the transition function δ of M are simulated by using communication rules. Each transition $\delta(q_1, \alpha) = (q_2, \beta, \lhd)$ is implemented by

$$\left.\begin{array}{l} \mathrm{H}_{i,q_1} \; [\;]_i^\alpha \to [\mathrm{H}_{i-1,q_2}]_i^\beta \\ [\mathrm{H}_{i-1,q_2}]_i^\beta \to [\;]_i^\beta \; \mathrm{H}_{i-1,q_2} \end{array}\right\} \text{for } 0 < i \leq p(n).$$

First the head-object is sent into the membrane corresponding to the current tape cell, rewriting its symbol (by changing the charge α to β), shifting to the tape cell on the left, and changing the state to the next one (by modifying its subscripts). Then, in the next step, the head-object is brought back to membrane S; this way we may proceed simulating another step of M.

Simulating a transition such as $\delta(q_1, \alpha) = (q_2, \beta, \rhd)$, where the tape head moves to the right, is completely analogous:

$$\left.\begin{array}{l} \mathrm{H}_{i,q_1} \; [\;]_i^\alpha \to [\mathrm{H}_{i+1,q_2}]_i^\beta \\ [\mathrm{H}_{i+1,q_2}]_i^\beta \to [\;]_i^\beta \; \mathrm{H}_{i+1,q_2} \end{array}\right\} \text{for } 0 \leq i < p(n).$$

We continue simulating M one step at a time (requiring *two* steps on $\Pi_{M,x}$ each) until the Turing machine halts, by entering its accepting or rejecting state. Since we assumed δ to be undefined on these states, no rule described in this phase applies, and we proceed to Phase C.

The total duration of Phase B is at most $2t(n)$ steps.

Phase C (Output) After the simulated machine M has halted, either object $\mathrm{H}_{i,q_{\mathrm{YES}}}$ or $\mathrm{H}_{i,q_{\mathrm{NO}}}$ will appear inside membrane S, for some $i \in \{0, \ldots, p(n)\}$. This object encodes the result of the simulated computation, and can be immediately used by $\Pi_{M,x}$ in order to produce the same result:

$$\left.\begin{array}{l} [\mathrm{H}_{i,q_{\mathrm{YES}}}]_{\mathrm{S}}^0 \to [\;]_{\mathrm{S}}^0 \; \mathrm{YES} \\ [\mathrm{H}_{i,q_{\mathrm{NO}}}]_{\mathrm{S}}^0 \to [\;]_{\mathrm{S}}^0 \; \mathrm{NO} \end{array}\right\} \text{for } 0 \leq i \leq p(n).$$

This concludes the description of the simulation algorithm, allowing us to obtain the following result.

Theorem 24. *A deterministic Turing machine M, running in time $t(n)$ and polynomial space $p(n)$, can be simulated by a uniform family of confluent P systems with active membranes $\mathbf{\Pi}_M = \{\Pi_{M,x} : x \in \Sigma^\star\}$ in time $O(t(n))$ and space $O(p(n))$, and using only communication rules.*

Proof. Each P system $\Pi_{M,x}$ correctly simulates the computation of M on input x and produces the same result. The simulation requires at most $1 + 2t(n) + 1 = O(t(n))$ steps for its three phases; the number of objects required by the simulation is $n + 1$ (n input objects and one head object) and the number of membranes is $p(n) + 2$, for a total of $O(p(n))$ space.

The family $\mathbf{\Pi}_M$ is uniform, since

- The encoding machine E_M runs in polynomial time given x, as it only needs to subscript the string symbols with their positions.
- The family machine F_M needs to output an initial configuration of size $O(p(n))$ and a set of rules of the same size (a constant given by the size of the transition table of M times the number of positions on the tape). These items can be also computed in polynomial time. □

The following corollary is immediate.

Corollary 25. *For all polynomials p we have*

$$\mathbf{SPACE}(p) \subseteq \mathbf{MCSPACE}_{\mathcal{AM}}(p).$$

As a consequence, $\mathbf{PSPACE} \subseteq \mathbf{PMCSPACE}_{\mathcal{AM}}$ *holds.* □

4.4 Simulating P systems via Turing machines

In order to prove the reverse inclusion $\mathbf{PMCSPACE}_{\mathcal{AM}} \subseteq \mathbf{PSPACE}$, we provide a simulation algorithm for P systems running on a Turing machine (actually, on an equivalent *random access machine*). Instead of just simulating confluent P systems, we provide a more general simulation that also works on nonconfluent ones, as this will allow us to prove the stronger result $\mathbf{NPMCSPACE}_{\mathcal{AM}} = \mathbf{PSPACE}$.

4.4.1 Simulation algorithm

For convenience, the actual simulating device we use is a nondeterministic random access machine (or RAM for short, see [23] for definitions), where the operations of addition and subtraction between registers are assumed to require constant time, while multiplication is performed in logarithmic time with respect to the size of the integers involved (i.e., linear time with respect to their length in bits); it is known that such a RAM can be simulated by a Turing machine with only a polynomial increase in time and space complexity [23].

The membrane structure is represented internally as a rooted tree, where each node corresponds to a membrane (the outermost one corresponds to the root) and possesses a list of children membranes and attributes describing the label and current charge of each membrane. The multisets of objects are represented as k-tuples of integers, where k is the size of the alphabet and the i-th entry represents the multiplicity of the i-th object under any fixed ordering of the alphabet. Finally, the set of rules of Π is represented as a list of records [30], each one containing a field for each component of the rule (i.e., the objects, and the labels and charges of the membranes involved in that rule).

The following is a high-level description of the simulation algorithm.

Algorithm 26. Simulating a nonconfluent P system with active membranes $\Pi = (\Gamma, \Lambda, \mu, w_{h_1}, \dots, w_{h_d}, R)$.

A For each rule in R, nondeterministically select the membranes and objects to which it has to be applied during the current simulated step.

B Check whether the rules have been chosen in a maximally parallel way; if this is not the case, abort the simulation by rejecting.

C Apply the rules in the current configuration of Π, starting from the elementary membranes and moving up towards the outermost one.

D If either YES or NO were sent out from the outermost membrane, then halt and accept or reject accordingly; otherwise, simulate the next step by jumping to **A**.

The nondeterministic assignment of rules to objects and membranes in step **A** can be described in details as follows.

A$_1$ Let R' be the set of currently unused rules; set $R' \leftarrow R$.

A$_2$ While $R' \neq \emptyset$ pick a rule $r \in R'$, otherwise go to step **B**.

A$_3$ If $r = [a \to w]_h^\alpha$ then, for each membrane of the form $[\ ]_h^\alpha$, nondeterministically choose an amount k of copies of object a to be rewritten into w; this amount can be anywhere from 0 to the multiplicity of a in that particular membrane. Subtract k from the number of available copies of a in h.

A$_4$ If $r = a\,[\ ]_h^\alpha \to [b]_h^\beta$ then, for each available membrane of the form $[\ ]_h^\alpha$ having an available instance of a in the region immediately outside, nondeterministically choose whether to apply r and, in that case, assign those particular instances of a and h to r making them unavailable.

A$_5$ If $r = [a]_h^\alpha \to [\ ]_h^\beta\, b$ or $r = [a]_h^\alpha \to b$ or $r = [a]_h^\alpha \to [b]_h^\beta\,[c]_h^\gamma$ then, for each available membrane of the form $[\ ]_h^\alpha$ containing an available instance of a, nondeterministically choose whether to apply r and, in that case, assign those particular instances of a and h to r making them unavailable.

A$_6$ If $r = [[\ ]_{h_1}^+ \cdots [\ ]_{h_k}^+ [\ ]_{h_{k+1}}^- \cdots [\ ]_{h_n}^-]_h^\alpha \to [[\ ]_{h_1}^\delta \cdots [\ ]_{h_k}^\delta]_h^\beta [[\ ]_{h_{k+1}}^\varepsilon \cdots [\ ]_{h_n}^\varepsilon]_h^\gamma$ then, for each available membrane of the form $[\ ]_h^\alpha$ containing the available membranes $[\ ]_{h_1}^+, \dots, [\ ]_{h_k}^+, [\ ]_{h_{k+1}}^-, \dots [\ ]_{h_n}^-$ and possibly some neutral available membranes, nondeterministically choose whether to apply r or not; if so, assign all the involved membranes to r making them unavailable.

A$_7$ Set $R' \leftarrow R' - \{r\}$ and go back to step **A**$_2$.

Deciding in step **B** whether the assignment of rules computed in **A** is indeed maximally parallel simply requires us to check, for each membrane and object still available in the current configuration, whether there exists a rule in R that could be applied to them. If this is the case, then the choice of rules is not maximal, and we abort the simulation by rejecting. This does not change the behaviour of the simulating device, since the existence of one accepting computation is not influenced by adding further rejecting ones (in other words, the RAM simulating Π is designed to have an accepting computation iff Π has one).

The actual application of the rules chosen in **A** is performed in step **C** according to the following sub-steps.

$\mathbf{C}_1$ For each $r = [a \to w]_h^\alpha$, remove k instances of a, where k is the multiplicity chosen for r in step $\mathbf{A}_3$, and add k times the objects in w.

$\mathbf{C}_2$ For $r = a\,[\,]_h^\alpha \to [b]_h^\beta$ remove an instance of a from the external region, add an instance of b to the internal one, and change the charge to β.

$\mathbf{C}_3$ For $r = [a]_h^\alpha \to [\,]_h^\beta\, b$ remove an instance of a from the internal region, add an instance of b to the external region, and change the charge to β.

$\mathbf{C}_4$ For $r = [a]_h^\alpha \to b$ move all the objects from the internal region to the external one, replacing an instance of a with b, then, remove the membrane h from the current configuration; the children of h are adopted by its parent.

$\mathbf{C}_5$ For $r = [a]_h^\alpha \to [b]_h^\beta\,[c]_h^\gamma$ duplicate membrane h and its contents, replacing an instance of a by one of b on one side, and by an instance of c on the other; set the charges of the new membranes to β and γ respectively.

$\mathbf{C}_6$ For $r = \big[[\,]_{h_1}^+ \cdots [\,]_{h_k}^+ [\,]_{h_{k+1}}^- \cdots [\,]_{h_n}^-\big]_h^\alpha \to \big[[\,]_{h_1}^\delta \cdots [\,]_{h_k}^\delta\big]_h^\beta \big[[\,]_{h_{k+1}}^\varepsilon \cdots [\,]_{h_n}^\varepsilon\big]_h^\gamma$ create a new instance of h, placing inside it all the negative membranes $h_{k+1}, \ldots, h_n$ from the current membrane, and a *deep copy* of all neutral membranes inside h (i.e., including the substructure having them as the root, including its contents); finally, update the charges according to r.

Finally, in step **D** we decide whether the computation of Π has ended, giving the same result if this is the case, and simulating the next step of Π otherwise.

4.4.2 Analysis of the algorithm

The correctness of this simulation algorithm follows immediately from its description, since it is a straightforward implementation of the usual semantics of P systems with active membranes. The only point worth highlighting

is that the rejecting computations of step **B**, due to having chosen a non-maximal combination of rules, do not influence the overall result of the simulating machine, which accepts if and only if the simulated P system accepts.

Before analysing the time and space required by the simulation algorithm, we prove two auxiliary lemmata.

Lemma 27. *Let Π be a nonconfluent P system with active membranes having an encoding of length m. Then, the number of membranes in any configuration of Π after t computation steps is bounded by $2^{tm+m\log m}$.*

Proof. Since the number of membranes that Π can generate depends on the shape of the initial membrane structure, we begin by choosing a "worst-case membrane structure" [18]. The initial number of membranes of Π is at most m; clearly, each membrane structure of degree m, when viewed as a tree, is a subtree of the complete m-ary tree T_m of depth $m-1$ (i.e., with m levels). Since this tree is *uniform*, that is, the number of children of each node only depends on its depth, it can be represented by a m-tuple of integers $T(k_0, \ldots, k_{m-1})$ where k_i denotes the number of nodes on level i, and k_0 is always 1, since the root is unique. Hence, the initial membrane structure is contained in $T_m = T(1, m, m^2, \ldots, m^{m-2}, m^{m-1})$.

Now suppose that, during each computation step, every possible membrane divides; for the sake of simplicity, also assume that nonelementary division rules cause the duplication of all the children (instead of separating positive and negative ones, and only duplicating the neutral ones). We consider this scenario only for the sake of finding an upper bound, since it cannot really occur in practice.

The result of the application of rules are computed in a bottom-up way: first the elementary membranes divide, and the resulting "intermediate" membrane structure is $T(1, m, m^2, \ldots, m^{m-2}, 2m^{m-1})$. Then, the membranes of depth $m-2$ divide; this also causes another doubling of those having depth $m-1$, giving $T(1, m, m^2, \ldots, 2m^{m-2}, 2^2m^{m-1})$. By repeating this process all the way to the level below the root (which does not divide) we obtain a structure which is a subtree of $T(1, 2m, 2^2m^2, \ldots, 2^{m-2}m^{m-2}, 2^{m-1}m^{m-1})$, that is

$$T\big((2m)^0, (2m)^1, (2m)^2, \ldots, (2m)^{m-2}, (2m)^{m-1}\big)$$

Notice how this tree can be obtained by replacing m with $2m$ in the initial tree T_m: analogously, after another computation step we obtain the tree T_{2^2m} and, in general, after t steps we obtain the tree T_{2^tm}. Since this tree has m levels, the number of nodes is bounded by m times the number of leaves, i.e.,

$$(2^tm)^{m-1} \cdot m \le (2^tm)^{m-1} \cdot 2^tm = (2^tm)^m = 2^{tm+m\log m}.$$

□

We also prove an upper bound on the number of objects.

Lemma 28. *Let* Π *be a nonconfluent P system with active membranes having an encoding of length* m*. Then, the number of objects in any configuration of* Π *after* t *computation steps is bounded by* $2^{O(t^2 m \log m)}$.

Proof. The initial number of objects of Π, and in particular the initial number contained inside each membrane, is bounded by the length m of the whole encoding. The only way to increase this number, besides using membrane division, is to use evolution rules. The right-hand side of each evolution rule $[a \to w]_h^\alpha$ contains at most m objects (once again, because of the length of the encoding), hence the number of objects after step $i + 1$ is at most m times the amount of step i. If no communication or dissolution rules are ever used, after t steps each membrane contains at most $m^{t+1} = 2^{(t+1)\log m}$ objects, for a total of $2^{tm+m\log m} \cdot 2^{(t+1)\log m} = 2^{O(t^2 m \log m)}$ objects in the whole configuration (using Lemma 27). This upper bound also holds when communication or dissolution rules are used, because these kinds of rules only move the objects around without increasing their number. □

We are now able to prove that the simulation is at most exponentially slower.

Theorem 29. *Let* Π *be a nonconfluent P system with active membranes, running in time* T *and having an encoding of length* m*. Then, the simulation algorithm computes the same result as* Π *in time* $2^{O(T^2 m \log m)}$.

Proof. We analyse the complexity of each step of the simulation algorithm, applied during the simulation of step t of Π, beginning with the sub-steps of **A**.

- In step $\mathbf{A}_1$ we copy the set of rules in time $O(m)$.
- Step $\mathbf{A}_2$ consists of checking if R' is empty and going to step **B**, which can be done in constant time; a nondeterministic choice of a rule involves scanning the set R', which requires $O(m)$ time.
- In $\mathbf{A}_3$ we traverse the whole membrane structure to find the membranes of the form $[\,]_h^\alpha$; for each of those (assume they all have this form for the sake of argument) we choose a number of objects to be rewritten: this requires a linear number of steps with respect to the number of bits used to store the multiplicities of the objects, that is $O(t^2 m \log m)$ by Lemma 28. Since by Lemma 27 the number of membranes is at most $2^{tm+m\log m}$, this step requires a total time bounded by $2^{tm+m\log m} \cdot O(t^2 m \log m)$, which is $2^{O(tm+m\log m)}$.
- Steps $\mathbf{A}_4$ and $\mathbf{A}_5$ also require traversing the membrane structure, but checking whether a rule is applicable and choosing whether to actually

apply it only requires constant time. Hence these steps require time proportional to the number of membranes, which is also bounded by $2^{O(tm+m\log m)}$.

- While traversing the membrane structure in $\mathbf{A}_6$, we need to inspect all the children of the current membrane in order to establish whether the nonelementary division rule can be applied. The number of children is bounded by $2^{tm+m\log m}$; thus, this step also requires $2^{O(tm+m\log m)}$ time.

- Finally, removing the selected rule from R' and going back to $\mathbf{A}_2$ in step $\mathbf{A}_7$ requires at most $O(m)$ time.

The loop consisting of steps $\mathbf{A}_2$–$\mathbf{A}_7$ is executed once for each of the $O(m)$ rules; hence, the total time required for executing step $\mathbf{A}$ (when simulating step t of Π) is $O(m) \cdot 2^{O(tm+m\log m)}$, which is still $2^{O(tm+m\log m)}$.

The analysis for step $\mathbf{B}$ is very similar; the only difference is that we need to iterate through the different kinds of object in order to establish whether an evolution rule could be applied. However, this only adds a smaller term to the exponent, and the time remains $2^{O(tm+m\log m)}$.

The cost of the sub-steps of step $\mathbf{C}$ are computed as follows.

- In $\mathbf{C}_1$ we perform a number of multiplications bounded by the size $O(m)$ of the alphabet of Π, and the size of the operands is bounded by $O(t^2 m\log m)$; the total time is thus $O(t^2 m^2 \log m)$.

- Steps $\mathbf{C}_2$ and $\mathbf{C}_3$ only require constant time.

- In $\mathbf{C}_4$ we need to move all the objects of the dissolving membrane to its parent: this requires $O(m)$ constant-time additions. We also need to move all its children, which are at most $2^{O(tm+m\log m)}$.

- Step $\mathbf{C}_5$ requires us to create a new membrane and duplicate the contents of the dividing one; duplicating the multiset requires $O(m)$ time.

- Finally, in $\mathbf{C}_6$ we duplicate a whole substructure of membranes with its contents. By Lemmata 27 and 28, the size of this structure is bounded by $2^{tm+m\log m} \cdot 2^{O(t^2 m\log m)}$, which is $2^{O(t^2 m\log m)}$.

One step among $\mathbf{C}_1$–$\mathbf{C}_6$ has to be executed for each membrane and each rule; since the most expensive one is $\mathbf{C}_6$, we can bound the total time required by step $\mathbf{C}$ by

$$O(m) \cdot 2^{tm+m\log m} \cdot 2^{O(t^2 m\log m)} \cdot 2^{O(t^2 m\log m)} = 2^{O(t^2 m\log m)}$$

The last step of the simulation algorithm is $\mathbf{D}$, and it requires only constant time.

This proves that executing steps **A**–**D** in order to simulate step t of Π requires $2^{O(t^2 m \log m)}$ time, due to step **C**. Simulating all T steps of Π requires summing this amount for $1 \leq t \leq T$; this sum is clearly bounded by T times the cost of the last step, that is, by $2^{O(T^2 m \log m)}$ time. □

Nevertheless, the space required by the simulation is only polynomially larger.

Theorem 30. *Let Π be a nonconfluent P system with active membranes, running in space S and having an encoding of length m. Then, the simulation algorithm computes the same result as Π using space $O(S \log m)$.*

Proof. Explicitly storing the current configuration of Π as described above requires at most the same space as Π (and much less in some cases, since we store the multiplicity of objects in binary instead of unary). However, our simulation also requires storing the labels of the membranes, which do not occupy space in Π. Since Π initially contains at most m membranes, $\log m$ bits are sufficient to represent the labels. Hence, the configuration of Π can be stored in space $O(S \log m)$.

The auxiliary data structures needed by the simulation algorithm do not exceed this amount of space: the largest one is a stack required to perform a depth-first search of the membrane structure, and the number of items on the stack is bounded by the depth of the structure. □

4.4.3 Complexity-theoretic implications

The analysis of Algorithm 26 allows us to prove that polynomial space has the same power for P systems and Turing machines, even when nonconfluence is taken into account.

Theorem 31. $\mathbf{PMCSPACE}_{\mathcal{AM}} = \mathbf{NPMCSPACE}_{\mathcal{AM}} = \mathbf{PSPACE}$.

Proof. The inclusion $\mathbf{PSPACE} \subseteq \mathbf{PMCSPACE}_{\mathcal{AM}}$ is proved in Section 4.3. The inclusion $\mathbf{NPMCSPACE}_{\mathcal{AM}} \subseteq \mathbf{PSPACE}$ is proved by first performing the uniform polynomial-time construction $x \mapsto \Pi_x$, then using the simulation algorithm on Π_x: this requires polynomial nondeterministic space by Theorem 30, which can be reduced to polynomial deterministic space by using Savitch's theorem [23]. □

Clearly, this result also proves that $\mathbf{NPMCSPACE}_{\mathcal{AM}}$ is also closed under complement, as stated previously. Theorem 30 also has another corollary, this time related to exponential space complexity classes.

Corollary 32. *Both* $\mathbf{EXPMCSPACE}_{\mathcal{AM}}$ *and* $\mathbf{NEXPMCSPACE}_{\mathcal{AM}}$ *are subsets of* $\mathbf{EXPSPACE}$.

In other words, exponential-space P systems with active membranes are no more powerful than exponential-space Turing machines [30]. Whether the reverse inclusions also hold is an open problem; we cannot extend the simulation of Turing machines of Section 4.3.1 to exponential-space machines, as our uniformity condition does not allow us to construct the required membranes, which are exponential in number.

As an aside, notice that Theorem 29 implies that $\mathbf{PMC}_{\mathcal{AM}} \subseteq \mathbf{NPMC}_{\mathcal{AM}} \subseteq \mathbf{EXP}$ holds: P systems with active membranes are at most exponentially faster than Turing machines.

Chapter 5

Counting by trading space for time

As we showed in Chapter 3, P systems with active membranes using division rules are able to solve problems that are conjectured to be intractable by classic computing devices. The speed-up is achieved at the expense of space, which becomes exponential. In this chapter we prove that polynomial-time P systems where division only applies to elementary membranes can solve *counting problems*, which are possibly even harder than **NP**-complete ones.

5.1 On the nature of membrane division

The most important feature of P systems with active membranes, at least from a complexity-theoretic standpoint, is the ability to generate exponentially many membranes in polynomial time. When no division rules are allowed, indeed, they cannot solve problems outside of **P** in polynomial time.

Theorem 33 (Milano Theorem [42]). *Let $\mathcal{D}$ denote the class of P systems with active membranes using no elementary or nonelementary division rules. Then* $\mathbf{PMC}_{\mathcal{D}} = \mathbf{P}$. □

Notice that P systems without division rules can still generate an exponentially large workspace, by using object evolution rules with more than one object on the right-hand side. What the Milano theorem essentially says is that *some* form of membrane division is required to go beyond **P**.

In the original paper where P systems with active membranes were introduced [32], the decision problem SAT was solved semi-uniformly in polynomial time by using both elementary and nonelementary division rules. This result was then [42] showed to hold even if only elementary division rules are allowed, and further improved to provide a uniform solution [24] (this is the result we presented as an example in Chapter 3).

It was also shown that P systems using nonelementary division can solve problems harder than **NP**: the **PSPACE**-complete ones, like *Quantified* SAT (QSAT) [4, 4]. The reason that makes nonelementary division apparently stronger than elementary division is related to the shape of the membrane structure that can be crated during the computation. Nonelementary division can create membrane structures shaped like full elementary binary trees having exponentially many nodes; this allows us to carry out what essentially amounts to a *quantifier elimination algorithm* for quantified Boolean formulae, based on the following identities:

$$\forall x \, \varphi(x, \vec{y}) = \varphi(0, \vec{y}) \wedge \varphi(1, \vec{y})$$
$$\exists x \, \varphi(x, \vec{y}) = \varphi(0, \vec{y}) \vee \varphi(1, \vec{y}).$$

Essentially, nonelementary division can be used to transform a membrane structure having the shape of the parse tree *quantified* Boolean formula to a membrane structure shaped like the parse tree of the *expanded* formula. The depth of the membrane structure, in this case, is directly proportional to the number of quantifiers in the original formula. This procedure cannot be carried out by using elementary division only, as this kind of rule can only create exponentially many *leaf* membranes.

Later, an upper bound to the computing power of polynomial-time P systems using division rules was provided by Sósik and Rodríguez-Patón [35]: they can always be simulated in polynomial *space* by deterministic Turing machines.

Theorem 34. $\mathbf{PMC}_{\mathcal{AM}} \subseteq \mathbf{PSPACE}$. □

This provides a characterisation of **PSPACE** in terms of P systems with active membranes using nonelementary division. On the other hand, the computing power of the variant using only elementary division has not been characterised yet.

5.2 Counting problems

Each positive instance x of an **NP** problem L possesses a *short certificate* proving the membership of x in L [23]. This certificate is a string y of polynomial length with respect to $|x|$ such that, given x and y, it can be checked in deterministic polynomial time that $x \in L$. For instance, a short certificate for the membership of a formula φ in SAT is simply a truth assignment satisfying φ.

Nondeterministic polynomial-time Turing machines can both *guess* the short certificate *and* check it in polynomial time. If nondeterminism is not available, we usually need to check a large amount of potential certificates (the total number is exponential for **NP**-complete problems), requiring superpolynomial time (unless $\mathbf{P} = \mathbf{NP}$).

A generalisation of **NP** problems involves not only deciding whether a short certificate exists, but *how many* candidates are actually certificates. A question of this form is "How many satisfying assignment does the Boolean formula φ admit?". Here we deal with a decision version of these problems, asking whether the *majority* of the candidates are membership certificates.

Definition 35. The complexity class **PP** [12] consists of all the languages $L \subseteq \Sigma^\star$ that can be decided by nondeterministic Turing machines N with the following acceptance criterion: N accepts $x \in \Sigma^\star$ if and only if more than half of the computations of N on input x are accepting.

The name **PP** stands for "Probabilistic **P**", as an equivalent definition can be given in terms of probabilistic Turing machines [12].

Alhazov et al. proved in 2009 [3] that P systems with elementary active membranes (and no dissolution rules) can be used to solve **PP**-complete problems; however, their result is not directly related to the complexity class $\mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$, as their solution requires either context-sensitive object evolution rules (a powerful feature that is not part of the standard definition) or post-processing an exponential-size multiset of objects.

In this chapter we show that $\mathbf{PP} \subseteq \mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$ actually holds [29, 26]. We prove this result by solving the following decision problem.

Problem 36 (Threshold-3SAT)**.** Given a Boolean formula φ over m variables and a non-negative integer $k < 2^m$, do more than k assignments (out of 2^m) satisfy it?

Proposition 37. Threshold-3SAT *is* **PP***-hard.*

Proof. We reduce the following standard **PP**-complete problem [23] to Threshold-3SAT.

> **Problem 38** (Majority-SAT)**.** Given a Boolean formula φ in CNF, having c clauses over m variables and such that each variable occurs at most once per clause, do more than half the assignments (i.e., more than 2^{m-1} assignments) satisfy it?

The reduction is similar to that from SAT to 3SAT described in [11]. We first transform φ into a formula having *at most* three literals per clause. Observe that φ is satisfied iff the formula obtained by replacing a clause of $p > 3$ literals $\bigvee_{i=1}^{p} \ell_i$ with

$$(y \Leftrightarrow \ell_1 \vee \ell_2) \wedge \left(y \vee \bigvee_{i=3}^{p} \ell_i \right)$$

is also satisfied, assuming y is a new variable. In CNF, that is equivalent to

$$(\neg \ell_1 \vee y) \wedge (\neg \ell_2 \vee y) \wedge (\ell_1 \vee \ell_2 \vee \neg y) \wedge \left(y \vee \bigvee_{i=3}^{p} \ell_i \right).$$

This substitution doubles the number of total assignments of the formula, due to the addition of a new variable, but the number of *satisfying* ones is left unchanged, as the value of y is forced to be equal to $\ell_1 \vee \ell_2$. The substitution decreases by one the number of literals of the initial clause; by repeating the process $p-3$ times, and then again to any other clause having more than three literals, we obtain a formula φ' having at most three literals per clause, and the same number of satisfying assignments as φ. The number of variables of φ' is bounded by $m + cm$.

Next, we transform every clause of one or two literals into a clause of exactly three. A clause of a single literal ℓ is replaced by

$$(\ell \vee z_1 \vee z_2) \wedge (\ell \vee \neg z_1 \vee z_2) \wedge (\ell \vee z_1 \vee \neg z_2) \wedge (\ell \vee \neg z_1 \vee \neg z_2),$$

where z_1 and z_2 are new variables, which is clearly satisfied iff ℓ is. Each replacement like this one multiplies by $2^2 = 4$ the number of satisfying assignments of the whole formula, as the values of z_1 and z_2 are actually irrelevant.

A clause of two literals $\ell_1 \vee \ell_2$ is replaced by

$$(\ell_1 \vee \ell_2 \vee z) \wedge (\ell_1 \vee \ell_2 \vee \neg z),$$

where z is a new variable, which is also equivalent to the original clause but doubles the number of satisfying assignments of the formula.

Call φ'' the formula obtained from φ' by replacing single and 2-literal clauses by conjunctions of 3-literal clauses as described above, and let q be the number of variables added in the process (notice that q is $O(cm)$). Then it should be clear that φ has more than 2^{m-1} satisfying assignments iff φ' does, and the latter is equivalent to φ'' having more than 2^{m+q-1} satisfying assignment.

Since the mapping $R(\varphi) = (\varphi'', 2^{m+q-1})$ is computable in polynomial time with respect to c and m, it is a reduction from MAJORITY-SAT to THRESHOLD-3SAT. $\square$

5.3 Solving THRESHOLD-3SAT

A solution to THRESHOLD-3SAT can be designed by considering our solution to 3SAT given by Algorithm 14. Notice that, at the beginning of Phase C, the number of instances of s_n that are sent out to the outermost membrane is exactly the number of satisfying assignments admitted by the input formula φ. A simple change of the structure of Algorithm 14 provides a way of checking whether that amount is larger than k.

Algorithm 39. Solving 3SAT on input φ, a 3CNF Boolean formula of m variables.

A *Initialise* the contents of the membranes.

B *Generate* 2^m membranes using elementary division, each one containing objects representing a different truth assignment to the variables of φ.

C *Evaluate* φ under the 2^m assignments, in parallel, and send out from each membrane an object s_n whenever the assignment contained in that membrane satisfies φ.

D *Delete* k instances of s_n (or all of them, if less than k exist).

E *Output* YES if at least one instance of s_n remains, and NO otherwise.

Here we add an initialisation step for technical reasons. The real deviation from the standard membrane computing algorithm for **NP**-complete problems is given by Phase D, first proposed by Alhazov et al. [3] but not implemented by them using standard P systems with active membranes. We will use elementary division and communication rules to implement it.

We also need a new encoding of the input instances, as we are not only given a Boolean formula φ as input but also the threshold k. The Boolean formula itself is encoded exactly as in Section . Also notice that the integer k, being in the range $[0, 2^m)$, can be encoded using exactly m binary digits. Let us denote by $\ulcorner k \urcorner$ the binary encoding of k. Then, the whole THRESHOLD-3SAT instance is encoded as the juxtaposition of the encodings of its parts:

$$\ulcorner \varphi, k \urcorner = \ulcorner \varphi \urcorner \ulcorner k \urcorner.$$

The length of this binary string for a formula of m variables is exactly $8\binom{m}{3} + m$, or $n + m$ by letting $n = 8\binom{m}{3}$. As in Section , we can recover m from $n + m$ and detect malformed input instances by finding the unique positive integer root of the polynomial $p(m) = 8\binom{m}{3} + m$.

Each input size $n+m$ (in unary notation) is mapped by a "family" machine F to a P system Π_{n+m}, having the following initial configuration:

$$\mathcal{C}_0 = \big[[\mathrm{I}_n]^0_{\mathrm{E}}\ [\ ]^0_{\mathrm{K}_0} \cdots [\ ]^0_{\mathrm{K}_{m-1}} \mathrm{O}_t\ \mathrm{NO}_{t+2}\big]^0_{\mathrm{S}}$$

that is, the same initial configuration of the P systems of Algorithm 14 augmented by m membranes representing the bits of k. We also have $t = 5n + 4$; this is due to a different duration of the computation.

The input provided to Π_{n+m} is computed by another polynomial time Turing machine E (the "encoding" machine) that, given an m-variable 3CNF formula as described in the previous section and an integer k, outputs the following set of objects:

$$\begin{aligned} E(\ulcorner \varphi, k \urcorner) = \{&\mathrm{C}_i : \text{the } i\text{-th clause does } \textit{not} \text{ appear in } \varphi, \text{ for } 1 \le i \le 8\tbinom{m}{3}\} \cup \\ &\{\mathrm{K}_i : \text{the } i\text{-th bit of } k \text{ (counting from 0) is 1, for } 1 \le i \le m-1\}. \end{aligned}$$

After the input multiset is placed inside the outermost membrane S of Π_{n+m}, the computation proceeds as described below.

Phase A (Initialise) In the first computation step, the object C_i, corresponding to the clause that do not appear in the input formula φ, are moved to membrane E by using the communication rules

$$\text{C}_i\ [\ ]_e^0 \rightarrow [\text{C}_i]_e^0 \qquad \text{for } 1 \leq i \leq n.$$

This takes a number of step at most equal to n (if φ contains no clauses). In the mean time, the object I_n has its subscript decremented by one for $n-1$ computation steps, and it is finally replaced during the n-th step, according to the rules

$$\begin{aligned}&[\text{I}_i \rightarrow \text{I}_{i-1}]_{\text{E}}^0 && \text{for } 1 \leq i \leq n\\ &[\text{I}_1 \rightarrow \text{X}_1 \cdots \text{X}_m \text{W}_m]_{\text{E}}^0.\end{aligned}$$

Hence, after n computation steps, membrane E contains C_i for each missing clause i, and the variable-objects $\text{X}_1, \ldots, \text{X}_m$.

At the same time, the objects K_i are first moved to their respective membranes in the first time step, making them positively charged

$$\text{K}_i\ [\ ]_{\text{K}_i}^0 \rightarrow [\text{K}_i]_{\text{K}_i}^+ \qquad \text{for } 0 \leq i < m$$

then each K_i divides its membrane i times:

$$[\text{K}_i]_{\text{K}_j}^+ \rightarrow [\text{K}_{i-1}]_{\text{K}_j}^+\ [\text{K}_{i-1}]_{\text{K}_j}^+ \qquad \text{for } 0 \leq j < m \text{ and } 1 \leq i \leq j.$$

After at most m steps (the largest possible subscript is $m-1$), there are exactly k positively charged membranes among those labelled by $\text{K}_0, \ldots, \text{K}_{m-1}$.

The total duration of Phase A is n steps.

Phase B (Generate) This phase works exactly as Phase A of Algorithm 14, generating all possible truth assignments inside copies of membrane E. The duration is also m computation steps.

Phase C (Evaluate) This phase is also identical to Algorithm 14 (Phase B), where each copy of membrane E evaluates φ under a different truth assignment. Its total duration is also $3n+2$ steps.

Phase D (Delete) At the beginning of the this phase, the objects S_n are sent out (without changing the charge of the membrane E they are located in) by using the rule

$$[\text{S}_n]_{\text{E}}^+ \rightarrow [\ ]_{\text{E}}^+\ \text{S}_n.$$

In the next step, k copies of S_n (or all of them, if less than k exist) are "deleted" from membrane S by sending them into any of the membranes

having label $\text{K}_0, \ldots, \text{K}_{m-1}$ *and positive charge*. The membranes are set to negative in the process, to avoid absorbing multiple objects:

$$\text{s}_n\ [\ ]^+_{\text{K}_i} \to [\#]^-_{\text{K}_i} \qquad\qquad \text{for } 0 \le i < m.$$

Recall that the number of positively charged membranes K_i is exactly k. Hence, after a single computation step, one or more copies of s_n remain in membrane s if and only if the number of satisfying assignments of φ was greater than k.

The duration of this phase is 2 computation steps.

Phase E (output) The output phase is the same as Phase C of Algorithm 14, except that the objects s_n have already been sent out, and that the initial value of t is different, as in this case we have $t = 5n + 4$.

5.4 Solving the PP problems

The existence of Algorithm 39 proves that elementary division suffices to solve one particular **PP**-complete problem.

Theorem 40. *The problem* THRESHOLD-3SAT *can be solved in polynomial time by a uniform family of confluent P systems with active membranes (without using dissolution or nonelementary division rules). In symbols,* 3SAT $\in \mathbf{PMC}_{\mathcal{AM}(-\text{d},-\text{n})}$. □

Being able to solve one **PP**-complete problem implies $\mathbf{PP} \subseteq \mathbf{PMC}_{\mathcal{AM}(-\text{d},-\text{n})}$ if the uniformity condition is defined as in [24], as closure under polynomial-time reductions is immediate. However, our uniformity condition is possibly weaker, as the P system associated with each input strictly depends on its size (as length of the string encoding it), and the class $\mathbf{PMC}_{\mathcal{AM}(-\text{d},-\text{n})}$ defined this way is currently not known to be closed under polynomial-time reductions. The main problem is that arbitrary polynomial-time reductions do not necessarily map strings of the same length into strings of the same length.

Hence, to prove the **PP** inclusion [26] we operate as follows.

Theorem 41. $\mathbf{PP} \subseteq \mathbf{PMC}_{\mathcal{AM}(-\text{d},-\text{n})}$.

Proof. Let $L \in \mathbf{PP}$, and let R be a Turing machine reducing L to the problem THRESHOLD-3SAT in polynomial time $p(n)$, where n is the length of the instance of L. Also let F and E be the Turing machines providing the uniformity condition for the family of P systems $\mathbf{\Pi}$ solving THRESHOLD-3SAT designed in the previous section; the P system deciding the instances of size $n + m$ is denoted by Π_{m+n}. We describe two polynomial-time Turing machines F' and E' constructing a family of P systems $\mathbf{\Pi}'$, also running in polynomial time, such that $L(\mathbf{\Pi}') = L$.

The machine F', on input 1^n (where $n = |x|$), constructs a P system able to solve the largest THRESHOLD-3SAT instance (φ, k) that might be produced as the output of R; if the actual output of R is smaller than that, we can pad it to the correct length by adding enough zeroes (this is a legitimate possibility, as described below). Let f be defined as follows:

$$f(n) = \min\{n' + m' : n' + m' \geq n \text{ and } n' = 8\tbinom{m'}{3}\}$$

that is, $f(n)$ is the smallest integer of the form $8\binom{m'}{3} + m'$ greater than or equal to n. Then, F' behaves as follows:

$$F'(1^n) = F\big(1^{f(p(n))}\big) = \Pi_{f(p(n))}.$$

Since R runs in time $p(n)$, the P system $\Pi_{f(p(n))}$ is large enough to receive as input any formula φ obtained via the reduction R, as $|R(x)| = |(\varphi, k)| \leq p(|x|)$, as long as it is padded to length $f(p(n))$ as described above.

Notice that the value $f(n)$ can be obtained in polynomial time with respect to n by simply computing $8\binom{m'}{3} + m'$ for all integers m' until n is reached or exceeded; furthermore, $f(n)$ itself is at most polynomial in n (e.g., a trivial upper bound is $8\binom{n}{3} + n$).

The encoding machine E', on input x, produces an output formula encoding φ', obtained from $(\varphi, k) = R(x)$ as follows:

$$\varphi' = \varphi 0^\ell \qquad\qquad \text{where } \ell = f(p(n)) - |\varphi|.$$

Observe that the enumeration algorithm of Figure 3.1 has the following property: the sequence of clauses over m variables is a prefix of the sequence of clauses over $m' \geq m$ variables. As a consequence, each formula over m variables can also be considered as a formula over m' variables (where the only clauses that actually appear only involve the original variables) by padding its encoding to the correct length $8\binom{m'}{3}$ with zeroes. Here m' satisfies $f(p(n)) = 8\binom{m'}{3} + m'$. The number of required assignments k has to be adjusted accordingly: every assignment of the original formula φ corresponds to $2^{m'-m}$ assignments of φ' (obtained by extending it with arbitrary values to the new variables) that satisfy it iff the original assignment satisfies φ, since the new $m' - m$ variables do not actually appear in φ'. Hence, we define $k' = 2^{m'-m} \times k$.

Summarising, the machine E' behaves as follows:

$$E'(x) = E\big(\varphi 0^\ell, 2^{m'-m} k\big) \qquad\qquad \text{where } (\varphi, k) = R(x).$$

Since $(\varphi', k') \in$ THRESHOLD-3SAT iff $(\varphi, k) \in$ THRESHOLD-3SAT by construction, and the latter is equivalent to $x \in L$ by reduction, we obtain $L \in \mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$. But L was an arbitrary **PP** language: hence the inclusion $\mathbf{PP} \subseteq \mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$ holds as required. □

Chapter 6

P systems as oracles

Having proved that P systems with elementary active membranes can solve the decision version of counting problems in polynomial time, we are interested in finding improvements to that result. In this chapter we describe a technique whereby an existing family of P systems solving a problem L can be embedded into another family of P systems, that simulates a Turing machine, as a "subroutine". The procedure is, in principle, quite general, although the technical details may vary (in particular, depending on the encoding of the instances of the problem we consider). We shall then exploit this technique in order to prove that certain classes of Turing machines with oracles can be simulated in polynomial time by P system [27]. This will provide us with a better lower bound on the complexity class $\mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$.

6.1 Oracle Turing machines

An oracle for a language $L \subseteq \Sigma^\star$ is a "device" of unspecified nature that can answer queries of the form "Is $y \in L$?" for all strings $y \in \Sigma^\star$, and provides the correct answer *instantaneously*. A Turing machine may be equipped with an oracle for L in order to increase its computing power by deciding membership in L "for free". Oracle Turing machines are not meant to model realistic computations, but are useful to characterise the hardness of problems in a finer way: for instance, the *polynomial hierarchy* (described below) classifies some of the problems between **P** and **PSPACE** with respect to the kind of oracle needed in order to solve them in polynomial time.

Oracle Turing machines normally [23] use an extra tape to write the query string y, then enter a distinguished "query state" and, during the next computation step, their state is automatically changed to either q_{Y} or q_{N} (distinct from the accepting and rejecting states q_{YES} and q_{NO}) depending on whether the query string belongs to the oracle language L or not. When simulating an oracle machine (Section 6.2) we will use a slightly different definition (which is equivalent with respect to the results we present). We

will also fix our query language to THRESHOLD-3SAT, as the details may differ for other problems.

Let us recall a few definitions related to oracle machines. [23]

Definition 42. Let $L \subseteq \Sigma^\star$ be a language. The complexity class $\mathbf{P}^L$ consists of all problems solvable by deterministic polynomial-time Turing machines with access to an oracle for L.

Similarly, $\mathbf{NP}^L$ is the class of problems solvable by polynomial-time nondeterministic Turing machines with an oracle for L, and $\mathbf{coNP}^L$ the class of problems solvable by co-nondeterministic Turing machines (i.e., machines that reject iff a rejecting computation exists) in polynomial time.

Definition 43. Let $\mathbf{C}$ be any complexity class. Then

$$\mathbf{P}^{\mathbf{C}} = \bigcup_{L \in \mathbf{C}} \mathbf{P}^L \qquad \mathbf{NP}^{\mathbf{C}} = \bigcup_{L \in \mathbf{C}} \mathbf{NP}^L \qquad \mathbf{coNP}^{\mathbf{C}} = \bigcup_{L \in \mathbf{C}} \mathbf{coNP}^L.$$

The polynomial hierarchy is defined as follows [23, 36].

Definition 44 (The polynomial hierarchy)**.** Let $\Delta_0\mathbf{P} = \Sigma_0\mathbf{P} = \Pi_0\mathbf{P} = \mathbf{P}$, and for all $i \in \mathbb{N}$

$$\Delta_{i+1}\mathbf{P} = \mathbf{P}^{\Sigma_i\mathbf{P}} \qquad \Sigma_{i+1}\mathbf{P} = \mathbf{NP}^{\Sigma_i\mathbf{P}} \qquad \Pi_{i+1}\mathbf{P} = \mathbf{NP}^{\Sigma_i\mathbf{P}}.$$

The *cumulative* polynomial hierarchy defined as $\mathbf{PH} = \bigcup_{i \in \mathbb{N}} \Sigma_i\mathbf{P}$.

The cumulative hierarchy **PH** thus contains some very hard problems. It is contained in **PSPACE**, and it is conjectured to be *strictly* contained in it, but it is, in a way, very close to it. Complete problems for each level [36] are given by the validity problems for quantified Boolean formulae having a number of quantifier alternations bounded by a constant.

However, an even larger class can be identified by choosing oracles for counting problems.

Theorem 45 (Toda's theorem [37])**.** $\mathbf{PH} \subseteq \mathbf{P}^{\mathbf{PP}}$. □

In other words, we can solve arbitrarily hard problems in the polynomial hierarchy with an oracle for a counting problem.

6.2 Simulating an oracle machine

First of all, we assume that the Turing machine with oracle M has a *single tape*, which is used as input space, scratch space, *and* as a place to write the oracle queries. The oracle querying procedure works as follows: first M writes down the query string y on the tape, then it moves the head to the first symbol of y, and finally enters the query state $q_?$. In the next computation step, the state of M will be changed, in order to reflect the answer of the

oracle to the question "is $y \in L$?", to $q_{\rm Y}$ or $q_{\rm N}$, and the tape head will be repositioned to the first tape cell. A further assumption we are allowed to make is that *all query strings have the same length* $\ell = 8\binom{m}{3} + m$, where ℓ is the largest integer of that form to be less than or equal to $p(n)$; we are allowed to do this because, as we described in Chapter 5, a pair (φ_1, k_1), where φ_1 is a formula over m_1 variables, is a positive instance of THRESHOLD-3SAT if and only if (φ_2, k_2) is, where φ_2 is a formula having the same clauses of φ_1 but over $m_2 \geq m_1$ variables.

Let $\Pi_\ell \in \mathbf{\Pi}$ be the P system associated to the THRESHOLD-3SAT instances of length $\ell = 8\binom{m}{3} + m$ (see Section 5.3). We construct the P system $\Pi_{M,n}$ simulating M on inputs of length n (a variation of the simulation of Section 4.3.1) as follows:

$$\big[_{\rm H}\,[\;]_0^0\,[\;]_1^0\,[\;]_2^0 \cdots [\;]_{2p(n)}^0 [\Pi_\ell]_{\rm Q}^0 [\Pi_\ell]_{\rm Q}^0 \ldots [\Pi_\ell]_{\rm Q}^0 [\;]_{\rm A}^0 \big]_{\rm S}^0.$$

This initial configuration contains $p(n)$ copies of Π_ℓ, each one enclosed by a further membrane having label Q. The number $p(n)$ is chosen as it is the maximum number of queries that M can perform[1]. However, the initial configuration of the embedded P systems Π_ℓ is changed by erasing the initial objects and keeping only the membrane structure and the rules; this is required in order for these P systems to avoid starting their computation immediately, as their input will be provided later during the computation of $\Pi_{M,n}$. Membrane A will be used to store (in its charge) the result of an oracle query.

Notice how the number of simulated tape cells has been increased to $2p(n) + 1$: this is due to the fact that we require all query strings to be of length ℓ, hence we need to leave extra space on the tape for padding them to this length.

The simulation of M by $\Pi_{M,n}$ works exactly as in Section 4.3.1 as long as M does not enter its query state. We shall describe how oracle queries are simulated, first in an informal way, then by giving all the technical details.

6.2.1 Informal description of the simulation

This is an overview of the oracle query simulation:

1. The tape positions corresponding to the query string are inspected, and the multiset of objects w encoding it is produced.

2. At the same time, one of unused copies of the embedded P systems Π_ℓ is chosen; it will be used to simulate the current query.

[1] Although, for the sake of a simpler exposition, this configuration contains several membranes having the same label, the labels can be made unique by subscripting them and replicating the rules accordingly (this does not affect the construction time by more than a polynomial amount).

3. The objects in w are moved to their initial position inside that copy of Π_ℓ.

4. The objects missing from the initial configuration of Π_ℓ are created and moved to the correct membranes.

5. Now the embedded P system performs its computation as in Section 5.3, and produces the answer to the query.

6. Finally, the answer is communicated to the object simulating the tape head of M; it switches to the corresponding state and resumes simulating the Turing machine.

In more details, the procedure works as follows. When M enters the query state $q_?$, the object $\text{H}_{i,q_?}$ is produced and moved to membrane S. According to the convention described above, the query string y (necessarily of length ℓ) is now located on tape cells $i, \ldots, i + \ell - 1$.

First, the object $\text{H}_{i,q_?}$ is rewritten into the following multiset:

$$\text{H}'_{i,q_?} \; \text{C}'_{1,i} \; \text{C}'_{2,i+1} \; \cdots \; \text{C}'_{\ell-m,i+\ell-m-1} \; \text{K}'_{0,i+\ell-m} \; \text{K}'_{1,i+\ell-m+1} \; \cdots \; \text{K}'_{m-1,i+\ell-1}$$

The objects $\text{C}_{j,i}$ and $\text{K}_{j,i}$ represent *potential* input objects C_j and K_j for an instance of Π_ℓ, which will be actually produced depending on the particular query string y. The procedure is the following one: each object $\text{C}_{j,i}$ and $\text{K}_{j,i}$ enters the corresponding membrane j, and is either rewritten into a "junk" object $\#$, or sent out as C_j or K_j depending on the symbol contained in the tape cell (thus simulating the encoding machine E_L described above).

In the mean time, the object $\text{H}'_{i,q_?}$ nondeterministically selects one of the neutral membranes having label Q and "opens" it by setting its charge to positive. The P system Π_ℓ contained inside that membrane will be used to answer the current query. The object is moved back as W_ℓ to membrane S, where it "waits" for ℓ steps by decreasing its subscript.

While the object W waits, the objects C_j and K_j that were actually produced (there are at most ℓ of them) move through the positive membrane Q and inside the membrane S of the selected copy of Π_ℓ, thus reaching their initial position. At that point, the subscript of W will have reached 0. The object W_0 then enters the active instance of Π_ℓ, and inside membrane S it produces by evolution the objects $\text{I}_{\ell-m+1}$, O_{t+2}, and NO_{t+4}, and is itself rewritten into W'_0. Notice how the subscripts of I, O, and NO are incremented by one: this is done in order to give the opportunity to $\text{I}_{\ell-m+1}$ to move (as $\text{I}_{\ell-m}$) to its initial position inside membrane E, and also to W'_0 to exit Π_ℓ and move back to membrane Q.

While W'_0 moves to membrane S, the chosen embedded P system Π_ℓ finally starts computing as if it were in stand-alone form, as in Section 5.3. This computation requires a polynomial amount of time, after which either the object YES or NO will be sent out to the surrounding membrane Q. That

result object then exits Q, setting its charge to negative (thus signalling that the enclosed P system Π_ℓ has been used, and cannot be reused again) and moving to membrane A. When entering it, the charge is set to positive (if the result object is YES) or negative (if it is NO); the result object is simultaneously rewritten into $\#$.

When membrane A is non-neutral, the object w'_0 can enter it, and be sent out as h_{0,q_Y} or h_{0,q_N} depending on the result; the charge of A is also reset to neutral to allow further queries. The simulation now proceeds again as in Section 4.3.1, until another oracle query is made or until the computation of M finally terminates.

6.2.2 Technical details

When the simulated Turing machine enters the query state, an object $\text{h}_{i,q_?}$ appears inside the outermost membrane s of $\Pi_{M,n}$. It is immediately subject to the following evolution rule, which is replicated for $0 \leq i \leq 2p(n)$:

$$[\text{h}_{i,q_?} \to \text{h}'_{i,q_?} \; \text{c}_{1,i} \; \cdots \; \text{c}'_{\ell-m,i+\ell-m-1} \; \text{k}'_{0,i+\ell-m} \; \cdots \; \text{k}'_{m-1,i+\ell-1}]^0_\text{s}$$

During the next step, the objects $\text{c}'_{j,i}$ enter the corresponding membrane labelled by i, where they are either deleted (if that membrane represents a tape cell containing 1, i.e., if the j-th clause occurs in the input formula φ), or sent back out as c_j (if the tape cell contains 0).

$$\left.\begin{array}{l} \text{c}'_{j,i} \; [\;]^\alpha_i \to [\text{c}'_{j,i}]^\alpha_i \\ [\text{c}'_{j,i} \to \#]^+_i \\ [\text{c}'_{j,i}]^-_i \to [\;]^-_i \; \text{c}_j \end{array}\right\} \text{for } 0 \leq i \leq 2p(n) \text{ and } 1 \leq j \leq \ell - m \text{ and } \alpha \in \{+,-\}$$

The same occurs to the objects $\text{k}_{j,i}$, except that they are deleted when the corresponding tape cell contains a 0, according to the encoding performed by the machine E_L.

$$\left.\begin{array}{l} \text{k}'_{j,i} \; [\;]^\alpha_i \to [\text{k}'_{j,i}]^\alpha_i \\ [\text{k}'_{j,i} \to \#]^-_i \\ [\text{k}'_{j,i}]^+_i \to [\;]^+_i \; \text{k}_j \end{array}\right\} \text{for } 0 \leq i \leq 2p(n) \text{ and } 0 \leq j \leq m-1 \text{ and } \alpha \in \{+,-\}$$

While the objects encoding the THRESHOLD-3SAT instance are produced, the object $\text{h}'_{i,q_?}$ changes the charge of one of the neutral membranes labelled by Q to positive, and moves back to s as w_ℓ.

$$\left.\begin{array}{l} \text{h}'_{i,q_?} \; [\;]^0_\text{Q} \to [\text{h}_{i,q_?}]^+_\text{Q} \\ [\text{h}'_{i,q_?}]^+_\text{Q} \to [\;]^+_\text{Q} \; \text{w}_\ell \end{array}\right\} \text{for } 0 \leq i \leq 2p(n)$$

The objects c_j and k_j (which are at most ℓ in number) are then sequentially moved to the system Π_ℓ corresponding to the positive membrane Q.

$$\left.\begin{aligned} \text{c}_j\ [\ ]_{\text{Q}}^{+} &\to [\text{c}_j]_{\text{Q}}^{+} \\ \text{c}_j\ [\ ]_{\text{IN}}^{0} &\to [\text{c}_j]_{\text{IN}}^{0} \end{aligned}\right\} \text{for } 1 \le j \le \ell - m$$

$$\left.\begin{aligned} \text{k}_j\ [\ ]_{\text{Q}}^{+} &\to [\text{k}_j]_{\text{Q}}^{+} \\ \text{k}_j\ [\ ]_{\text{IN}}^{0} &\to [\text{k}_j]_{\text{IN}}^{0} \end{aligned}\right\} \text{for } 0 \le j \le m - 1$$

The object w_ℓ "waits" until the last object has entered membrane Q by decreasing its subscript

$$[\text{w}_i \to \text{w}_{i-1}]_{\text{S}}^{0} \qquad \text{for } 1 \le i \le \ell$$

then it also enters the selected embedded P system Π_ℓ in order to initialise its configuration:

$$\begin{aligned} &\text{w}_0\ [\ ]_{\text{Q}}^{+} \to [\text{w}_0]_{\text{Q}}^{+} \\ &\text{w}_0\ [\ ]_{\text{IN}}^{0} \to [\text{w}_0]_{\text{IN}}^{0} \\ &[\text{w}_0 \to \text{w}_0'\ \text{i}_{\ell-m+1}\ \text{o}_{t+2}\ \text{no}_{t+4}]_{\text{IN}}^{0} \end{aligned}$$

The object w_0' is sent back to membrane S, while the newly created objects reach their actual position and/or subscript inside Π_ℓ, allowing the actual computation of the embedded P system to start.

$$\begin{aligned} &[\text{w}_0']_{\text{IN}}^{0} \to [\ ]_{\text{IN}}^{0}\ \text{w}_0' \\ &[\text{w}_0']_{\text{Q}}^{+} \to [\ ]_{\text{Q}}^{+}\ \text{w}_0' \\ &\text{i}_{\ell-m+1}\ [\ ]_{\text{E}}^{0} \to [\text{i}_{\ell-m}]_{\text{E}}^{0} \\ &[\text{o}_{t+2} \to \text{o}_{t+1}]_{\text{IN}}^{0} \\ &[\text{no}_{t+4} \to \text{no}_{t+3}]_{\text{IN}}^{0} \end{aligned}$$

When the computation of the active instance of Π_ℓ terminates, a result object is sent out to membrane Q, and it is moved in order to set the charge of membrane A appropriately.

$$\begin{aligned} &[\text{yes}]_{\text{Q}}^{+} \to [\ ]_{\text{Q}}^{-}\ \text{yes} \\ &[\text{no}]_{\text{Q}}^{+} \to [\ ]_{\text{Q}}^{-}\ \text{no} \\ &\text{yes}\ [\ ]_{\text{A}}^{0} \to [\#]_{\text{A}}^{+} \\ &\text{no}\ [\ ]_{\text{A}}^{0} \to [\#]_{\text{A}}^{-} \end{aligned}$$

Now the object w_0 can "read" the answer from the charge of A, reset it to neutral and be rewritten into the corresponding object encoding the new state of the simulated machine M.

$$\begin{aligned} &\text{w}_0\ [\ ]_{\text{A}}^{\alpha} \to [\text{w}_0]_{\text{A}}^{\alpha} \qquad \text{for } \alpha \in \{+, -\} \\ &[\text{w}_0]_{\text{A}}^{+} \to [\ ]_{\text{A}}^{0}\ \text{h}_{0,q_{\text{Y}}} \\ &[\text{w}_0]_{\text{A}}^{-} \to [\ ]_{\text{A}}^{0}\ \text{h}_{0,q_{\text{N}}} \end{aligned}$$

Now the simulation of M continues as in Section 4.3.1.

6.2.3 Main result

We are finally able to prove the result anticipated above.

Theorem 46. $\mathbf{P}^{\mathbf{PP}} \subseteq \mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$.

Proof. The previous discussion shows how any polynomial-time Turing machine equipped with an oracle for THRESHOLD-3SAT can be simulated with a polynomial slowdown. Since THRESHOLD-3SAT is **PP**-hard, any other problem in **PP** can be efficiently reduced to it by the simulated Turing machine before performing the query. As a consequence, the whole class $\mathbf{P}^{\mathbf{PP}}$ is included in $\mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$. □

By Toda's theorem ($\mathbf{PH} \subseteq \mathbf{P}^{\mathbf{PP}}$) [37], this result implies that the whole polynomial hierarchy **PH** is contained in $\mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$, bringing this class closer to the known **PSPACE** upper bound [35].

Chapter 7

Conclusions

This work started as an attempt to analyse in formal terms the advantage given by allowing P systems with active membranes the use of an exponential workspace, while restricting the amount of available time to polynomial.

We first defined a measure of space complexity for P systems, first by considering them from an (approximated) biological and physical standpoint (rather than a purely mathematical point of view), and then abstracting by taking into account our planned use of this measure, that is, the asymptotical analysis of the space required to solve computationally hard problems. The resulting notion of space complexity is mathematically simple and similar in nature to that of Turing machines.

The computing power of polynomial-space families of P systems with active membranes has then been analysed, showing that it coincides with the computing power of polynomial-space Turing machines. Mutual simulations among the two kinds of devices have been instrumental in proving this result.

By allowing exponential space, we also improved the previously known results related to P systems using elementary membrane division. Starting from the known inclusion of **NP** in $\mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$, we first improved the result to $\mathbf{PP} \subseteq \mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$, then finally to $\mathbf{P}^{\mathbf{PP}} \subseteq \mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$. The latter result builds on the intermediate one and on the simulation of Turing machines mentioned above; the way they have been presented here actually reflects the way these results were obtained during the research.

In the opinion of the present author, the topics discussed in this thesis have another interesting aspect: most of the definitions can be immediately transferred to other variants of P systems, and some work in progress seems to suggest that some of the actual results (in particular, the solution to **PP**-complete problems) may also apply there.

7.1 Open problems

Although this work provides some new and improved results in the area of the complexity theory of P systems, several open questions remain to be settled.

Our characterisation of the computational power of P systems with active membranes working under space restrictions only holds as far as polynomial space is involved. As soon as we try to investigate exponential-space bounded P systems, the techniques we described here break down. Indeed, our simulation of Turing machines relies on the ability to create one membrane per tape cell, and all those membranes required different labels. While creating exponentially many membranes by division is possible, they do not possess exponentially many individual labels, and it is doubtful that they can be correctly identified by a "tape head object" moving back and forth, as in the simulation of Section 4.3.1.

We also think it might be worth investigating sub-polynomial space P systems, particularly those working in logarithmic space, in order to establish whether their behaviour differs from similarly limited Turing machines. In this case, another complication arises: the polynomial-time uniformity conditions are too powerful (as $\mathbf{L} \subseteq \mathbf{P}$), and they could be used for "cheating" and solving the problem during the construction phase. The most logical solution here seems to be the use of weaker uniformity conditions such as $\mathbf{AC}^0$ circuits, taking inspiration from the research on complexity classes for P systems below $\mathbf{P}$ [21].

Finally, a precise characterisation of the efficiency gained by trading space for time when using P systems with elementary active membranes is still missing. While or $\mathbf{P}^{\mathbf{PP}} \subseteq \mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$ result might suggest that the latter class could be equal to **PSPACE**, proving this result does not seem to be a trivial task. On the other hand, we find no reason to consider the result $\mathbf{P}^{\mathbf{PP}} = \mathbf{PMC}_{\mathcal{AM}(-\mathrm{d},-\mathrm{n})}$ implausible, and it would probably be more interesting and insightful, as complexity classes for P systems often tend to be identified either with **P** or **PSPACE**.

Bibliography

[1] Scott Aaronson. Why philosophers should care about computational complexity. Technical report, http://eccc.hpi-web.de/report/2011/108/, 2011.

[2] Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021–1024, 1994.

[3] Artiom Alhazov, Liudmila Burtseva, Svetlana Cojocaru, and Yurii Rogozhin. Solving PP-complete and #P-complete problems by P systems with active membranes. In David W. Corne, Pierluigi Frisco, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing 9th International Workshop, WMC 2008*, volume 5931 of *Lecture Notes in Computer Science*, pages 108–117. Springer, 2009.

[4] Artiom Alhazov, Carlos Martín-Vide, and Linqiang Pan. Solving a PSPACE-complete problem by recognizing P systems with restricted active membranes. *Fundamenta Informaticae*, 58(2):67–77, 2003.

[5] Roberto Barbuti, Andrea Maggiolo-Schettini, Paolo Milazzo, Giovanni Pardini, and Luca Tesei. Spatial P systems. *Natural Computing*, 10(1):3–16, 2011.

[6] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71*, pages 151–158, 1971.

[7] Jack Copeland, editor. *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life, Plus the Secrets of Enigma.* Oxford University Press, 2004.

[8] Martin Davis, editor. *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions.* Raven Press, 1965.

[9] Jack Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[10] Lance Fortnow and Steve Homer. A short history of computational complexity. *Bulletin of the European Association for Theoretical Computer Science*, 80:95–133, 2003.

[11] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[12] John T. Gill. Computational complexity of probabilistic Turing machines. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing, STOC '74*, pages 91–95, 1974.

[13] Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[14] Tom Head. Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors. *Bulletin of Mathematical Biology*, 49(6):737–759, 1987.

[15] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–104. Plenum Press, 1972.

[16] Tsuneyoshi Kuroiwa, Haruko Kuroiwa, Atsushi Sakai, Hidenori Takahashi, Kyoko Toda, and Ryuuichi Itoh. The division apparatus of plastids and mitochondria. *International Review of Cytology*, 181:1–41, 1998.

[17] Carlos Martín-Vide, Gheorghe Păun, Juan Pazos, and Alfonso Rodríguez-Patón. Tissue P systems. *Theoretical Computer Science*, 296(2):295–326, 2003.

[18] Giancarlo Mauri, Alberto Leporati, Antonio E. Porreca, and Claudio Zandron. Computational complexity aspects in membrane computing. In Fernando Ferreira, Benedikt Löwe, Elvira Mayordomo, and Luís Mendes Gomes, editors, *Programs, Proofs, Processes, 6th Conference on Computability in Europe, CiE 2010*, volume 6158 of *Lecture Notes in Computer Science*, pages 317–320. Springer, 2010.

[19] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 7:115–133, 1943.

[20] Marin Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.

[21] Niall Murphy and Damien Woods. The computational power of membrane systems under tight uniformity conditions. *Natural Computing*, 10(1):613–632, 2011.

[22] Adam Obtułowicz. Note on some recursive family of P systems with active membranes. `http://ppage.psystems.eu/index.php/Papers`, 2001.

[23] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1993.

[24] Mario J. Pérez-Jiménez, Álvaro Romero-Jiménez, and Fernando Sancho-Caparrini. Complexity classes in models of cellular computing with membranes. *Natural Computing*, 2(3):265–284, 2003.

[25] Antonio E. Porreca, Alberto Leporati, Giancarlo Mauri, and Claudio Zandron. Introducing a space complexity measure for P systems. *International Journal of Computers, Communications & Control*, 4(3):301–310, 2009.

[26] Antonio E. Porreca, Alberto Leporati, Giancarlo Mauri, and Claudio Zandron. Elementary active membranes have the power of counting. *International Journal of Natural Computing Research*, 2(3):329–342, 2011.

[27] Antonio E. Porreca, Alberto Leporati, Giancarlo Mauri, and Claudio Zandron. P systems simulating oracle computations. In Marian Gheorghe, Gheorhge Păun, and Sergey Verlan, editors, *Pre-Proceedings of the Twelfth International Conference on Membrane Computing 2011 (CMC12)*, pages 433–445, 2011.

[28] Antonio E. Porreca, Alberto Leporati, Giancarlo Mauri, and Claudio Zandron. P systems with active membranes: Trading time for space. *Natural Computing*, 10(1):167–182, 2011.

[29] Antonio E. Porreca, Alberto Leporati, Giancarlo Mauri, and Claudio Zandron. P systems with elementary active membranes: Beyond NP and coNP. In Marian Gheorghe, Thomas Hinze, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing, 11th International Conference, CMC 2010*, volume 6501 of *Lecture Notes in Computer Science*, pages 338–347. Springer, 2011.

[30] Antonio E. Porreca, Giancarlo Mauri, and Claudio Zandron. Complexity classes for membrane systems. *RAIRO Theoretical Informatics and Applications*, 40(2):141–162, 2006.

[31] Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.

[32] Gheorghe Păun. P systems with active membranes: Attacking NP-complete problems. *Journal of Automata, Languages and Combinatorics*, 6(1):75–90, 2001.

[33] Gheorghe Păun. Membrane computing: History and brief introduction. In Erol Gelenbe and Jean-Pierre Kahane, editors, *Fundamental Concepts in Computer Science*, pages 17–41. Imperial College Press, 2009.

[34] Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors. *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.

[35] Petr Sosík and Alfonso Rodríguez-Patón. Membrane computing and complexity theory: A characterization of PSPACE. *Journal of Computer and System Sciences*, 73(1):137–152, 2007.

[36] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976.

[37] Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 20(5):865–877, 1991.

[38] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[39] Andrea Valsecchi, Antonio E. Porreca, Alberto Leporati, Giancarlo Mauri, and Claudio Zandron. An efficient simulation of polynomial-space Turing machines by P systems with active membranes. In Gheorghe Păun, Mario J. Pérez-Jiménez, Agustín Riscos-Núñez, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing, 10th International Workshop, WMC 2009*, volume 6501 of *Lecture Notes in Computer Science*, pages 461–478. Springer, 2010.

[40] Heribert Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Texts in Theoretical Computer Science: An EATCS Series. Springer, 1999.

[41] Richard Zach. Hilbert's program then an now. In Dale Jacquette, editor, *Philosophy of Logic*, volume 5, pages 411–447. Elsevier, 2006.

[42] Claudio Zandron, Claudio Ferretti, and Giancarlo Mauri. Solving NP-complete problems using P systems with active membranes. In Ioannis Antoniou, Cristian S. Calude, and Michael J. Dinneen, editors, *Unconventional Models of Computation, UMC'2K, Proceedings of the Second International Conference*, pages 289–301. Springer, 2001.