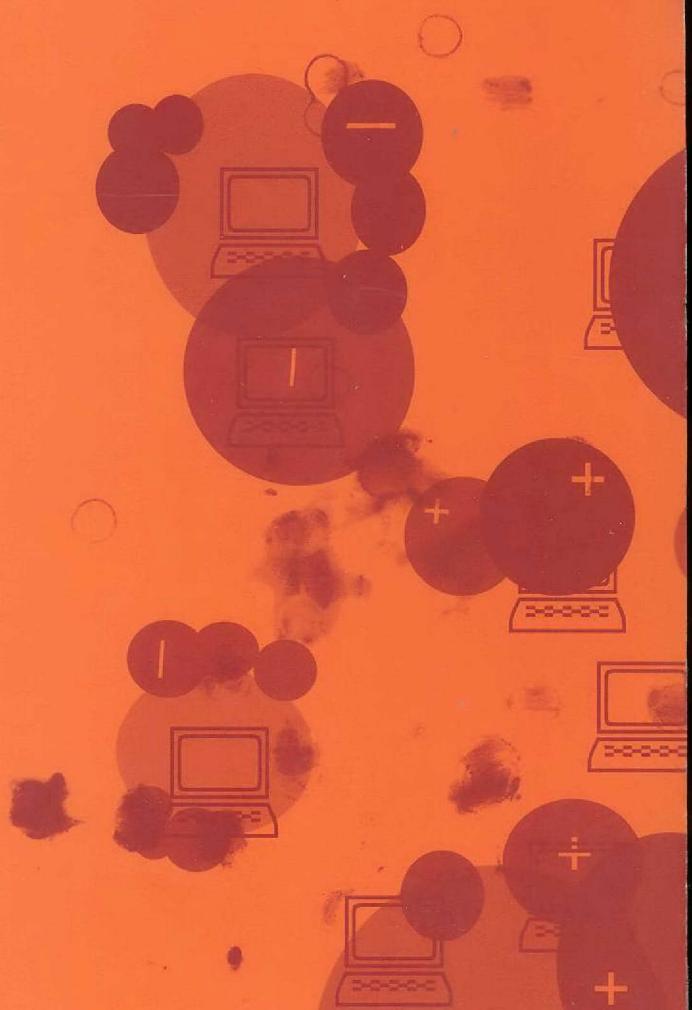


COMPUTING WITH CELLS AND ATOMS

An introduction to quantum, DNA
and membrane computing

Cristian S. Calude and Gheorghe Păun



Computing with Cells and Atoms

Computing with Cells and Atoms

An introduction to quantum, DNA and membrane computing

Cristian S. Calude
Gheorghe Păun



London and New York

First published 2001 by Taylor & Francis
11 New Fetter Lane, London EC4P 4EE

Simultaneously published in the USA and Canada
by Taylor & Francis Inc,
29 West 35th Street, New York, NY 10001

Taylor & Francis is an imprint of the Taylor & Francis Group

© 2001 Cristian S. Calude and Gheorghe Păun

Printed and bound in Great Britain by
Biddles Ltd, Guildford and King's Lynn

All rights reserved. No part of this book may be reprinted or reproduced or utilised in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying and recording, or in any information storage or retrieval system, without permission in writing from the publishers.

Every effort has been made to ensure that the advice and information in this book is true and accurate at the time of going to press. However, neither the publisher nor the authors can accept any legal responsibility or liability for any errors or omissions that may be made. In the case of drug administration, any medical procedure or the use of technical equipment mentioned in this book, you are strongly advised to consult the manufacturer's guidelines.

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

Library of Congress Cataloging in Publication Data

Calude, Cristian, 1952-

Computing with cells and atoms: an introduction to quantum, DNA and membrane computing/Cristian S. Calude, Gheorghe Păun.

p. cm.

Includes bibliographical references and index.

1. Quantum computers. 2. Molecular computers. I. Păun, Gheorghe, 1950- II. Title.

QA76.889.C35 2000

004.1-dc21

00-057745

ISBN 0-7484-0899-1

Contents

<i>Preface</i>	vii
1 Prerequisites	1
1.1 Preliminary Notions and Notations	1
1.2 Operations on Strings and Languages	2
1.3 A General Computing Framework	3
1.4 Chomsky Grammars	8
1.5 Lindenmayer Systems	10
1.6 Automata and Transducers	11
1.7 Characterizations of Computably Enumerable Languages	14
1.8 Universal Turing Machines and Type-0 Grammars	16
1.9 Complexity	19
1.10 Bibliographical Notes	21
2 DNA Computing	23
2.1 The Structure of DNA	23
2.2 Complementarity Induces Computational Completeness	26
2.3 Operations on DNA Molecules	30
2.4 Adleman's Experiment	36
2.5 Other DNA Solutions to NP Complete Problems	43
2.6 A Two-Dimensional Generalization	50
2.7 Computing by Carving	55
2.8 Sticker Systems	64
2.9 Extended H Systems	77
2.10 Controlled H Systems	86
2.11 Distributed H Systems	95
2.12 Bibliographical Notes	106
3 Membrane Computing	109
3.1 P Systems with Labelled Membranes	110
3.2 Examples	117
3.3 The Power of P Systems	121
3.4 Decidability Results	127

3.5	Rewriting P Systems	131
3.6	P Systems with Polarized Membranes	135
3.7	Normal Forms	144
3.8	P Systems on Asymmetric Graphs	154
3.9	P Systems with Active Membranes	156
3.10	Splicing P Systems	168
3.11	Variants, Problems, Conjectures	176
3.12	Bibliographical Notes	178
4	Quantum Computing	179
4.1	The Church–Turing Thesis	179
4.2	Computation is Physical	181
4.3	Reversible Computation	183
4.4	The Copy Computer	186
4.5	Maxwell’s Demon	187
4.6	Quantum World	189
4.7	Bits and Qubits	190
4.8	Quantum Calculus	191
4.9	Qubit Evolution	195
4.10	No Cloning Theorem	203
4.11	Measurements	204
4.12	Zeno Machines	206
4.13	Inexhaustible Uncertainty	209
4.14	Randomness	213
4.15	The EPR Conundrum and Bell’s Theorem	216
4.16	Quantum Logic	223
4.17	Have Quantum Propositions Classical Meaning?	226
4.18	Quantum Computers	234
4.19	Quantum Algorithms	241
4.20	Quantum Complexity	263
4.21	Quantum Cryptography	272
4.22	Information and Teleportation	274
4.23	Computing the Uncomputable?	279
4.24	Bibliographical Notes	280
5	Final Remarks	283
	<i>Bibliography</i>	285
	<i>Index</i>	307

Preface

*Quand je vous aimer?
Ma foi, je ne sais pas.
Peut-être jamais,
Peut-être demain!*
Carmen

The computers as we know them today, based on silicon chips, are getting better and better, and are doing more and more for us; in practice, we can no longer live without our intelligent prostheses. Nonetheless, they still give rise to frustrations, and not just among academics. Though ever faster in speed and ever larger size, silicon-based computers are not yet able to cope with many tasks of practical interest. There are still too many problems that are effectively intractable because to “solve” them using “classical” machines requires a huge amount of computing time. It seems that progress in electronic hardware (and the corresponding software engineering) is not enough; for instance, the miniaturization is approaching the quantum boundary, where physical processes obey laws based on probabilities and non-determinism, something almost completely absent in the operation of “classical” computers. So, new breakthrough is needed.

The idea of such a breakthrough is not really new. It involves going “to the bottom”, where – Feynman assures us – “there is plenty of room”. So one possible way to proceed is to get closer to the innermost structure of matter, beyond *micro*, into *nano*, even into *femto*. This means computing directly with molecules and atoms, maybe even with their parts.

From the physical point of view, this means quantum physics, an incredible successful, but controversial subject with a great history and a most promising future. One hope is for a *quantum computer*, a computer of a completely different type that can make use of the strange but wonderful phenomena that occur at the atomic and intra-atomic levels. However, there are problems: just think about the wave-particle duality of quanta, about the possibility of a particle existing in several places at the same time and of several particles being in the same place at the same time, about teleportation, and so forth. There are a number of challenging difficulties from the computability point of view: for instance, that observation modifies the

observed system, which means that reading some information destroys that information, or that quantum interference of particles cannot be controlled using current technology.

Fortunately, for billions of years nature itself has been “computing” with molecules and cells. When speaking about computability in nature, we think it is better to use quotation marks; we do not accept that nature computes except in a metaphorical sense. Bees know no geometry, rain drops solve no differential equation when falling. Only man sees hexagons in beehives and a potential field on the surface of a water droplet. However, less metaphorical and so more visible is the way nature has used (and is still using ...) DNA in evolution. Thus another hope for the future is to use DNA and other proteins, manipulated by tools already known to biochemistry and genetic engineering (for instance, enzymes, “one of the most intelligent gifts nature has made to us”, as it has been said), and to develop “wet computers”, with core chips consisting of bio-molecules.

Both these ideas, Quantum Computing and DNA Computing, have been explored intensively in recent years. A third idea, explored only recently, is also biochemically inspired: Membrane Computing. This seeks to make use of the way nature “computes” at the cellular level, where an intricate processing of materials, energy, and information takes place in a membrane structure which determines the architecture of the “cell-computer” and actively participate in the “computation”.

The aim of this book is to introduce the reader to these fascinating areas, currently located towards the science-fiction edge of science: *computing with cells and atoms*.

Although mathematically oriented (more precisely, mathematics of the kind involved in theoretical computer science), the book does not go into full formal detail (complete proofs, for instance). Our intention is to present basic notions and results from the three domains mentioned above, Quantum Computing, DNA Computing, and Membrane Computing, providing just a glimpse of each. Of course our selection is biased and partisan, favouring subjects which we have dealt with in our personal research, but a number of fundamental matters are included: Adleman’s experiment and its continuations, in the DNA area, and Shor’s algorithm, in the quantum area.

Because our interest (and our competence) tends toward formal models of computability, we shall skip over much “practical” information to do with physics and biochemistry. The general strategy we adopt is to look to reality (whatever this means;¹ passively, but safely, for us *reality* is what we can find in books and call such) in search of *data supports* (hence *data structures*) and *operations* on these data structures. With these ingredients we can define a process (in the form of a sequence of *transitions* among *configurations* describing states of a *system*) which, provided that an input and an output can

¹“We are both onlookers and actors in the great drama of existence”, according to N. Bohr.

be associated with it, is considered a *computation*. The complete machinery is individualized as a *computing system*. Although this is crucial from a practical point of view, we do not restrict ourselves to “realistic” systems, implementable today or in the immediate future. We look for *plausible* computers and, in the main, for models of them. Progress in technology, physical or biochemical, is so rapid (and unpredictable) that it is not efficient to limit consideration to knowledge and tools that are available today (or tomorrow). In a sense, we are replaying Turing’s game: just consider machines with infinite tapes and prove universality theorems about them; maybe, one day this will at least encourage some practitioners to build real devices that resemble these machines.

In the areas of DNA and Membrane Computing, we are inspired by what happens *in vivo* and we try to approach as possible what can be done *in vitro*, but we work *in info*, in symbolic terms. In Quantum Computing we also look at possibilities of transcending Turing computability barrier.

The constant test bed for the new computability devices we consider will be the usual hierarchies in computability theory, namely Turing and Chomsky classes, and complexity classes. More specifically, we are interested in attaining the power of Turing machines which (according to the Church–Turing thesis) have the most comprehensive level of algorithmic computability, if possible, in a constructive manner. Constructively simulating a Turing machine by means of a device from a new class of computing machines provides us with universality for free: starting from a universal Turing machine we get a universal device in our new class. This is important if we are looking for *programmable* computers of the new type. Programmability means universality. In the sense of Conrad [63], it also means non-learnability and/or non-efficiency (Conrad advocates convincingly a general trade-off between programmability, learnability and efficiency), but where we do not consider such “details”. The particular domains of Quantum and DNA Computing (not to speak about Membrane Computing) are not yet able to deal with such questions.

In order to make the book as self-contained as possible, we start by specifying a series of prerequisites drawn variously from automata and language theory, complexity, quantum physics and the biochemistry of DNA. Of course, some broad familiarity with these subjects would be useful, but we “pretend” that the book can be read by any (mathematically) educated layman.

Acknowledgements

Parts of the work brought together in this book have been done while the authors have visited a number of institutions. The first author has been supported by a research grant of the University of Auckland Research Committee, Sandia National Laboratories, Albuquerque (April 1999), the Technical University of Vienna (May–July 1999), the Japanese Advanced Institute of Science and Technology and the Japanese Ministry of Education, Science, Sports and Culture (September–December 1999). The second au-

thor has benefited from research in Tokyo (April-May 1999 and supported by the “Research for Future” Program no. JSPS-RFTF 96I00101, from the Japanese Society for the Promotion of Science), Magdeburg (July-August 1999, supported by the Alexander von Humboldt Foundation), and Turku (September-October 1999, supported by the Academy of Finland, Project 137358).

The authors are much indebted to Ioannis Antoniou, Elena Calude, John Casti, Greg Chaitin, Michael Dinneen, Peter Hertling, Hajime Ishihara, Sorin Istrail, Seth Lloyd, David Mermin, Georgio Odifreddi, Bernard Ömer, Grzegorz Rozenberg, Boris Pavlov, Ilya Prigogine, Arto Salomaa, Shao Chin Sung, Karl Svozil, Marius Zimand for comments, discussions and criticism. Special thanks are due to Dilys Alam, Luke Hacker, Tony Moore, Grant Soanes, to the Taylor and Francis team in London and to Steven Gardiner Ltd of Cambridge for their most pleasant and efficient co-operation.

Cristian S. Calude
Gheorghe Păun

Chapter 1

Prerequisites

The aim of this chapter is to introduce the notions and notation of theoretical computer science we shall use in the subsequent chapters, thus making the book self-contained from this point of view. Of course, some previous familiarity with these notions would be useful; a reader familiar with the field can consult this chapter only when need arises.

1.1 Preliminary Notions and Notations

The family of subsets of a set X is denoted by $\mathcal{P}(X)$; if X is an infinite set, then we denote by $\mathcal{P}_f(X)$ the family of finite subsets of X . The cardinality of X is denoted by $card(X)$. The set of natural numbers, $\{0, 1, 2, \dots\}$ is denoted by \mathbb{N} . The empty set is denoted by \emptyset .

An *alphabet* is a finite nonempty set of abstract symbols. For an alphabet V we denote by V^* the set of all strings of symbols in V . The empty string is denoted by λ . The set of nonempty strings over V , that is $V^* - \{\lambda\}$, is denoted by V^+ . Each subset of V^* is called a *language* over V . A language which does not contain the empty string (hence it is a subset of V^+) is said to be λ -*free*.

The *length* of a string $x \in V^*$ (the number of symbol occurrences in x) is denoted by $|x|$. The number of occurrences of a given symbol $a \in V$ in $x \in V^*$ is denoted by $|x|_a$. If $x \in V^*$, $U \subseteq V$, then by $|x|_U$ we denote the length of the string obtained by erasing from x all symbols not in U , that is,

$$|x|_U = \sum_{a \in U} |x|_a.$$

If $V = \{a_1, a_2, \dots, a_n\}$ (the order of symbols is important) and $w \in V^*$, then $\Psi_V(w) = (|w|_{a_1}, \dots, |w|_{a_n})$ is the *Parikh vector* associated with w (and V). For a language $L \subseteq V^*$, $\Psi_V(L) = \{\Psi_V(w) \mid w \in L\}$ is called the *Parikh set* of L .

A *multiset* (over a set X) is a mapping $M : X \longrightarrow \mathbf{N} \cup \{\infty\}$. For $a \in X$, $M(a)$ is called the multiplicity of a in the multiset M . The *support* of M is the set $\text{supp}(M) = \{a \in X \mid M(a) > 0\}$. A multiset M of finite support, $\text{supp}(M) = \{a_1, \dots, a_n\}$ can be written in the form $\{(a_1, M(a_1)), \dots, (a_n, M(a_n))\}$. We can also represent this multiset by the string $w(M) = a_1^{M(a_1)} \dots a_n^{M(a_n)}$, as well as by any permutation of $w(M)$. Conversely, having a string $w \in V^*$, we can associate with it the multiset $M(w) : V \longrightarrow \mathbf{N} \cup \{\infty\}$ defined by $M(w)(a) = |w|_a, a \in V$.

For two multisets M_1, M_2 (over the same set X) we say that M_1 is included in M_2 , and we write $M_1 \subseteq M_2$ if $M_1(a) \leq M_2(a)$ for all $a \in X$. The union of M_1, M_2 is the multiset $M_1 \cup M_2$ defined by $(M_1 \cup M_2)(a) = M_1(a) + M_2(a)$. We define here the difference $M_2 - M_1$ of two multisets only if $M_1 \subseteq M_2$ and this is the multiset defined by $(M_2 - M_1)(a) = M_2(a) - M_1(a)$.

1.2 Operations on Strings and Languages

The union and the intersection of two languages L_1, L_2 are denoted by $L_1 \cup L_2, L_1 \cap L_2$, respectively. The *concatenation* of L_1, L_2 is $L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}$.

We define further:

$$\begin{aligned} L^0 &= \{\lambda\}, \\ L^{i+1} &= LL^i, \quad i \geq 0, \\ L^* &= \bigcup_{i=0}^{\infty} L^i \text{ (the * -Kleene closure),} \\ L^+ &= \bigcup_{i=1}^{\infty} L^i \text{ (the + -Kleene closure).} \end{aligned}$$

A mapping $h : V \longrightarrow U^*$, extended to $s : V^* \longrightarrow U^*$ by $h(\lambda) = \{\lambda\}$ and $h(x_1x_2) = h(x_1)h(x_2)$, for $x_1, x_2 \in V^*$, is called a *morphism*. If $\lambda \notin h(a)$, for each $a \in V$, then h is a λ -*free* morphism.

A morphism $h : V^* \longrightarrow U^*$ is called a *coding* if $h(a) \in U$ for each $a \in V$ and a *weak coding* if $h(a) \in U \cup \{\lambda\}$ for each $a \in V$. If $h : (V_1 \cup V_2)^* \longrightarrow V_1^*$ is the morphism defined by $h(a) = a$ for $a \in V_1$, and $h(a) = \lambda$ otherwise, then we say that h is a *projection* (associated with V_1) and we denote it by pr_{V_1} .

For $x, y \in V^*$ we define their *shuffle* by

$$\begin{aligned} x \amalg y &= \{x_1y_1 \dots x_ny_n \mid x = x_1 \dots x_n, y = y_1 \dots y_n, \\ &\quad x_i, y_i \in V^*, 1 \leq i \leq n, n \geq 1\}. \end{aligned}$$

The *mirror image* of a string $x = a_1a_2 \dots a_n$, for $a_i \in V, 1 \leq i \leq n$, is the string $mi(x) = a_n \dots a_2a_1$.

In general, if we have an n -ary operation on strings, $g : V^* \times \dots \times V^* \longrightarrow \mathcal{P}(U^*)$, we extend it to languages over V by

$$g(L_1, \dots, L_n) = \bigcup_{\substack{x_i \in L_i \\ 1 \leq i \leq n}} g(x_1, \dots, x_n).$$

For instance, $mi(L) = \{mi(x) \mid x \in L\}$.

A family FL of languages is *closed* under an n -ary operation g if, for all languages L_1, \dots, L_n in FL , the language $g(L_1, \dots, L_n)$ is also in FL .

1.3 A General Computing Framework

The aim of this section is to “define”, in a general, somewhat formal, but not in a very precise way, the notions of *computation* and *computing device/model*. In fact, we are not looking for a comprehensive definition, but for a comprehensive *framework* for an activity which can be called a computation. We do not address here semiotic or philosophical questions (can a computation be a natural process, or is it a specific human artifact? is a process a computation *per se*, or only for some observer? and so on), but we step into an operational direction: we want to systematize the notions we can encounter when devising a computing machinery, in a comprehensive manner which is also general enough. Otherwise stated, we do not adopt a minimalistic approach, like Turing’s one, but a globalistic one, able to cover all/most of the computing devices we will discuss in the subsequent chapters.

A *computation* takes place in an *environment* where the following three elements are available:

- a set S , potentially infinite, of *states* (or *configurations*);
- a *transition* relation on this set, $\rightarrow \subseteq S \times S$; any sequence of transitions between elements of S , $s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_n$, for $s_1, \dots, s_n \in S, n \geq 1$ (we write $s_1 \rightarrow^* s_n$; we also accept that $s_1 = s_n$ and we say that \rightarrow^* is the reflexive and transitive closure of \rightarrow), is called a *computation*; we denote by \mathcal{K} the set of all computations with respect to S and \rightarrow ;
- a *stop* predicate $\pi : \mathcal{K} \longrightarrow \{0, 1\}$; by \mathcal{K}_π we denote the set of computations which stop correctly with respect to π , that is, $\mathcal{K}_\pi = \{s_1 \rightarrow^* s_2 \in \mathcal{K} \mid \pi(s_1 \rightarrow^* s_2) = 1\}$.

In order to assign a *result* to a computation, we need two further items:

- a set R of possible results;
- an *output* mapping $\rho : \mathcal{K}_\pi \longrightarrow R$.

Let us denote by Γ the system of the five elements mentioned above,

$$\Gamma = (S, \rightarrow, \pi, R, \rho).$$

Such a system defines two mappings, one from S to the power set of S , and one from S to the power set of R :

$$\begin{aligned}\Gamma(s_1) &= \{s_2 \mid s_1 \rightarrow^* s_2 \in \mathcal{K}_\pi\}, \text{ for each } s_1 \in S, \\ res_\Gamma(s_1) &= \{\rho(s_1 \rightarrow^* s_2) \mid s_2 \in S, s_1 \rightarrow^* s_2 \in \mathcal{K}_\pi\}.\end{aligned}$$

In some sense, we have here two levels of observing the system, one which deals only with states (configurations), without assigning a “meaning” to these states or to sequences of transitions between them, and one which also assigns a “result” to a correctly completed computation. It is important to note that the result is associated with the whole computation, not only with the last configuration, the halting one.

The previous setup is rather vague, but already at this stage we can introduce a series of important computability concepts: determinism, equivalence, universality, reversibility, time complexity. We do not approach at this moment such questions, but we make one further step “toward reality”, specifying less general ingredients of a computing framework.

We call a *computing framework* a construct where the following items can be identified (some of them can also be present by default):

- An *alphabet* A , whose elements are identified with symbols, without parts, without a structure (they are *atoms*, in the etymological sense of the word).
- Using the elements in A , one constructs certain types of *data structures*, whose set is denoted by $D(A)$; it is possible that $D(A)$ contains data structures of various forms, for instance, both strings and double stranded sequences; it is also possible that, in order to construct the set $D(A)$, a *structure* of A is needed, in the form of a relation over A (an order relation, a complementarity relation, a semi-commutativity relation, etc.), of a partition of A (we can distinguish between input symbols, output symbols, state symbols, and so on), etc., but we do not consider in this moment such distinctions.
- For computing, we need certain *resources*; we consider here the case when these resources consist of elements of $D(A)$, with multiplicities, that is, we consider a multiset $W : D(A) \longrightarrow \mathbf{N} \cup \{\infty\}$; a computation is assumed to consume elements of W , which means that only computations which do not need more resources than those provided by W are possible. (Of course, in a more general approach, we can consider a specified set M of resources – raw materials, energy, etc. – and a multiset W defined on this set, $W : M \longrightarrow \mathbf{N} \cup \{\infty\}$.)

- A set \mathcal{C} of *configurations*, which describe the state of the computing machinery at a given moment of time; a comprehensive enough representation of such a state is that of a tuple of elements from $D(A)$ (remember that $D(A)$ can contain elements of various types, from symbols from A to complex constructs based on such symbols).
- Essential for defining the notion of a *computation* is to have a well-defined *transition* relation among configurations, $\Rightarrow_f \subseteq \mathcal{C} \times \mathcal{C}$ (the subscript f indicates the fact that we consider \Rightarrow_f a *free* transition relation, unrestricted by a *control*, as we will mention immediately); as usual, the reflexive and transitive closure of \Rightarrow_f is denoted by \Rightarrow_f^* .

In many – if not all – cases of a practical interest, the transitions are defined starting from a given set of *operations* on data structures: having such operations, one extends them to configurations in order to get transitions.

- For various reasons, a restriction of \Rightarrow_f or of \Rightarrow_f^* can be useful/necessary, that is, subsets $\Rightarrow_{ctr} \subseteq \Rightarrow_f$ or $\Rightarrow_{ctr}^* \subseteq \Rightarrow_f^*$ should be considered; note that the control can be defined directly for \Rightarrow_f^* , not first for \Rightarrow_f and then extended to \Rightarrow_f^* .

In this moment, we can define two sets of computations, as sequences of transitions, namely, *arbitrary computations* (without limiting the resources they use),

$$\mathcal{K} = \{C_1 \Rightarrow_{ctr}^* C_2 \mid C_1, C_2 \in \mathcal{C}\},$$

and *feasible computations* (which do not exceed the resources in W). Let us denote by $w(C_1 \Rightarrow_{ctr}^* C_2)$ the multiset of resources consumed by the computation $C_1 \Rightarrow_{ctr}^* C_2$. Then, we define

$$\mathcal{K}_W = \{C_1 \Rightarrow_{ctr}^* C_2 \mid C_1, C_2 \in \mathcal{C}, w(C_1 \Rightarrow_{ctr}^* C_2) \subseteq W\}.$$

(Of course, the inclusion above holds in the multisets sense.)

- A *stop condition* $\pi : \mathcal{K}_W \rightarrow \{0, 1\}$, in the form of a predicate; $\pi(C_1 \Rightarrow_{ctr}^* C_2) = 1$ means that the (feasible) computation $C_1 \Rightarrow_{ctr}^* C_2$ is completed in an acceptable manner (whatever this means in a specific situation). By $\mathcal{K}_{W,\pi}$ we denote the set of all feasible computations which halt in a correct way.
- An *output space* R , which can be a set containing various types of elements, such as data structures from $D(A)$, natural numbers or relations over natural numbers, Boolean values, and so on.
- An *output mapping*, $\rho : \mathcal{K}_{W,\pi} \rightarrow R$, associating to each feasible and correctly halted computation a *result*.

The previous presentation is still rather general and informal, but sufficient for our purposes.

Let us denote by Γ such a computing framework (also called a *computing system*).

Of course, for pragmatic and also for mathematical reasons, the elements of Γ are supposed to be “as simple as possible”: the alphabet A is either finite or, at most, denumerable (and in a specific computation only a finite set of symbols are used), the relations and the mappings involved (the transition relation \Rightarrow , free or controlled, the stop predicate π , and the output mapping ρ) are supposed to be at least effectively computable, if not “easily” computable. Such conditions will be implicitly fulfilled by all specific models we shall consider in the subsequent chapters.

As above, with a configuration C_1 we can associate both the set of all configurations which can be reached at the end of feasible and correctly completed computations which start in C_1 ,

$$\Gamma(C_1) = \{C_2 \mid C_1 \xrightarrow{\text{ctr}}^* C_2 \in \mathcal{K}_{W,\pi}\},$$

and the set of possible results of these computations,

$$\text{res}_\Gamma(C_1) = \{\rho(C_1 \xrightarrow{\text{ctr}}^* C_2) \mid C_1 \xrightarrow{\text{ctr}}^* C_2 \in \mathcal{K}_{W,\pi}\}.$$

Several properties of computing frameworks (of a specified class) can be defined in this context. We consider here only two of them.

A computing framework Γ is said to be (strongly) *deterministic* if, for each configuration $C_1 \in \mathcal{C}$, there is a unique configuration $C_2 \in \mathcal{C}$ such that $C_1 \xrightarrow{\text{ctr}} C_2$. For some configuration $C_1 \in \mathcal{C}$, a computing framework Γ is said to be C_1 -*deterministic*, if, for each $C_2 \in \mathcal{C}$ such that $C_1 \xrightarrow{\text{ctr}}^* C_2 \in \mathcal{K}_W$ (in particular, for $C_2 = C_1$), there is a unique configuration $C_3 \in \mathcal{C}$ such that $C_1 \xrightarrow{\text{ctr}}^* C_2 \xrightarrow{\text{ctr}} C_3 \in \mathcal{K}_W$ (all computations starting from C_1 proceed deterministically, with one possible choice at each step). Note that we take into account only feasible computations.

A computing framework Γ_u is said to be *universal* if, for each computing framework Γ , there is $\text{code}(\Gamma) \in D(A)$ such that for each configuration C_1 of Γ we can get a configuration C'_1 of Γ_u which “contains” (in a specified sense) both C_1 and $\text{code}(\Gamma)$ and we have

$$\text{res}_{\Gamma_u}(C'_1) = \text{res}_\Gamma(C_1).$$

That is to say, if we introduce the code of Γ in an initial configuration of Γ_u , together with an initial configuration of Γ , then the universal computing framework Γ_u behaves exactly as the particular one, Γ , when starting from that initial configuration.

The previous setup is so general that it covers many particular cases, so it is a good reference framework, but it is *too general* in order to obtain

relevant mathematical results. For specific cases, we have to particularize also the items considered above.

The overall strategy is the following: we examine the reality¹, looking for the basic ingredients of our computing framework, the data structures and the operations on them; also basic, but somewhat of a second level, are the possible controls on the operations. (Here, *possible* means, in general, *realistic*, *plausible*, sometimes only *desirable and not proved yet to be impossible*, and not necessarily feasible, implementable in the present day laboratories.)

After having these basic ingredients, data structures, operations, controls, we pass to the next stage, of integrating these items in a computing machinery (of a theoretical type): we define the transitions, the stop conditions, hence the computations and, after that, the results of computations. In short, we have to define the *computing models* we look for.

Then, a third phase is to be followed, of a mathematical type, aiming to answer questions of the following forms: How powerful the defined models are (in comparison with classic models investigated in theoretical computer science)? Are there universality results? (A positive answer ensures the theoretical possibility of programmability.) How useful, from a computational point of view the obtained models are? This means mainly time complexity of solving problems or of computing functions in the new framework, in the hope that difficult problems/functions can be solved/computed significantly faster than with a classic computing device.

Of course, the fourth phase should be concerned with the implementation of a computing device, the actual building of a “computer” based on the above specified model. This closes the circle, returning to the reality, and this is definitely the validating step from a practical point of view. However, here we will not go that far, because the implementation of the models we consider is out of the scope of the present book. Currently, both Quantum and Molecular Computing are mainly developed at the level of the first three phases and the reasons are obvious: the theory acts in an ideal world where all wishes which are logically consistent (and not contradicted by the state-of-the-art of biochemistry and physics) can be assumed as being fulfilled, everything which is plausible can also be assumed as being possible, which is not exactly what the technology provides to us. There also is a great attraction towards theory and theoretical results, as a manifestation of the need for knowledge. Knowledge *is* in some sense a goal *per se*: How nature “computes”? is a fascinating question (even if, always when speaking about nature, we keep the quotation marks around the term “computation”).

¹What is the reality is another story. In fact, we look for the reality as it appears in books – biochemical books in the case of Molecular Computing, books of physics in the case of Quantum Computing. Otherwise stated, we always have in mind a representation/conceptualization of “reality” and we do not care too much how adequate this representation is, we simply rely on the current state-of-the-art of the mentioned scientific fields.

1.4 Chomsky Grammars

We consider now one of the most important classes of computing devices in theoretical computer science, the rewriting systems, with their particular form of Chomsky grammars. We do not systematically particularize the items of a computing framework as introduced in the previous section, but we only mention that we work with *strings* as data structures, while the operation we use is *rewriting* (replacing a short substring of a string by another short string).

A *rewriting system* is a pair $\gamma = (V, P)$, where V is an alphabet and P is a finite subset of $V^* \times V^*$; the elements (u, v) of P are written in the form $u \rightarrow v$ and are called *rewriting rules/productions* (or simply *rules* or *productions*). For $x, y \in V^*$ we write $x \Rightarrow_\gamma y$ if $x = x_1ux_2, y = x_1vx_2$, for some $u \rightarrow v \in P$ and $x_1, x_2 \in V^*$. If the rewriting system γ is understood, then we write \Rightarrow instead of \Rightarrow_γ . The reflexive and transitive closure of \Rightarrow is denoted by \Rightarrow^* .

If an *axiom* is added to a rewriting system and all rules $u \rightarrow v$ have $u \neq \lambda$, then we obtain the notion of a *pure grammar*. For a pure grammar $G = (V, w, P)$, where $w \in V^*$ is the axiom, we define the *language generated* by G by

$$L(G) = \{x \in V^* \mid w \Rightarrow^* x\}.$$

A *Chomsky grammar* is a quadruple $G = (N, T, S, P)$, where N, T are disjoint alphabets, $S \in N$, and P is a finite subset of $(N \cup T)^*N(N \cup T) \times (N \cup T)^*$.

The alphabet N is called the *nonterminal alphabet*, T is the *terminal alphabet*, S is the *axiom*, and P is the set of *production rules* of G . The rules (we also say *productions*) (u, v) of P are written in the form $u \rightarrow v$. Note that $|u|_N \geq 1$.

For $x, y \in (N \cup T)^*$ we write

$$\begin{aligned} x \Rightarrow_G y \quad \text{iff} \quad & x = x_1ux_2, y = x_1vx_2, \\ & \text{for some } x_1, x_2 \in (N \cup T)^* \text{ and } u \rightarrow v \in P. \end{aligned}$$

One says that x *directly derives* y (with respect to G). Each string $w \in (N \cup T)^*$ such that $S \Rightarrow_G^* w$ is called a *sentential form*.

The language generated by G , denoted by $L(G)$, is defined by

$$L(G) = \{x \in T^* \mid S \Rightarrow^* x\}.$$

(Note that the stop condition is the condition to have no nonterminal symbol present in the obtained string and that the result of a computation precisely consists of this last string, composed of only terminal symbols.) Two grammars G_1, G_2 are called *equivalent* if $L(G_1) - \{\lambda\} = L(G_2) - \{\lambda\}$ (the two languages coincide modulo the empty string).

In general, in this book we consider two generative mechanisms equivalent if they generate the same language when we ignore the empty string (the empty string has no “real life” counterpart, so in many cases we ignore it).

According to the form of their rules, the Chomsky grammars are classified as follows. A grammar $G = (N, T, S, P)$ is called:

- *monotonous/context-sensitive*, if for all $u \rightarrow v \in P$ we have $|u| \leq |v|$;
- *context-free*, if each production $u \rightarrow v \in P$ has $u \in N$;
- *linear*, if each rule $u \rightarrow v \in P$ has $u \in N$ and $v \in T^* \cup T^*NT^*$;
- *regular*, if each rule $u \rightarrow v \in P$ has $u \in N$ and $v \in T \cup TN \cup \{\lambda\}$.

The arbitrary, monotonous, context-free, and regular grammars are also said to be of *type 0*, *type 1*, *type 2*, and *type 3*, respectively.

We denote by *CE*, *CS*, *CF*, *LIN*, and *REG* the families of languages generated by arbitrary, context-sensitive, context-free, linear, and regular grammars, respectively (*CE* stands for *computably enumerable*; we keep here this “classic” terminology, although we agree with [273] that *computably enumerable* is a more adequate sintagma). By *FIN* we denote the family of finite languages.

The following strict inclusions hold:

$$FIN \subset REG \subset LIN \subset CF \subset CS \subset CE.$$

This is the *Chomsky hierarchy*, the constant reference for the investigations in the following chapters.

As the context-free grammars are not powerful enough for covering most of the important syntactic constructions in natural and artificial languages, while the context-sensitive grammars are too powerful (for instance, the family *CS* has many negative decidability properties and the derivations in a non-context-free grammar cannot be described by a derivation tree), it is of interest to increase the power of context-free grammars by controlling the use of their rules (by imposing a control on the computations in a context-free grammar, as we have considered in the general computing framework discussed in Section 1.3.) This leads to considering *regulated* context-free grammars. We do not enter here into details (the reader can consult [75] as well as the corresponding chapter from [258]), although several of the controls used in formal language theory will also suggest useful ways of controlling the computations in various models defined in the DNA Computing area.

Another chapter of formal language theory which suggests fruitful ideas to DNA Computing is grammar systems theory, which deals with systems consisting of several grammars which work together in a well specified manner and generate one single language. There are two main classes of grammar systems, the sequential ones (introduced in [68] under the name of *cooperating distributed grammar systems*), where the components work in turns, on a

common sentential form, and the parallel systems (introduced in [229] under the name of *parallel communicating grammar systems*), whose components work synchronously, on their own sentential forms, and communicate by request or by command. In both cases, a significant increase of the power of context-free grammars is obtained. Details can be found in [69] and in the corresponding chapter from [258].

For certain classes of grammars in the Chomsky hierarchy it is possible to work with grammars of a specified form without losing the generative power. We mention here only two normal forms of this type, for grammars characterizing CE , because they will be useful later.

Theorem 1.1 (Kuroda normal form) *For every type-0 grammar G , an equivalent grammar $G' = (N, T, S, P)$ can be effectively constructed, with the rules in P of the forms $A \rightarrow BC, A \rightarrow a, A \rightarrow \lambda, AB \rightarrow CD$, for $A, B, C, D \in N$ and $a \in T$.*

A similar result holds true for monotonous grammars, where rules of the form $A \rightarrow \lambda$ are no longer allowed.

Theorem 1.2 (Geffert normal forms) (i) *Each computably enumerable language can be generated by a grammar $G = (N, T, S, P)$ with $N = \{S, A, B, C\}$ and the rules in P of the forms $S \rightarrow uSv, S \rightarrow x$, with $u, v, x \in (T \cup \{A, B, C\})^*$, and only one non-context-free rule, $ABC \rightarrow \lambda$.*

(ii) *Each computably enumerable language can be generated by a grammar $G = (N, T, S, P)$ with $N = \{S, A, B, C, D\}$ and the rules in P of the forms $S \rightarrow uSv, S \rightarrow x$, with $u, v, x \in (T \cup \{A, B, C, D\})^*$, and only two non-context-free rules, $AB \rightarrow \lambda, CD \rightarrow \lambda$.*

1.5 Lindenmayer Systems

A variant of rewriting systems, related to Chomsky grammars, but with the derivation steps performed in a maximally parallel manner (see precise definitions below), are the Lindenmayer systems, introduced with biological motivations. We present here only a few elementary notions.

Basically, a 0L (0-interactions Lindenmayer) system is a context-free pure grammar with parallel derivations: $G = (V, w, P)$, where V is an alphabet, $w \in V^*$ (axiom), and P is a finite set of rules of the form $a \rightarrow v$ with $a \in V, v \in V^*$, such that for each $a \in V$ there is at least one rule $a \rightarrow v$ in P (we say that P is *complete*). For $w_1, w_2 \in V^*$ we write $w_1 \Rightarrow w_2$ if $w_1 = a_1 \dots a_n, w_2 = v_1 \dots v_n$, for $a_i \rightarrow v_i \in P, 1 \leq i \leq n$. The generated language is $L(G) = \{x \in V^* \mid w \Rightarrow^* x\}$.

If for each rule $a \rightarrow v \in P$ we have $v \neq \lambda$, then we say that G is *propagating* (non-erasing); if for each $a \in V$ there is only one rule $a \rightarrow v$ in P , then G is said to be *deterministic*. If we distinguish a subset T of V and

we define the generated language by $L(G) = \{x \in T^* \mid w \Rightarrow^* x\}$, then we say that G is *extended*. The family of languages generated by 0L systems is denoted by $0L$; we add the letters P, D, E in front of $0L$ if propagating, deterministic, or extended 0L systems are used, respectively.

A *tabled* 0L system, abbreviated $T0L$, is a system $G = (V, w, P_1, \dots, P_n)$, such that each triple (V, w, P_i) , $1 \leq i \leq n$, is a 0L system; each P_i is called a *table*, $1 \leq i \leq n$. The generated language is defined by

$$\begin{aligned} L(G) = \{x \in V^* \mid & w \Rightarrow_{P_{j_1}} w_1 \Rightarrow_{P_{j_2}} \dots \Rightarrow_{P_{j_m}} w_m = x, \\ & m \geq 0, 1 \leq j_i \leq n, 1 \leq i \leq m\}. \end{aligned}$$

(Each derivation step is performed by the rules of the same table.)

A $T0L$ system is deterministic when each of its tables is deterministic. The propagating and the extended features are defined in the usual way.

The family of languages generated by $T0L$ systems is denoted by $T0L$; the $ET0L$, $EDT0L$, etc. families are obtained in the same way as $E0L$, $ED0L$, etc.

The $D0L$ family is incomparable with FIN , REG , LIN , CF , whereas $E0L$ strictly includes the CF family; $ET0L$ is the largest family of Lindenmayer languages without interactions (context-free) and it is strictly included in CS .

1.6 Automata and Transducers

Automata are language defining devices which work in the direction opposite to grammars: they start from the strings over a given alphabet and *analyze* them (we also say *recognize*), telling us whether or not any input string belongs to a specified language.

The five basic families of languages in the Chomsky hierarchy, REG , LIN , CF , CS , CE , are also characterized by recognizing automata. These automata are: the finite automaton, the one-turn pushdown automaton, the pushdown automaton, the linear-bounded automaton, and the Turing machine, respectively. We present here only the finite automaton and the Turing machine, which mark, in some sense, the two poles of computability.

A (non-deterministic) *finite automaton* is a construct

$$M = (K, V, s_0, F, \delta),$$

where K and V are disjoint alphabets, $s_0 \in K$, $F \subseteq K$, and $\delta : K \times V \rightarrow \mathcal{P}(K)$; K is the set of states, V is the alphabet of the automaton, s_0 is the initial state, F is the set of final states, and δ is the transition mapping. If $\text{card}(\delta(s, a)) \leq 1$ for all $s \in K, a \in V$, then we say that the automaton is *deterministic*. A relation \vdash is defined as follows on the set $K \times V^*$: for $s, s' \in K$, $a \in V$, $x \in V^*$, we write $(s, ax) \vdash (s', x)$ if $s' \in \delta(s, a)$; by definition,

$(s, \lambda) \vdash (s, \lambda)$. If \vdash^* is the reflexive and transitive closure of the relation \vdash , then the language recognized by the automaton M is defined by

$$L(M) = \{x \in V^* \mid (s_0, x) \vdash^* (s, \lambda), s \in F\}.$$

Deterministic and non-deterministic finite automata characterize the same family of languages, namely *REG*. The power of finite automata is not increased if we also allow *λ -transitions*, that is, if δ is defined on $K \times (V \cup \{\lambda\})$ (the automaton can also change its state when reading no symbol from its tape), or when the input string is scanned in a two-way manner, going along it to the right or to the left, without changing its symbols.

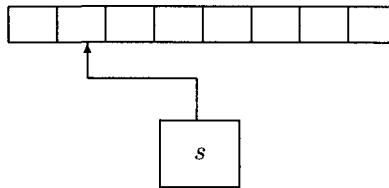


Figure 1.1: A finite automaton.

An important related notion is that of a *sequential transducer*; we shall use the abbreviation *gsm*, from “generalized sequential machine”. Such a device is a system $g = (K, V_1, V_2, s_0, F, \delta)$, where K, s_0, F are the same as in a finite automaton, V_1, V_2 are alphabets (the input and the output alphabet, respectively), and $\delta : K \times V_1 \longrightarrow \mathcal{P}_f(V_2^* \times K)$. If $\delta(s, a) \subseteq V_2^+ \times K$ for all $s \in K, a \in V_1$, then g is said to be *λ -free*. If $\text{card}(\delta(s, a)) \leq 1$ for each $s \in K, a \in V_1$, then g is said to be *deterministic*. For $s, s' \in K, a \in V_1, y \in V_1^*, x, z \in V_2^*$, we write $(x, s, ay) \vdash (xz, s', y)$ if $(z, s') \in \delta(s, a)$. Then, for $w \in V_1^*$, we define

$$g(w) = \{z \in V_2^* \mid (\lambda, s_0, w) \vdash^* (z, s, \lambda), s \in F\}.$$

The mapping g is extended in the natural way to languages over V_1 .

If $V_1 \subseteq V_2$, then g can be iterated. We denote: $g^0(L) = L, g^{i+1}(L) = g(g^i(L)), i \geq 0$, and $g^* = \bigcup_{i \geq 0} g^i(L)$.

We can imagine a finite automaton as in Figure 1.1, where we distinguish the input tape, in whose cells we write the symbols of the input string, the read head, which scans the tape from the left to the right, and the memory, able to hold a state from a finite set of states. In the same way, a gsm is a device as in Figure 1.2, where we also have an output tape, where the write head can write the string obtained by translating the input string.

Sometimes it is useful to present the transition mapping of finite automata and of gsm's as a set of rewriting rules: we write $sa \rightarrow as'$ instead of $s' \in$

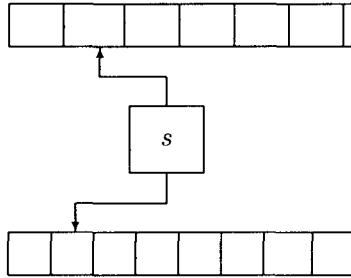


Figure 1.2: A sequential transducer.

$\delta(s, a)$ in the case of finite automata and $sa \rightarrow zs'$ instead of $(z, s') \in \delta(s, a)$ in the case of gsm's. Then the relations \vdash, \vdash^* are exactly the same as \implies, \implies^* in the rewriting system obtained in this way and, for a gsm g and a language $L \in V_1^*$, we get

$$g(L) = \{z \in V_2^* \mid s_0 w \implies^* zs, w \in L, s \in F\}.$$

For finite automata we have a special case: $L(M) = \{x \in V^* \mid s_0 x \implies^* xs, s \in F\}$.

A *Turing machine* is a construct

$$M = (K, V, T, B, s_0, F, \delta),$$

where K, V are disjoint alphabets (the set of states and the tape alphabet), $T \subseteq V$ (the input alphabet), $B \in V - T$ (the blank symbol), $s_0 \in K$ (the initial state), $F \subseteq K$ (the set of final states), and δ is a partial mapping from $K \times V$ to $\mathcal{P}(K \times V \times \{L, R\})$ (the move mapping; if $(s', b, d) \in \delta(s, a)$, for $s, s' \in K, a, b \in V$, and $d \in \{L, R\}$, then the machine reads the symbol a in state s and passes to state s' , replaces a with b , and moves the read-write head to the left when $d = L$ and to the right when $d = R$). If $\text{card}(\delta(s, a)) \leq 1$ for all $s \in K, a \in V$, then M is said to be *deterministic*.

An *instantaneous description* of a Turing machine as above is a string xsy , where $x \in V^*, y \in V^*(V - \{B\}) \cup \{\lambda\}$, and $s \in K$. In this way we identify the contents of the tape, the state, and the position of the read-write head: it scans the first symbol of y . Observe that the blank symbol may appear in x, y , but not in the last position of y ; both x and y may be empty. We denote by ID_M the set of all instantaneous descriptions of M .

On the set ID_M one defines the *direct transition* relation \vdash_M as follows:

$$\begin{aligned} xsy &\vdash_M xbs'y \text{ iff } (s', b, R) \in \delta(s, a), \\ xs &\vdash_M xbs' \text{ iff } (s', b, R) \in \delta(s, B), \end{aligned}$$

$$\begin{aligned} xcsay \vdash_M xs'cby &\text{ iff } (s', b, L) \in \delta(s, a), \\ xcs \vdash_M xs'cb &\text{ iff } (s', b, L) \in \delta(s, B), \end{aligned}$$

where $x, y \in V^*, a, b, c \in V, s, s' \in K$.

The language recognized by a Turing machine M is defined by

$$L(M) = \{w \in T^* \mid s_0 w \vdash_M^* xsy \text{ for some } s \in F, x, y \in V^*\}.$$

(This is the set of all strings such that the machine reaches a final state when starting to work in the initial state, scanning the first symbol of the input string.)

It is also customary to define the language accepted by a Turing machine as consisting of the input strings $w \in T^*$ such that the machine, starting from the configuration $s_0 w$, reaches a configuration where no further move is possible (we say that the machine *halts*). The two modes of defining the language $L(M)$ are equivalent.

Graphically, a Turing machine can be represented as a finite automaton (Figure 1.1). The difference between a finite automaton and a Turing machine is visible only in their functioning: the Turing machine can move its head in both directions and can rewrite the scanned symbol, possibly erasing it (replacing it with the blank symbol).

Both the deterministic and the non-deterministic Turing machines characterize the family of computably enumerable languages.

A Turing machine can be also viewed as a mapping-defining device, not only as a mechanism defining a language. Specifically, consider a Turing machine $M = (K, V, T, B, s_0, F, \delta)$. If $\beta \in ID_M$ such that $\beta = x_1 s a x_2$ and $\delta(s, a) = \emptyset$, then we write $\beta \downarrow$ (we say that β is a halting configuration). We define the mapping $F_M : ID_M \longrightarrow \mathcal{P}(ID_M)$ by $F_M(\alpha) = \{\beta \in ID_M \mid \alpha \vdash_M^* \beta \text{ and } \beta \downarrow\}$. If M is deterministic, then F_M is a mapping from ID_M to ID_M .

Given a mapping $f : U_1^* \longrightarrow U_2^*$, where U_1, U_2 are arbitrary alphabets, we say that f is *computed* by a deterministic Turing machine M if there are two (computable) mappings C and D (of *coding* and *decoding*),

$$C : U_1^* \longrightarrow ID_M, \quad D : ID_M \longrightarrow U_2^*,$$

such that

$$D(F_M(C(x))) = f(x).$$

In Section 1.8, when discussing and presenting universal Turing machines, we shall use this interpretation of Turing machines (as well as the termination of a computation by halting configurations, not by using final states).

1.7 Characterizations of Computably Enumerable Languages

We have mentioned that our constant framework here is Chomsky hierarchy and the two poles of computability we refer to are the regular languages

(corresponding to the power of finite automata) and the computably enumerable languages (characterized by Turing machines). According to the Church–Turing Thesis, the power of Turing machines is the highest level of algorithmic computability. For this reason (and because we do not have good universality results at the level of finite automata – see also Section 1.8), when looking for new computability models it is desirable to obtain models equal in power to Turing machines. In order to prove such a result, the characterizations of the computably enumerable languages (for instance, by grammars in the normal forms discussed in Section 1.4) will be very useful. When such a direct simulation is not possible/visible, representation results available for computably enumerable languages can be of a great help. Some of these results are quite non-intuitive, which makes their consequences rather interesting. We present here without proofs some theorems of this type, where we start from “small” subfamilies of CE and, by using powerful operations (such as intersection, quotients, etc.), we cover the whole family CE .

Theorem 1.3 *Each language $L \in CE, L \subseteq T^*$, can be written in the form $L = g^*(a_0) \cap T^*$, where $g = (K, V, V, s_0, F, P)$ is a gsm and $a_0 \in V$.*

A quite powerful (and useful for some of the next chapters) representation of computably enumerable languages starts from *equality sets* of morphisms.

For two morphisms $h_1, h_2 : V^* \rightarrow U^*$, the set

$$EQ(h_1, h_2) = \{w \in V^* \mid h_1(w) = h_2(w)\}$$

is called the *equality set* of h_1, h_2 .

Theorem 1.4 *Every computably enumerable language $L \subseteq T^*$ can be written in the form $L = pr_T(EQ(h_1, h_2) \cap R)$, where h_1, h_2 are two morphisms, R is a regular language, and pr_T is the projection associated with the alphabet T .*

A variant of this result, useful in Section 2.8, is the following one.

Theorem 1.5 *For each computably enumerable language $L \subseteq T^*$, there exist two λ -free morphisms h_1, h_2 , a regular language R , and a projection pr_T such that $L = pr_T(h_1(EQ(h_1, h_2)) \cap R)$.*

Note the difference between the representations in Theorems 1.4 and 1.5: in the first case the language L is obtained as a projection of the intersection of the equality set with a regular language, whereas in the second case the language L is the projection of the intersection of a regular language with the image of the equality set under one of the morphisms defining the equality set.

1.8 Universal Turing Machines and Type-0 Grammars

A computer is a *programmable* machine, able to execute any program it receives. From a theoretical point of view, this corresponds to the notion of a *universal Turing machine*, in general, to the notion of universality as introduced in Section 1.3.

Consider an alphabet T and a Turing machine $M = (K, V, T, B, s_0, F, \delta)$. As we have seen above, M starts working with a string w written on its tape and reaches or not a final state (and then halts), depending on whether or not $w \in L(M)$. A Turing machine can be also codified as a string of symbols over a suitable alphabet. Denote such a string by $\text{code}(M)$. Imagine a Turing machine M_u which starts working from a string which contains both $w \in T^*$ and $\text{code}(M)$ for a given Turing machine M , and stops in a final state if and only if $w \in L(M)$.

In principle, the construction of M_u is simple. M_u only has to simulate the way of working of Turing machines, and this is clearly possible: look for a transition, as defined by the mapping δ , depending on the current state and the current position of the read-write head (this information is contained in the instantaneous descriptions of the particular machine); whenever several choices are possible, make copies of the current instantaneous description and branch the machine evolution; if two copies of the same instantaneous description appear, then delete one of them; if at least one of the evolution variants leads to an accepting configuration, stop and accept the input string, otherwise continue.

Such a machine M_u is called *universal*. It can simulate any given Turing machine, providing that a code of a particular one is written on the tape of the universal one, together with a string to be dealt with by the particular machine.

The parallelism with a computer, as we know the computers in their general form, is clear: the code of a Turing machine is its *program*, the strings to be recognized are the input data, the universal Turing machine is the computer itself (its operating system).

Let us stress here an important distinction, that between *computational completeness* and *universality*. Given a class \mathcal{C} of computability models, we say that \mathcal{C} is *computationally complete* if the devices in \mathcal{C} can characterize the power of Turing machines. This means that given a Turing machine M we can find an element C in \mathcal{C} such that C is equivalent with M . Thus, completeness refers to the capacity of covering the level of computability (in grammatical terms, this means to generate all computably enumerable languages). Universality is an internal property of \mathcal{C} and it means the existence of a fixed element of \mathcal{C} which is able to simulate any given element of \mathcal{C} , in the way described above for Turing machines.

Of course, we can define the completeness in a relative way, not referring to

the whole class of Turing machines but to a subclass of them. For instance, we can look for context-free completeness (the possibility of generating all context-free languages). Accordingly, we can look for universal elements in classes of computing devices which are computationally complete for smaller families of languages than the family of computably enumerable languages. However, important for any theory which attempts to provide general models of computing are the completeness and universality with respect to Turing machines, and this will be the level we shall consider in this book.

The idea of a universal Turing machine was introduced by A. Turing himself, who has also produced such a machine [289]. Many universal Turing machines are now available in the literature, mainly for the case when Turing machines are considered as devices which compute mappings. In such a framework, we say that a Turing machine is *universal* if it computes a universal partial computable function (modulo the coding-decoding “interface” mentioned in Section 1.6). Similarly, a Turing machine M_1 *simulates* a Turing machine M_2 if there are two coding-decoding mappings

$$C : ID_{M_2} \longrightarrow ID_{M_1}, \quad D : ID_{M_1} \longrightarrow ID_{M_2},$$

such that for each $\alpha \in ID_{M_2}$ we have

$$D(F_{M_1}(C(\alpha))) = F_{M_2}(\alpha).$$

The (descriptive, static) complexity of a Turing machine can be evaluated from various points of view: the number of *states*, the number of *tape symbols* (the blank symbol included), or the number of *moves* (quintuples (s, a, b, d, s') such that $(s', b, d) \in \delta(s, a)$).

We denote by $UTM(m, n)$ the class of universal deterministic Turing machines with m states and n symbols (because we must have halting configurations, there can exist at most $m \cdot n - 1$ moves).

Small universal Turing machines were produced already in [267] (with two states) and [194] (with seven states and four symbols). The up-to-date results in this area are summarized in [252] (we also include the improvement from [253]):

Theorem 1.6 (i) *The classes $UTM(2, 3)$, $UTM(3, 2)$ are empty.*

(ii) *The following classes are non-empty: $UTM(22, 2)$, $UTM(10, 3)$, $UTM(7, 4)$, $UTM(5, 5)$, $UTM(4, 6)$, $UTM(3, 10)$, $UTM(2, 18)$.*

For the remaining 49 classes $UTM(m, n)$ the question is open.

In most of the constructions on which the proofs in the subsequent chapters are based, we shall start from a Chomsky type-0 grammar.

Given a Turing machine M we can effectively construct a type-0 grammar G such that $L(M) = L(G)$. (Similarly, we can produce a type-0 grammar G such that G computes, in a natural way and using appropriate coding-decoding mappings, the same mapping F_M as M . So, a grammar can be

considered a function computing device, not only a language generating mechanism.)

The idea is very simple. Take a Turing machine $M = (K, V, T, B, s_0, F, \delta)$ and construct a non-restricted Chomsky grammar G working as follows: starting from its axiom, G non-deterministically generates a string w over V , then it makes a copy of w (of course, the two copies of w are separated by a suitable marker; further markers, scanners and other auxiliary symbols are allowed, because they can be erased when they are no longer necessary). On one of the copies of w , G can simulate the work of M , choosing non-deterministically a computation as defined by δ ; if a final state is reached, then the witness copy of w is preserved and everything else is erased.

Applying a construction of this type to a universal Turing machine M_u , we obtain a *universal type-0 Chomsky grammar* G_u , a grammar which is universal in the following sense: the language generated by G_u consists of strings of the form, say, $w\#code(M)$, such that $w \in L(M)$. (We can call the language $\{w\#code(M) \mid w \in L(M)\}$ itself universal, and thus any grammar generating this language is universal.) A “more grammatical” notion of universality can be the following.

A triple $G = (N, T, P)$, where the components N, T, P are as in a usual Chomsky grammar, is called a *grammar scheme*. For a string $w \in (N \cup T)^*$ we define the language $L(G, w) = \{x \in T^* \mid w \Rightarrow^* x\}$, the derivation being performed according to the productions in P .

A *universal type-0 grammar* is a grammar scheme $G_u = (N_u, T_u, P_u)$, where N_u, T_u are disjoint alphabets, and P_u is a finite set of rewriting rules over $N_u \cup T_u$, with the property that for any type-0 grammar $G = (N, T, S, P)$ there is a string $w(G)$ such that $L(G_u, w(G)) = L(G)$.

Therefore, the universal grammar simulates any given grammar, provided that a code $w(G)$ of the given grammar is taken as a starting string of the universal one.

There are universal type-0 grammars in the sense specified above. The reader can find a complete construction in [54] (it is also presented in [52] and [224]). That universal grammar codifies in terms of type-0 grammars the derivation process used by a grammar: choose a rule, remove an occurrence of its left hand member and introduce instead of it an occurrence of its right hand member, check whether or not a terminal string is obtained.

A natural question here, also important for molecular computing, is whether or not universality results hold also for other classes of automata and grammars than Turing machines and type-0 grammars, in particular for finite automata.

If the question is understood in the strict sense, then the answer is negative for finite automata: no finite automaton can be universal for the class of all finite automata. The main reason is the fact that one cannot encode the way of using a finite automaton in terms of a finite automaton (we have to remember symbols in the input string without marking them, and this

cannot be done with a finite set of states).

However, universal finite automata in a certain restricted sense can be found; in some extent, this is a matter of definition of universality. For instance, in [56] one proves that given a finite set of finite automata, in a certain completion of this set one can find an automaton which is universal for the considered set. Similarly, a universal finite automaton is constructed in [224] for the finite set of automata with a bounded number of states, providing that the starting string (containing both the string to be recognized and the code of a particular finite automaton) is of a complex form (of a non-context-free type). We do not enter here into details, because, although the topic is of a central interest for our book, the solutions in [56] and [224] are not “good enough” from a practical point of view (they are too particular).

1.9 Complexity

Let M be a Turing machine. For each input of length n , if M makes at most $t(n)$ moves before it stops, then we say that M runs in time t or M has *time complexity* t . If M uses at most $s(n)$ tape cells in the above computation, then we say that M uses s space, or has *space complexity* s . Accordingly, we can define the following classes of languages:

- the class of languages accepted by deterministic Turing machines in time $O(t)$,² DTIME[t],
- the class of languages accepted by non-deterministic Turing machines in time $O(t)$, NTIME[t];
- the class of languages accepted by deterministic Turing machines in space $O(t)$, DSPACE[t];
- the class of languages accepted by non-deterministic Turing machines in space $O(t)$, NSPACE[t];
- the class of languages accepted by deterministic Turing machines in polynomial time, $P = \bigcup_c \text{DTIME}(n^c)$;
- the class of languages accepted by non-deterministic Turing machines in polynomial time, $\text{NP} = \bigcup_c \text{NTIME}(n^c)$;
- the class of languages accepted by deterministic Turing machines in polynomial space, $\text{PSPACE} = \bigcup_c \text{DSPACE}(n^c)$;
- the class of languages accepted by non-deterministic Turing machines in polynomial space, $\text{NPSPACE} = \bigcup_c \text{NTIME}(n^c)$.

²For two real-valued functions f, g defined on non-negative integers, $f(n) = O(g(n))$ if there exist two constants $c, N > 0$ such that $|f(n)| \leq c \cdot |g(n)|$, for all $n \geq N$.

The following relations hold true:

$$P \subseteq NP \subseteq PSPACE = NPSPACE.$$

An important *open problem* is to check which of the above inclusions is proper.

Let's come back to the notion of non-deterministic Turing machine $M = (K, V, T, B, s_0, F, \delta)$. Without losing generality we will assume that for every $s \in K, a \in V, \text{card}(\delta(s, a)) \leq 2$ and for every input x the machine M executes the same number of steps. In this way, the set of all possible computations of M on an input x can be described by a binary tree: the nodes of the tree are configurations of the machine, the root is the initial configuration, and for every node c , its children are those configurations which can be reached from c in one move according to the transition function of the machine. Leaves of the tree are final configurations, some of which may accept, others reject. An accepting computation is a path starting at the root and finishing in an accepting leaf. According to the definition of acceptance, an input is accepted if and only if there is at least one accepting leaf in its tree of computation. At each internal node, the selection of the continuation of the path is done in a non-deterministic manner. According to our convention, the computation tree on every input which is accepted is finite.

Here is a simple example of a non-deterministic algorithm solving the *satisfiability problem* SAT: given a Boolean formula with n variables, decide whether F is satisfiable (i.e., there is an assignment mapping variables to $\{0, 1\}$ which satisfies F). We will assume a fixed encoding of Boolean formulas, $e(F)$.

```

input e(F)
check that e(F) encodes a correct Boolean formula
for each variable x occurring in F do
    choose in a non-deterministic manner
        F = F|_{x=0} or
        F = F|_{x=1}
    simplify the resulting formula without variables
    if the result is 1, then accept and halt
end

```

In contrast with a deterministic algorithm for solving SAT, that might need to explore *all* possible assignments, the non-deterministic algorithm has to guess “correctly” just one assignment (guessing the correct solution) and then to verify its correctness. So, SAT is in NP, but it is not known whether SAT is or is not in P. In fact SAT has a central position: if SAT would be in P, then $P = NP$.

1.10 Bibliographical Notes

There are many monographs in automata, formal language theory and complexity theory. Several titles can be found in the bibliographical list which concludes the book: [4], [11], [69], [75], [113], [123], [198], [256], [258], [259], [260], [273]. We recommend them to the reader interested in further details.

Results similar to Theorem [207] were given in [207], [255], [299].

Characterizations of computably enumerable languages starting from equality sets of morphisms were given in [261], [72]; complete proofs can be found in [262]. The variant of Theorem 1.4 given in Theorem 1.5 is proved in [149].

Universal Turing machines can be found in [194], [267], and, mainly, in [252].

Chapter 2

DNA Computing

The aim of this chapter is to have a glimpse in the fast emerging area of DNA Computing. We start by briefly presenting the structure of DNA molecule and the operations which can be carried (mainly *in vitro*) on it, then we present a series of experiments already done in this area (starting with the famous Adleman's experiment) or only proposed to be done, we derive from this practical approach a series of theoretical developments, and we end with an overview of the most developed chapter of DNA Computing, the theory of splicing systems.

2.1 The Structure of DNA

The main data structure we work with is the double stranded sequence, with the paired elements related by a complementarity relation. After introducing this data structure, we discuss its intrinsic (computational) power, by relating it to computably enumerable languages via a characterization of *CE* by means of the so-called *twin-shuffle languages*. We anticipate the very important conclusion of this connection: the DNA (structure) has a sort of in-built computational completeness, it is a “blue print” of computability at the level of Turing machines.

Our approach to the structure of DNA is directed to our goals, so we consider the DNA molecule in a rather simplified manner, ignoring a lot of biochemical information not necessary for the subsequent sections. We follow the style of Chapter 1 of [224], where more details can be found.

DNA is an abbreviation for *Deoxyribonucleic Acid*. A DNA molecule is a polymer consisting of two sequences of monomers, which are also called deoxyribonucleotides, in short, nucleotides. Each nucleotide consists of three components: a *sugar*, a *phosphate group*, and a *nitrogenous base*.

The sugar has five carbon atoms, numbered from 1' to 5'. The phosphate group is attached to the 5' carbon, and the base is attached to the 1' carbon.

To the 3' carbon there is attached a hydroxyl (OH) group. We have mentioned these biochemical technicalities because they are important in giving a directionality to the two strands of the DNA molecule.

Different nucleotides differ only by their bases, which are of two types: *purines* and *pyrimidines*. There are two purines: *adenine* and *guanine*, abbreviated A and G, and two pyrimidines: *cytosine* and *thymine*, abbreviated C and T. Since the nucleotides differ only by their bases, they are simply referred to as A, G, C, and T.

We are not interested here in the structure of a nucleotide, but in the way the nucleotides are linked together. This induces the very structure of DNA molecules.

Any two nucleotides can link together by a covalent bond; thus, several nucleotides can join and form a sequence in the same way as several symbols form a string. As for symbols in a string, there is no restriction on the nucleotides in the sequence. We say that we get a single stranded sequence, in order to distinguish the obtained polymer from that obtained by binding together two such sequences by means of hydrogen bonds between corresponding nucleotides.

There are three fundamental facts in building double stranded sequences:

- The hydrogen bonds can be established only between A and T, as well as between C and G. One says that the nucleotides in the pairs (A, T) and (C, G) are *complementary*.
- The single stranded sequences of nucleotides have a directionality, given by the carbons used by the covalent bonds. The first and the last nucleotide establish only one such bond, hence a place for a further bond is available in these positions. Because these places refer to the 3' and the 5' carbons, one end of the sequence is marked with 3' and the other one by 5'.
- The hydrogen bonds, based on complementarity, can bring together single stranded sequences of nucleotides only if they are of opposite directionality.

The above mentioned complementarity is called the *Watson-Crick complementarity*, after James D. Watson and Francis H. C. Crick who discovered the double helix structure of DNA in 1953, and won the Nobel Prize for that.

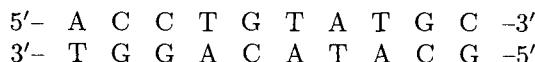


Figure 2.1: Example of a DNA molecule.

Following a standard convention in biochemistry, when we will represent a double stranded molecule, we will draw the two strands one over the other,

with the upper strand oriented from left to right in the 5'-3' direction and the lower strand oriented in the opposite direction. Figure 2.1 presents a DNA molecule composed of ten pairs of nucleotides. The reader is urged to notice the two important restrictions which work here: the complementarity of the paired nucleotides and the opposite directionality of the two strands. These two features are crucial for all our investigations and they induce the computability power of DNA.

Of course, representing a double stranded DNA molecule as two linear strands bound together by Watson–Crick complementarity is a major simplification of reality. First, in a DNA molecule the two strands are wound around each other to form the famous double helix – see Figure 2.2. Moreover, the DNA molecule is folded in an intricate way in order to find room in the limited space of the cell. For the purposes of this book, the spatial structure of DNA is not important, so we ignore it and we assume that a DNA molecule is a double string-like structure as that in Figure 2.1.

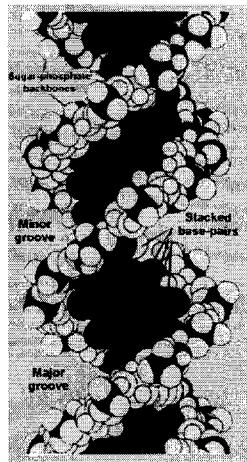


Figure 2.2: The double helix.

There also are many cases where the DNA molecule is not a linear, but a circular one (this happens for many bacteria).

We have called above “molecules” both single stranded and double stranded sequences of nucleotides. We will also work with “incomplete double stranded” molecules, of the type suggested in Figure 2.3. One sees that four nucleotides in the left end and five in the right end are not paired with nucleotides from the opposite strand. We say that the molecule in this figure has *sticky ends*.

We close this section by pointing out again that we work here in an idealized world, governed by clear and reliable rules, error-free, completely predictable. This is far from reality, especially from what happens *in vivo*,

A	C	C	T	G	G	T	T	A	A
	C	C	A	A	T	T	A	T	A

Figure 2.3: A DNA molecule with sticky ends.

where there are many exceptions, non-determinism, probabilistic behaviours, mismatches. For instance, there also are other nucleotides, less frequent and less important than A, C, G, T, while these four basic nucleotides are sometimes paired in a wrong way. We disregard such “real life errors”, on the one hand, because we develop here *computability models* rather than *computer designs*, on the other one, because there are biochemical techniques for isolating the errors, diminishing their ratio in the whole amount of DNA molecules we work with. Moreover, the progresses in the DNA technology are so fast that it is wiser to work in a plausible framework rather than at the present-day level of the technology.

2.2 Complementarity Induces Computational Completeness

We now step back to a formal language theory framework in order to point out a rather surprising connection between the structure of the DNA molecule and computability (at the level of Turing machines). This connection is established via the following consequence of Theorem 1.4.

Consider an alphabet V and its barred variant, $\bar{V} = \{\bar{a} \mid a \in V\}$. The language

$$TS_V = \bigcup_{x \in V^*} (x \amalg \bar{x})$$

is called the *twin-shuffle* language over V . (For a string $x \in V^*$, \bar{x} denotes the string obtained by replacing each symbol in x with its barred variant.)

For the morphism $h : (V \cup \bar{V})^* \longrightarrow V^*$ defined by

$$\begin{aligned} h(a) &= \lambda, \text{ for } a \in V, \\ h(\bar{a}) &= a, \text{ for } a \in V, \end{aligned}$$

we clearly have the equality $TS_V = EQ(h, pr_V)$. In view of Theorem 1.4, this makes the following result plausible.

Theorem 2.1 *Each computably enumerable language $L \subseteq T^*$ can be written in the form $L = pr_T(TS_V \cap R)$, where V is an alphabet and R is a regular language.*

In this representation, the language TS_V depends on the language L . This can be avoided in the following way. Take a coding, $f : V \longrightarrow \{0, 1\}^*$, for

instance, $f(a_i) = 01^i0$, where a_i is the i th symbol of V in a specified ordering. The language $f(R)$ is regular. A gsm can simulate the intersection with a regular language, the projection pr_T , as well as the decoding of elements in $f(TS_V)$. Thus we obtain

Corollary 2.1 *For each computably enumerable language L there is a gsm g_L such that $L = g_L(TS_{\{0,1\}})$.*

Therefore, each computably enumerable language can be obtained by a sequential transducer starting from the unique language $TS_{\{0,1\}}$. One can also prove that this transducer can be a deterministic one.

We stress the fact that the language $TS_{\{0,1\}}$ is *unique*, the same for all computably enumerable languages, while the (deterministic) gsm in this representation depends on the particular language. We may say that all computably enumerable languages are “hidden” in $TS_{\{0,1\}}$ and in order to recover them we only need a gsm, the simplest type of a transducer, nothing else than a finite automaton with outputs; one reading of strings in $TS_{\{0,1\}}$, from left to right, with the control ensured by a finite set of states, suffices. Otherwise stated, the twin-shuffle language plus a finite state sequential transducer are equal in power to a Turing machine. It is clear that the “main” computability capacity lies in the twin-shuffle language.

The technically surprising fact here is the simplicity and the uniqueness of the twin-shuffle language. Still more surprising – and significant from the DNA computing viewpoint – is the connection between $TS_{\{0,1\}}$ and DNA.

Let us start from the simple observation that we have four “letters” both in the case of DNA and in the case of the twin-shuffle language over $\{0, 1\}$. Let us consider a correspondence between these “letters”, taking care of the complementarity: a barred letter is the complement of its non-barred variant. For instance, consider the pairing

$$A = 0, \quad G = 1, \quad T = \bar{0}, \quad C = \bar{1}.$$

Thus, the letters in the pairs $(0, \bar{0})$ and $(1, \bar{1})$ are complementary and this complementarity corresponds to the Watson–Crick complementarity of the nucleotides associated with the symbols $0, 1, \bar{0}, \bar{1}$.

Consider now a double stranded molecule, for instance, that in Figure 2.1,

ACCTGTATGC
TGGACATACG

and let us imagine the following scenario: two clever “insects” are placed at the left end of the two strands of this molecule and asked to “read” the strands, from left to right, step by step, with the same speed, telling us what nucleotide they meet according to the codification specified above (0 for A , 1 for G , etc.). It is clear that what we obtain is a string of symbols $0, 1, \bar{0}, \bar{1}$ which is the letter by letter shuffle of the strings associated with the two strands of the molecule.

For simplicity, let us first rewrite the letters according to the association indicated:

$$\begin{array}{c} \bar{0}\bar{1}\bar{0}1\bar{0}0\bar{1}\bar{1} \\ \bar{0}110\bar{1}0\bar{0}0\bar{1}1 \end{array}$$

Then, the string obtained as suggested above is

$$00\bar{1}\bar{1}1\bar{0}01\bar{1}\bar{0}000\bar{0}01\bar{1}\bar{1}\bar{1}$$

This is a string in $TS_{\{0,1\}}$. Other readings of the molecule also lead to a string in this language; this is the case, for instance, of reading first completely the upper strand and then the lower one.

However, not all strings in $TS_{\{0,1\}}$ can be obtained in this way: the string $00\bar{0}1\bar{0}\bar{1}$ is clearly in $TS_{\{0,1\}}$, but it cannot be produced by the step by step up-down reading (it starts with two copies of 0), or by completely reading one strand and then the other (the two halves of the string are not complementary to each other).

Thus, with the codification of the nucleotides considered above, for each DNA molecule we can find a string in $TS_{\{0,1\}}$ which corresponds to a reading of the molecule, but the converse is not true. This is mainly explained by the fact that the reading of the molecule is too rigid. Unfortunately, if the two “insects” are left to move with uncorrelated speeds along the two strands, then we can get strings which are not in $TS_{\{0,1\}}$. The reader is advised to try with the previous example.

There are other codifications of the nucleotides, also taking into account their places with respect to the two strands, which lead to characterizations of the language $TS_{\{0,1\}}$. For instance, consider the encoding suggested below:

	upper strand	lower strand
A, T	0	$\bar{0}$
C, G	1	$\bar{1}$

In other words, both nucleotides A and T are identified with 0, without a bar when appearing in the upper strand and barred when appearing in the lower strand; the nucleotides C, G are identified with 1 in the upper strand and with $\bar{1}$ in the lower strand. Now, given a DNA (double-stranded) molecule, by reading the two strands from left to right, with non-deterministic non-correlated speeds in the two strands, we get a string in $TS_{\{0,1\}}$: because of the complementarity, if we ignore the bars, then the two strands are described by the same sequence of symbols 0 and 1.

The reader might try with the molecule considered above, imposing no restriction on the speeds of the two “insects”.

Moreover, and this is very important, in this way we can obtain *all* strings in $TS_{\{0,1\}}$: just consider *all* molecules (complete double stranded sequences) and *all* possibilities to read them as specified above. The same result is obtained if we use molecules containing in the upper strand only nucleotides in any of the pairs

$$(A, C), (A, G), (T, C), (T, G).$$

Consequently, we may metaphorically write

$$DNA \equiv TS_{\{0,1\}},$$

so, also metaphorically, we can replace the twin-shuffle language in Corollary 2.1 with DNA.

Thus, we may say that *all possible computations are encoded in the DNA and we can recover them by means of a deterministic finite state sequential transducer – and two “clever insects” as used above.*

The reader is requested to accept the “newspaper style” of this discussion, without losing the significance of the statement (and the precise mathematical meaning of it). Of course, we dare to say nothing about how realistic this discussion is if we think of implementing a “DNA computer” based on these observations.

In particular, there is here a point where the previous discussion is vulnerable: the two “insects” are both placed in the left end of the two strands of a molecule, although we know (see again Figure 2.1) that the two strands are oriented in opposite directions. Let us take into account this opposite directionality and place the “insects”, say, in the 5' ends of the two strands. Pleasantly enough, the conclusion will be the same, because of the robustness of the characterization of computably enumerable languages by means of twin-shuffle languages.

More specifically, the result in Corollary 2.1 is true also for a “mirror” variant of the twin-shuffle language.

For an alphabet V , consider the language

$$RTS_V = \bigcup_{x \in V^*} (x \sqcup mi(\bar{x})).$$

This is the *reverse twin-shuffle language* associated to V .

Theorem 2.2 *For each computably enumerable language L there is a deterministic gsm g_L such that $L = g_L(RTS_{\{0,1\}})$.*

Clearly, reading the upper strand of a molecule from left to right and the lower one from right to left, with non-deterministic uncorrelated speeds, with the codification of nucleotides as used above, we get strings in $RTS_{\{0,1\}}$; all such strings can be obtained by using all molecules and all possible readings. Again, we can identify the language $RTS_{\{0,1\}}$ with DNA and claim that it characterizes CE modulo a deterministic gsm and a certain codification of nucleotides.

We will make use of the twin-shuffle characterizations of computably enumerable languages and of the connection with the DNA structure also in subsequent sections.

We close this section by emphasizing that we have worked here with complete DNA molecules, without sticky ends, and that we have mainly used the structure of the DNA molecule. This suggests to consider a general data structure with similar properties: double stranded sequences, composed of symbols from an arbitrary alphabet, subject to a complementarity relation (which should be at least symmetric).

2.3 Operations on DNA Molecules

We return to the biochemistry of DNA, looking for *operations* possible with molecules. Of course, we will consider these operations in a simplified manner, in terms of the data structure we have specified in Section 2.1 and found to be so useful/powerful in Section 2.2. More biochemical details can be again found in [224] and in the bibliographical sources we indicate at the end of this chapter.

Separating DNA strands. Because the hydrogen bonds between complementary nucleotides are much weaker than the covalent bonds between nucleotides adjacent in the two strands, we can separate the two strands without breaking the single strands. This is easily done by heating the DNA solution (at about 85°–95° C) until the two strands come apart. This operation is called *denaturation*.

Binding together DNA strands. The operation opposite to denaturation is also possible: if a DNA solution of a high temperature (as above, of 85°–95° C) is cooled down, then the separated strands fuse again by hydrogen bonds, and we get double stranded molecules. This operation is called *renaturation*, or *annealing*.

Of course, when the single strands in the solution do not match completely, then by annealing we will obtain DNA molecules with sticky ends; Figure 2.4 presents such a situation.

5' – ACCTAGCGC – 3'

3' – TCGCGTTA – 5'

↓

ACCTAGCGC
TCGCGTTA

Figure 2.4: Annealing of partially matching strands.

Many operations on DNA molecules can be mediated by *enzymes*, which are proteins catalysing various reactions where DNA is involved.

Filling in incomplete strands. A molecule as that in Figure 2.4 can be “completed” to a molecule without sticky ends by using the enzymes called *polymerases*. These enzymes are able to add nucleotides, in the 5'-3' direction, until pairing each nucleotide with its Watson–Crick complement.

There are two important details here: the overhanging strand acts as a template (it precisely specifies the nucleotides to be added, by means of the complementarity restriction) and there already should exist a sequence which is bonded to the template; the addition of nucleotides starts from this sequence and proceeds step by step in the 5'-3' direction. This sequence is called a *primer*. Consequently, the polymerase extends repeatedly the 3' end of the “shorter strand” (starting from the primer), complementing the sequence on the template strand. (Of course, the required nucleotides should be available in the solution where the reaction takes place.) This is called a polymerase chain reaction, in short, PCR.

All polymerases require a primer in order to start adding nucleotides, but there are some polymerases that will extend a DNA molecule without using a prescribed template.

Synthesizing DNA molecules. If we need a specific double stranded molecule for which we already have one strand (a template), then we can obtain it by priming the given strand and then using a polymerase to extend the primer according to the template (in the 5'-3' direction).

One can also synthesize single stranded molecules following a prescribed sequence of nucleotides. This synthesis adds nucleotide by nucleotide to the already synthesized chain in the 3'-5' direction. The procedure is well-controlled and there already are many “synthesizing robots”. We may conclude that “writing” on DNA molecules is already an easy operation (from a technological point of view).

Short chemically synthesized single stranded molecules are called *oligonucleotides* or simply *oligos*. They are very useful in genetic engineering, as well as in DNA computing (for instance, they are used as primers).

Shortening DNA molecules. There are certain enzymes, called *DNA nucleases*, that degrade DNA, removing nucleotides from the two strands. There are two classes of nucleases, *exonucleases* and *endonucleases*.

Exonucleases shorten DNA by removing nucleotides from the ends of the DNA molecule. Some of them remove nucleotides from the 5' end while others will do this from the 3' end. Some exonucleases may be specific for single stranded molecules while others will be specific for double stranded ones (and some can degrade both).

Figure 2.5 shows the action of a 3'-nuclease (it degrades strands in the 3'-5' direction). In this way a molecule is obtained with overhanging 5' ends. Because the removing of nucleotides from the two ends takes (almost) the same time, the operation is synchronized, the two sticky ends have (almost) the same length.

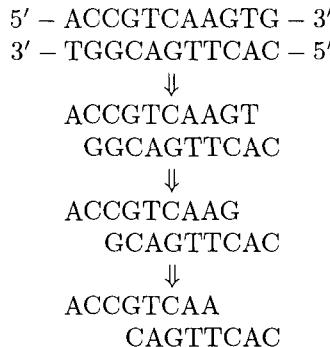


Figure 2.5: An exonuclease in action.

Other exonucleases are cutting pairs of nucleotides from the ends of a molecule, thus synchronously shortening the whole molecule from the two ends (and leaving blunt ends).

Cutting DNA molecules. Endonucleases are able to cut DNA molecules (by destroying the covalent bonds between adjacent nucleotides). Some of them cut only single strands, others can cut double strands; some endonucleases cut DNA molecules at any place, others are *site restricted*, they “recognize” a certain pattern in the molecule and cut at a specified place (in general, inside this pattern).

We are not particularly interested in cutting molecules at random, while the enzymes which can cut at specific sites are crucial for DNA computing, so we will insist on them, recalling some examples from [224].

Endonucleases which cut at specific sites are called restriction endonucleases or *restriction enzymes*.

The cut itself can be *blunt* (straight through both strands) or *staggered*, leaving sticky ends.

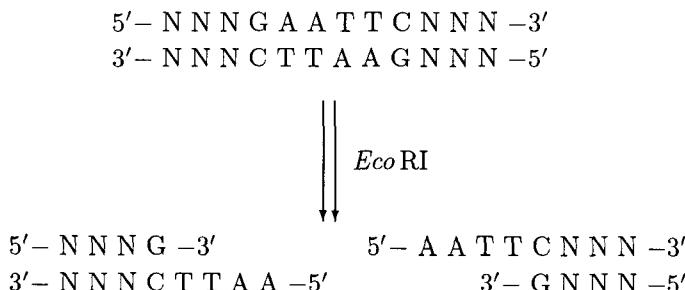


Figure 2.6: Eco RI in action.

Figure 2.6 presents the action of the restriction enzyme *EcoRI*; the sub-molecules filled in with N are arbitrary (of course, the pairs in the two strands are complementary), what counts is the recognition site, 5'-GAATTC, where *EcoRI* will bind. The directionality is very important: *EcoRI* will not bind to 3'-GAATTC. The cut is staggered, leaving two overhanging 5' ends.

Note that the recognition site is a *palindrome* in the sense that reading one of the strands in the 5'-3' direction one gets the same result (GAATTC) as reading the other strand in the 5'-3' direction. This is often the case for restriction enzymes.

Of course, if a molecule of DNA contains several recognition sites, then the restriction enzyme will in principle cut all of them.

Figure 2.7 presents the case of a restriction endonuclease (*SmaI*) which produces blunt ends. The recognition site is 5'-CCCGGG.

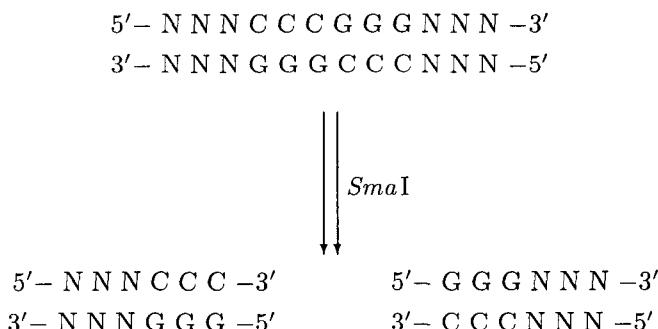


Figure 2.7: *SmaI* in action.

There are many enzymes which have the same restriction sites and produce identical sticky ends. There also are enzymes which produce 3' sticky ends, as well as enzymes which cut outside the restriction site.

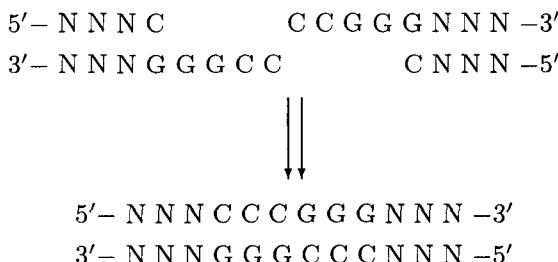


Figure 2.8: A ligation operation.

We emphasize the fact that we can cut DNA molecules at fairly precise

places and producing sticky ends known in advance, and that the enzymes are catalysts, they are not consumed during the operation. Thus, if we wait long enough, then at least in principle all sites where a restriction enzyme can cut will be used.

Linking DNA molecules. Fragments of molecules having complementary sticky ends can be linked together, by an operation called *ligation* (mediated by a class of enzymes called *ligases*). Figure 2.8 presents such a situation.

Thus, by restriction enzymes we can cut DNA molecules and produce sticky ends, while by ligation we can paste together the obtained fragments, possibly obtaining new molecules, by the recombination of fragments. Together with denaturation and annealing, this way of manipulating DNA molecules is one of the basic operations which can be used in DNA computing.

Also possible, but not so useful, is the ligation of blunt ends; in that case the linking of fragments is done without dependence on the nucleotides placed in the two ends.

Inserting or deleting short subsequences. Such operations can be performed by using the annealing of partially matching strands. Schematically, this can be done as follows. Assume that we want to insert a sequence γ in between the sequences α, β of a longer molecule. For any molecule π , denote by π_u the upper strand and by π_d the lower strand of π . We first denature the DNA molecules, thus producing single stranded sequences containing either subsequences $\alpha_u\beta_u$ or subsequences $\alpha_d\beta_d$. The strands of the latter form are removed from the solution, then we add sequences $\alpha_d\gamma_d\beta_d$ and decrease the temperature. The complementary sequences α_u, β_u will anneal to α_d, β_d , while the sequence γ_d will be folded. By a polymerase chain reaction we complete the molecules to double stranded sequences with a single stranded loop γ_d (a primer z_d is also needed, see step (b) in Figure 2.9). We melt again the DNA and we use once more the PCR technique for obtaining double stranded molecules; they have the subsequence γ inserted in the context α, β , as desired.

In a similar way one can delete (or even substitute) a subsequence from a DNA molecule.

Multiplying DNA molecules. One of the main attractive features of DNA from the point of view of computability is the parallelism: in a small space one can have a huge number of molecules. Very important in this context is the fact that we can also have a huge number of *identical* molecules. This can be achieved by a technique called *amplification*, which is performed by means of a Polymerase Chain Reaction.

The amplification by the PCR technique is very simple and very efficient: in n steps it can produce 2^n copies of the same molecule. The operation is done in cycles consisting of three phases: denaturation, priming, extension.

Let us suppose that we have a molecule α which we want to amplify. We

first heat the solution in such a way that all copies of α are denatured, the upper strand of α – denoted as above by α_u – is separated from the lower strand, α_d . We add primers to the solution, short sequences of nucleotides which are complementary to the left end of α_u and to the right end of α_d . By cooling down the temperature, these primers will anneal to the single strands α_u and α_d (at their 5' ends). By adding a polymerase (and a large enough amount of free nucleotides), both α_u and α_d will act as templates, they will be completed with complementary nucleotides, starting from the primers, until obtaining double stranded molecules. In this way, the number of copies of the molecule α is doubled. The process is then repeated.

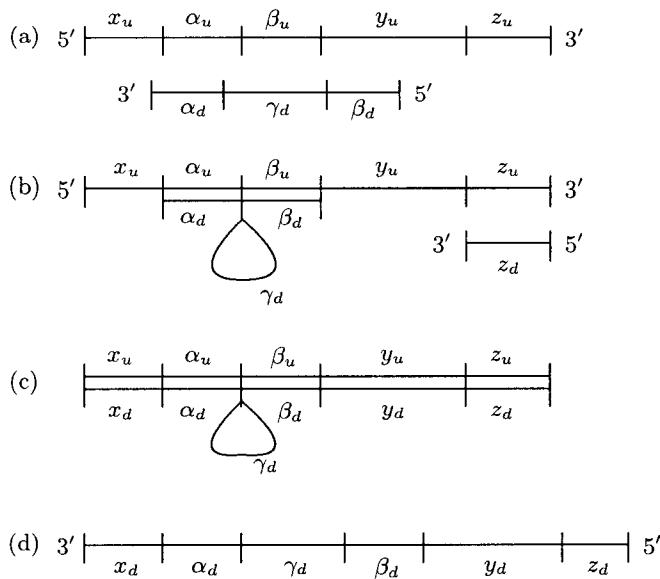


Figure 2.9: Insertion by mismatching annealing.

There are many technical details here which are not important for us. We emphasize only the efficiency of the operation, and the fact that by means of the primers we can select the molecules which are amplified. This is very useful for controlling the errors. Assume that we know that in a solution we have approximately the same number of molecules of type α and of type β and that molecules of type β represent errors. We can increase the ratio of molecules of type α by amplifying only them (using primers which attach to single strands of α and not to single strands of β). In this way, we can increase as much as we want the number of molecules of type α , without modifying the number of molecules of type β .

Filtering a solution. The separation of molecules of type α from those

of type β , as discussed above, can also be done in other ways. If we know that molecules of type β contain a pattern where a given restriction enzyme can act, then we can cut them and after that we can filter the solution by making use of the sticky ends obtained in this way. The operation can also be done for single stranded molecules which contain a known pattern. Principally, the process is very simple: we attach to a solid support strands which are complementary to the known patterns and we pour the solution over this support; the strands complementary to those bound to the support will remain there, the others will be collected separately.

This technique can also be used for separating from a given solution the set of single stranded molecules which contain a specified subsequence.

Separation of molecules by length. It is also possible to separate the molecules present in a solution according to their length, by a technique called *gel electrophoresis*. In fact, the molecules are separated according to their weight, which is almost proportional to their length. In several wells performed at one end of a rectangular sheet of gel, one pours a small amount of DNA, then one applies an electrical field. Because the DNA molecules are negatively charged, they will move toward the positive pole, with a speed which depends on their weight (and the gel porosity). In this way, longer molecules will remain behind the shorter ones and we can both check the existence of molecules of a given length (by comparing the trace of the unknown molecules with that of a probe which is run in parallel), as well as separate the molecules according to their length.

Gel electrophoresis (that is, separation by length) is one of the ways of “reading” the result of a DNA computation. The procedure is rather precise, even molecules which differ by one nucleotide only can be distinguished from each other.

Reading DNA molecules. Of course, separation by length cannot tell us the precise content of molecules, so there are problems where this technique is not sufficient. Fortunately, there also are techniques able to read a DNA molecule nucleotide by nucleotide. We do not present here the details of such a procedure (Chapter 1 of [224] can be consulted in this aim), but we only mention that such procedures exist (and that, for the time being, they are slower than the procedures for synthesizing molecules with a specified contents; at this moment, it is easier to “write” on DNA than to “read” it).

2.4 Adleman’s Experiment

Most of the experiments in DNA computing reported so far are based on the annealing operation. This is the case also with Adleman’s seminal experiment. We start now to present several experiments of this type, then we will discuss some theoretical or practical generalizations of them, leading to new

computability models or to new computability strategies suggested by these experiments.

Speculations about using DNA molecules as a support for computations were already made some decades ago, e.g., by Ch. Bennett [23] and M. Conrad [63], but they were not followed by practical attempts to implement those revolutionary ideas. The first successful experiment of using DNA molecules and techniques for computing was reported by L. M. Adleman in 1994 [1]: a small instance of the Hamiltonian path problem in a directed graph was solved by purely biochemical means. The number of the laboratory steps was linear in terms of the size of the graph (the number of vertices), although the problem itself is known to be NP-complete (hence intractable for usual computers). Following the terminology of [124], this was a convincing *demo*, which has proved that genetic engineering materials and techniques constitute a possible new framework for computability. Adleman's experiment is important not only because it was the first of this type, but also by the way it was conducted (by the way it has made use of the structure of the DNA and of the operations possible with DNA molecules). Thus, we will recall this history making experiment (skipping the biochemical details and emphasizing the steps of the procedure).

The graph considered by Adleman was that in Figure 2.10. We have seven vertices and fourteen arcs. The question is whether or not there is a path from vertex 0 to vertex 6 which passes exactly once through each of the other vertices. That is, we have a decidability problem, with a yes/no answer.

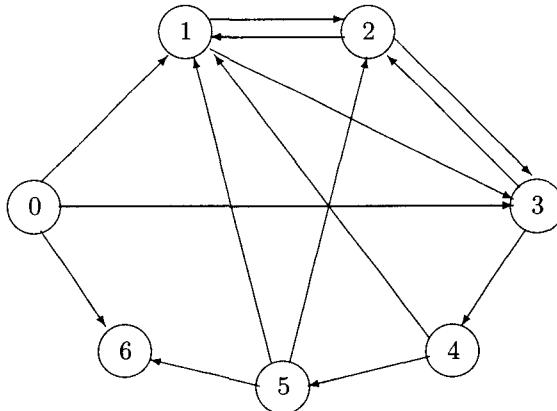


Figure 2.10: The graph in Adleman's experiment.

By a simple examination of the graph, we see that there is a path as desired, namely that following the numbering of the vertices: 0, 1, 2, 3, 4, 5, 6. However, we have mentioned that the problem is a hard one, among the hardest which can be solved in polynomial time by non-deterministic algorithms:

it is an NP-complete problem. Otherwise stated, all known deterministic algorithms for solving this problem are essentially equally complex as the exhaustive search. However, we can trade time for space, and this is exactly what Adleman has done, making use of the massive parallelism of DNA (in some sense, massive parallelism can simulate non-determinism, and in this way non-deterministic algorithms can be implemented).

The algorithm used by Adleman was the following:

Input: A directed graph G with n vertices, among which there are two designated vertices v_{in} and v_{out} .

Step 1: Generate paths in G randomly in large quantities.

Step 2: Remove all paths that do not begin with v_{in} or do not end in v_{out} .

Step 3: Remove all paths that do not involve exactly n vertices.

Step 4: For each of the n vertices v , remove all paths that do not involve v .

Output: “Yes” if any path remains, “No” otherwise.

It is easy to see that, providing that each of the operations in the algorithm takes exactly one time unit, the solution is obtained in a number of time units which is linear in n , the number of vertices in the graph: steps 1, 2, 3 need a constant number of time units (say, 4: generate all paths, select the paths starting with v_{in} , select the paths ending with v_{out} , select the paths of length n), while step 4 takes n time units (check for each vertex its presence in the currently non-rejected paths).

The main difficulty lies in step 1, where we have to generate a large number of paths in the graph, as large as possible, in order to reach with a high enough probability the Hamiltonian paths, if any. Of course, when the graph also contains cycles, the set of paths is infinite. In a graph without cycles, the set of paths is finite, but it can be of an exponential cardinality with respect to the number of vertices. This is the point where the massive parallelism and the non-determinism of chemical reactions were cleverly used by Adleman in such a way that this step was performed in a time practically independent of the size of the graph.

The biochemical implementation of the above described algorithm was the following.

Each vertex of the graph was encoded by a single stranded sequence of nucleotides, namely of length 20. These codes were constructed at random; the length 20 is enough in order to ensure that the codes are “sufficiently different”. A huge number of these oligonucleotides (amplified by PCR) were placed in a test tube. Then, in the same test tube are added (a huge number of) codes of the graph edges, of the following form: if there is an edge from

vertex i to vertex j and the codes of these vertices are $s_i = u_i v_i$, $s_j = u_j v_j$, where u_i, v_i, u_j, v_j are sequences of length 10, then the edge $i \rightarrow j$ is encoded by the Watson-Crick complement of the sequence $v_i u_j$.

For instance, for the codes of vertices 2, 3, 4 specified below

$$\begin{aligned}s_2 &= 5' - \text{TATCGGATCGGTATATCCGA} - 3', \\ s_3 &= 5' - \text{GCTATTGAGCTTAAAGCTA} - 3', \\ s_4 &= 5' - \text{GGCTAGGTACCAAGCATGCTT} - 3',\end{aligned}$$

the edges $2 \rightarrow 3, 3 \rightarrow 2$ and $3 \rightarrow 4$ were encoded by

$$\begin{aligned}e_{2 \rightarrow 3} &= 3' - \text{CATATAGGCTCGATAAGCTC} - 5', \\ e_{3 \rightarrow 2} &= 3' - \text{GAATTCGATATAGCCTAGC} - 5', \\ e_{3 \rightarrow 4} &= 3' - \text{GAATTCGATCCGATCCATG} - 5'.\end{aligned}$$

By annealing, the codes of the vertices act as splints in rapport with codes of edges and longer molecules are obtained, encoding paths in the graph. The reader can easily see how the molecules specified above will lead to sequences encoding the paths $2 \rightarrow 3 \rightarrow 4, 3 \rightarrow 2 \rightarrow 3 \rightarrow 4$, etc.

Adleman has let the process to proceed four hours, in order to be sure that all ligation operations take place. What we obtain is a solution containing a lot of non-Hamiltonian paths (short paths, cycles, paths passing twice through the same vertex). The rest of the procedure consists of checking whether or not at least a molecule exists which encodes a Hamiltonian path which starts in 0 and ends in 6.

There is here a very important point, which will be referred to later. The difficult step of the computation was carried out "automatically" by the DNA molecules, making use of the parallelism and the Watson-Crick complementarity. In this way, we get a large set of *candidate solutions*, (hopefully) both molecules which encode paths which are looked for, but also many molecules which should be filtered out, a sort of "garbage" to be rejected.

This second part of the procedure, of filtering the result of step 1 in order to see whether a solution to our problem exists, was carried out by Adleman in about seven days of laboratory work, by performing the following operations: By a PCR amplification with primers representing the input and the output vertices (0 and 6), only paths starting in 0 and ending in 6 were preserved. After that, by gel electrophoresis there have been extracted the molecules of the proper length: 140, because we have 7 vertices encoded by 20 nucleotides each. Thus, at the end of step 3 we have a set of molecules which encode paths in the graph which start in 0, end in 6, and pass through 7 vertices. (It is worth noting that this does not ensure that such a path is Hamiltonian in our graph: 0, 3, 2, 3, 4, 5, 6 is a path which visits seven vertices, but it passes twice through 3 and never through 1.) Roughly speaking, step 4 is performed by repeating for each vertex i the following operations: melt the

result of step 3, add the complement of the code s_i of vertex i and leave to anneal; remove all molecules which do not anneal.

If any molecule survives step 4, then it encodes a Hamiltonian path in our graph, namely one which starts in vertex 0 and ends in vertex 6.

The practical details of this procedure are not very important for our goals. They depend on the present day laboratory possibilities and can be performed also by other techniques; furthermore, the algorithm itself can be changed, improved or completely replaced by another one. What is important here is the proof that such a computation is possible. Purely biochemical means were used in order to solve a hard problem, actually an intractable one, in a linear time as the number of lab operations. These operations, in an abstract formulation, form another main output of this experiment and of the thought about it, leading to a sort of a programming language based on test tubes and DNA molecules manipulation.

Such a “test tube programming language” was proposed in [168], developed in [2] and then discussed in many places. In short, the framework is the following.

By definition, a (*test*) *tube* is a multiset of words (finite strings) over the alphabet {A, C, G, T}. The following basic operations are initially defined for tubes, that is, multisets of DNA single strands [2]. However, appropriate modifications of them will be applied for DNA double strands as well.

Merge. Given tubes N_1 and N_2 , form their union $N_1 \cup N_2$ (understood as a multiset).

Amplify. Given a tube N , produce two copies of it, N_1 and N_2 .

Detect. Given a tube N , return *true* if N contains at least one DNA strand, otherwise return *false*.

Separate (or *Extract*). Given a tube N and a word w over the alphabet {A, C, G, T}, produce two tubes $+(N, w)$ and $-(N, w)$, where $+(N, w)$ consists of all strands in N which contain w as a (consecutive) substring and, similarly, $-(N, w)$ consists of all strands in N which do not contain w as a substring.

The four operations of **merge**, **amplify**, **detect**, and **separate** allow us to program simple questions concerning the occurrence and non-occurrence of subwords. For instance, the following program extracts from a given test tube all strands containing at least one of the purines A and G, preserving at the same time the multiplicity of such strands:

- (1) *input(N)*
- (2) *amplify(N)* to produce N_1 and N_2
- (3) $N_A \leftarrow +(N_1, A)$
- (4) $N_G \leftarrow +(N_2, G)$

- (5) $N'_G \leftarrow -(N_G, A)$
- (6) $\text{merge}(N_A, N'_G)$

Besides the four operations listed above, Adleman's experiment makes use of the following modifications of the operation **separate**:

Length-separate. Given a tube N and an integer n , produce the tube $(N, \leq n)$ consisting of all strands in N with the length less than or equal to n .

Position-separate. Given a tube N and a word w , produce the tube $B(N, w)$ (resp. $E(N, w)$) consisting of all strands in N which begin (resp. end) with the word w .

Using them, we can describe the filtering procedure in Adleman's experiment (steps 2, 3, 4) as follows. We start with the input tube N , consisting of the result of step 1, in the form of single stranded molecules (obtained by melting the result of step 1). The program is the following:

- (1) $\text{input}(N)$
- (2) $N \leftarrow B(N, s_0)$
- (3) $N \leftarrow E(N, s_6)$
- (4) $N \leftarrow (N, \leq 140)$
- (5) $\text{for } i = 1 \text{ to } 5 \text{ do } \begin{aligned} & N \leftarrow + (N, s_i) \\ & \text{end} \end{aligned}$
- (6) $\text{detect}(N).$

The operations **merge**, **amplify**, **detect**, **separate** will be used also in the subsequent sections. We do not persist here in further discussing these operations, but we point out one of the main weakness of Adleman's procedure, from a practical point of view: the number of necessary single strands, codes of vertices or of edges, is of the order of $n!$, where n is the number of vertices in the graph. This imposes drastic limitations on the size of the problems which can be solved in this manner. (J. Hartmanis has computed [125] that in order to handle in this manner graphs with 200 nodes, a size of a practical importance and easily handled by electronic computers, we need $3 \cdot 10^{25}$ Kg of DNA, which is more than the weight of the Earth!) In short, Adleman's procedure is elegant, copes in a nice way with the errors, but it cannot be scaled-up.

The question of increasing the size of the graphs for which we can solve the Hamiltonian path problem was approached by several authors. A general strategy is to diminish the set of candidate solutions one has to generate by a basic step where the parallelism of DNA is essentially used. This leads to the idea of evolutionary computing, based on a repeated improvement of the current set of candidate solutions, similar to the style of genetic algorithms, where only "individuals" with a high fitness survive each iteration. Such an approach to our problem can be found in [15].

Another approach to the Hamiltonian path problem was proposed by T. Head [128]. It starts from the observation that single stranded DNA cannot be used for large graphs, because of the “bad behaviour” of long sequences of nucleotides (they are fragile, can selfanneal, etc.). Head’s procedure still has two phases, one when candidate solutions are generated and one when non-solutions are eliminated, but the first phase is carried out in a way that ensures the fact that the candidate solutions does not grow too much. More specifically, one takes care that only molecules encoding paths of a length less than or equal to the length of the Hamiltonian paths are produced. To this aim, a number of steps which is proportional to the number of vertices is performed. The filtering phase is similar to that in Adleman’s procedure; thus, in total, the number of biochemical steps is again linear with respect to the size of the graph.

The key tool used in Head’s algorithm is the restriction enzyme *Dra*III, which recognizes a pattern of the form

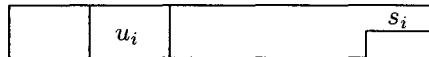


and cuts the molecule such that the following sticky ends are produced:



Here, N denotes any nucleotide (of course, the upper nucleotides should be complementary to the lower ones), therefore we can have $4^3 = 64$ possibilities.

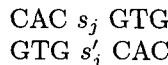
Let us choose a single stranded sequence of length three, s_i , for each vertex i of a graph, as well as another sequence, u_i , which is double stranded, also associated to vertex i . For the initial edges, $0 \rightarrow i$, we consider molecules of the form



where the parts of the molecule placed to the left and to the right of u_i are used for purposes we do not mention here. For each edge $i \rightarrow j$ we consider molecules of the form



where s'_i is the Watson-Crick complement of s_i and α_j is the sequence



that is, a pattern recognized by *Dra*III, with the three central pairs of nucleotides associated with vertex j .

If we place these molecules in the same test tube, they will anneal, and paths of length two of the form $0 \rightarrow j$ are produced, encoded by molecules with blunt ends. If *Dra*III is added, then the end of these molecules is cut and a sticky end associated with vertex j is produced.

We continue the procedure: add molecules associated with edges $j \rightarrow k$, of the same form as those associated with edges $i \rightarrow j$, then add the restriction enzyme, and so on and so forth. At each step, the length of the generated paths increases by one. After n steps, where n is the number of vertices in the graph, we have completed the first phase, of generating candidate solutions. By a filtering phase, similar to that in Adleman's experiment or a different one, making use of the specific molecules we use, we can check whether or not at least a path is Hamiltonian.

The efficiency of this procedure with respect to the quantity of DNA used is obvious (but we know no information about a possible implementation of it). A fundamental limitation of Head's proposal is the number of patterns of length three we can use for encoding the vertices: 64 for *Dra*III. However, as T. Head says ([128], page 81), "the report of a laboratory solution based on a graph with 50 vertices and 100 edges would be very reassuring" and we agree with that.

2.5 Other DNA Solutions to NP Complete Problems

Adleman's biochemical solution to the Hamiltonian path problem was immediately followed by several generalizations and extensions to other NP complete problems. It is perhaps interesting to note that several authors have proposed experiments aiming to solve hard and interesting problems from a theoretical or a practical point of view, but not all of these experiments were effectively done in the laboratory. G. Rozenberg has coined a nice term for describing this activity, *memmology*, from *mental molecular biology*. It is important to note the practical flavour of *memmology* in comparison with DNA computing *in info*, which is focused on models of computability, not on their possible implementation.

We continue here in "memmological terms", by briefly describing several experiments, some of them actually done, which have followed Adleman's seminal paper and were of a similar type (they have basically used the operations `merge`, `amplify`, `separate`, `detect` described in the previous section). Together with M. Amos, the first author of a PhD Thesis in DNA computing, [8], we say that these experiments are based on *filtering models*. M. Amos considers two other classes: *splicing models* and *constructive models*. We will devote several sections to splicing models, while constructive models will be

considered in subsequent sections of this chapter, without explicitly calling them “constructive”.

One of the most significant immediate extensions of Adleman’s method was proposed by R. Lipton [168], who indicates a way to solve the *satisfiability problem for propositional formulas* (in short, the SAT problem) in terms of DNA manipulations; the paper [168] is the place where the idea of the “DNA programming language” based on the operations mentioned above was considered for the first time.

We do not give too many prerequisites from logics, but we only recall that the SAT problem refers to the existence of a truth assignment for the variables of a well-formed formula in the conjunctive normal form which satisfies the formula.

For instance,

$$\alpha = (x_1 \vee \sim x_2 \vee x_3) \wedge (x_2 \vee x_3) \wedge (x_1 \vee x_3) \wedge \sim x_3$$

is such a formula. It involves three variables, x_1, x_2, x_3 , as well as the *connectives* \sim, \vee, \wedge (negation, disjunction, conjunction). We have four *clauses*, $x_1 \vee \sim x_2 \vee x_3$, $x_2 \vee x_3$, $\sim x_1 \vee x_3$, and $\sim x_3$. If we can find a truth assignment such that each clause is true, then the whole formula would be satisfiable. (Incidentally, this is the case for α above: the last clause is *true* only when x_3 is *false*; from the second clause we then need x_2 *true* and from the third clause we need that x_1 is *true*; these truth values also make the first clause *true*. Consequently, formula α is satisfiable, the truth assignment $x_1 = x_2 = \text{true}$, $x_3 = \text{false}$ leads to value *true* for the formula.)

The SAT problem is probably the most used NP complete problem, in very many places in order to prove that a problem is NP complete one reduces it (in polynomial time) to SAT.

Lipton’s solution is based on a natural reduction to a graph problem and consists of two phases, the generation of all paths in a graph and looking for a truth assignment which satisfies a given formula. Here are some details.

Let us assume that we have a propositional formula with k variables. We construct the graph in Figure 2.11.

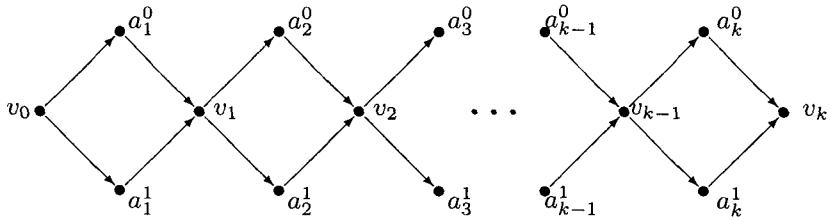


Figure 2.11: A graph associated with a truth assignment.

The vertices v_1, \dots, v_k are associated with the variables, the vertices a_i^0, a_i^1

are associated with the truth values *false*, *true*, respectively. A path $v_{i-1}a_i^jv_i$ encodes the fact that the variable x_i takes the truth value j , where $0 = \text{false}$ and $1 = \text{true}$. Thus, a path from v_0 to v_k will indicate a truth assignment to the k variables x_1, \dots, x_k .

The first phase of Lipton's method consists precisely of generating all paths from v_0 to v_k , hence all truth assignments for the k variables. This can be done in the same way as in Adleman's experiment: consider single stranded oligonucleotides associated with vertices, as well as complementary sequences associated with prefixes and suffixes of these sequences; by annealing, these sequences will play a "domino game" which will lead to molecules encoding paths in our graph. Select all paths which start in v_0 and end in v_k . This condition is sufficient in order to ensure that all vertices $v_i, 1 \leq i \leq k-1$, are visited exactly once, while from each pair $(a_i^0, a_i^1), 1 \leq i \leq k$, exactly one vertex is visited.

Several details are worth emphasizing. The graph is associated with the variables, not with the formula, that is, we can prepare a tube containing all the paths from v_0 to v_k once for all formulas with k variables. This graph has no cycles, therefore the set of possible paths is finite, of a well definite upper length. There is no Hamiltonian path in this graph, but also Head's method for the Hamiltonian path problem can be used in the first phase of Lipton's method.

In short, making use of the DNA parallelism, we can generate all possible truth assignments for a formula with a given number of variables (in a number of biochemical steps which is linear with respect to the number of variables). Let us denote by N_0 the test tube (in the sense of the programming language introduced in the previous section, that is, as a multiset of molecules) produced by the first phase.

The second phase of the algorithm takes N_0 and selects from it all paths which satisfy the first clause, produces a new tube, N_1 , and continues in this way; after having a tube $N_i, 0 \leq i \leq k-1$ (which contains paths which correspond to truth assignments which satisfy the first i clauses), one selects those paths from N_i which also satisfy clause $i+1$ and one produces the tube N_{i+1} . If tube N_k still contains any molecule, then the formula is satisfiable.

All these operations can be performed biochemically and can be written as instructions `merge`, `separate`, `detect`. We exemplify with the formula considered in [168], by recalling some paragraphs from Chapter 2 of [224].

Consider the propositional formula

$$\beta = (x_1 \vee x_2) \wedge (\sim x_1 \vee \sim x_2).$$

We have two variables, hence the corresponding graph is as shown in Figure 2.12.

Each of the four paths through this graph corresponds to one of the four truth assignments for the variables x_1 and x_2 . The initial test tube, N_0 , constructed as indicated above, contains strands for each of the paths

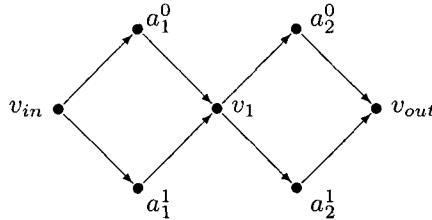


Figure 2.12: The graph associated with formula β .

and, consequently, for each of the truth assignments. Given the length of the oligonucleotides encoding the vertices a_j^i , these oligonucleotides will be easily distinguishable from each other, even in the case of a much larger number of variables. For instance, the oligonucleotide encoding a_1^1 does not appear in the paths elsewhere than in the intended position. If we apply the operation **separate**, forming the test tube $+(N_0, a_1^1)$, then we get those truth assignments where x_1 assumes the value 1 (*true*). This simple observation is the basis of the whole procedure.

We denote the truth assignments by two-bit sequences in the natural way. Thus, 01 stands for the assignment $x_1 = 0, x_2 = 1$. A similar notation is also used if there are more than two variables. This notation of bit sequences is extended to the DNA strands resulting from our basic graphs. Thus, the strand $v_{in}a_1^0v_1a_2^1v_{out}$ is denoted simply by 01.

The following program solves the satisfiability problem for the propositional formula β :

- (1) *input*(N_0)
- (2) $N_1 = +(N_0, a_1^1)$
- (3) $N'_1 = -(N_0, a_1^1)$
- (4) $N_2 = +(N'_1, a_2^1)$
- (5) *merge*(N_1, N_2) = N_3
- (6) $N_4 = +(N_3, a_1^0)$
- (7) $N'_4 = -(N_3, a_1^0)$
- (8) $N_5 = +(N'_4, a_2^0)$
- (9) *merge*(N_4, N_5) = N_6
- (10) *detect*(N_6)

The reader can check that the contents of the tubes at the different steps of the program are as follows:

Step	1	2	3	4	5	6	7	8	9
Tube	00,01,10,11	10,11	00,01	01	10,11,01	01	10,11	10	01,10

Thus, at step (10) the program returns *true*, that is, formula β is satisfiable.

The program is based on exhaustive search. The initial tube at step (1) contains all possible truth assignments. The tube at step (5) contains the assignments satisfying the first clause of the propositional formula β . (Either x_1 or x_2 must assume the value 1. At step (2) we have those assignments for which x_1 is 1. Of the remaining ones we still take, at step (4), those for which x_2 is 1.) The assignments in this tube, N_3 , are filtered further to yield at step (9) those assignments that also satisfy the second clause of the propositional formula β .

A further extension of the filtering methods of Adleman's type was proposed in [9] and developed in a series of papers of the authors of [9] and their collaborators. The method uses the following four basic operations (N, N_1, \dots, N_k are test tubes, that is, multisets of molecules, and S is a set of molecules):

- **remove**(N, S): remove from the tube N any molecule which contains as a subsequence at least one occurrence of each element of S ;
- **union**($\{N_1, \dots, N_k\}, N$): create the tube N by merging the tubes N_1, \dots, N_k ;
- **copy**($N, \{N_1, \dots, N_k\}$): produce k copies of N , denoted by N_1, \dots, N_k ;
- **select**(N): if N is not empty, then select an element of N at random; otherwise return *empty*.

(We have preserved the terminology of [8], although some of these operations have counterparts also in the programming language used above.)

These basic operations are assumed to take constant time when executed in a parallel manner.

In this framework, algorithms for many NP complete problems can be described. Actually, it is claimed in [8] – but only proved by examples – that *all* NP complete problems can be approached in this way; among the considered problems, we mention: generating all permutations of a given finite set, the three-vertex-colourability problem (given a graph, decide whether or not the vertices can be coloured with three colours in such a way that no two adjacent vertices have the same colour), the Hamiltonian path problem, the subgraph isomorphism problem (given two graphs, find whether or not one of them is a subgraph of the other, modulo renaming the vertices), find the maximum clique in a graph (a clique is a complete graph, that is, containing all possible edges).

It is interesting to mention the complexity of the algorithms able to solve these problems, as the number of operations **remove**, **union**, **copy**, **select**: the permutations of n elements are generated in $O(n!)$, the same time is needed in the case of three-vertex-colouring, where n is the number of vertices, the

Hamiltonian path problem is solved in constant time providing that one starts from a solution of the permutation generation problem, the last two problems are again solved in parallel time $O(n)$, where n is the number of vertices.

Because of the (historical) importance of the Hamiltonian path problem, we recall from [8] the algorithm for solving it:

Problem: Generate the set P_n of all permutations of the set $\{1, 2, \dots, n\}$.

Input: A tube N containing all strings of the form $p_1 i_1 p_2 i_2 \dots p_n i_n$, where p_j is a code for “position j ” and $i_j \in \{1, 2, \dots, n\}$, for each $1 \leq j \leq n$ (note that each integer i_j can appear several times in an element of N). Tube N can be produced in an easy way, by the “splint procedure” based on annealing as in Adleman’s experiment.

Algorithm:

```

for  $j = 1$  to  $n - 1$  do
    begin
        copy( $N, \{N_1, \dots, N_n\}$ )
        for  $i = 1, 2, \dots, n$ 
            in parallel do remove( $N_i, \{p_j l \mid l \neq i\} \cup \{p_k i \mid k > j\}$ )
        union( $\{N_1, \dots, N_n\}, N$ )
    end
 $P_n \leftarrow N$ 

```

Complexity: $O(n)$ parallel time.

After the j th iteration of the *for* loop, in the surviving sequences the integer i_j does not appear at any of the positions $k > j$. Thus, at the end of the computation each of the surviving strings will contain exactly one occurrence of each integer in the set $\{1, 2, \dots, n\}$, hence the string represents a permutation of this set. The output set, P_n , is used as input in the following algorithm.

Problem: Determine whether or not a given graph $G = (V, E)$ (with n vertices in the set V) contains a Hamiltonian path.

Input: The tube P_n produced by the previous algorithm. An integer i at position p_k in a permutation present in P_n is interpreted as follows: the string represents a candidate solution to the problem in which vertex i is visited at step k .

Algorithm:

```

for  $2 \leq i \leq n - 1$  and  $j, k$  such that  $(j, k) \notin E$ 
    in parallel do remove( $P_n, \{j p_i k\}$ )
select( $P_n$ )

```

Complexity: Constant parallel time given P_n .

Clearly, in the surviving molecules there is an edge in G for each consecutive pair of vertices encoded in the molecules. Since we start from a permutation of the vertex set, the surviving paths are Hamiltonian.

Note that we obtain *all* Hamiltonian paths, not only the answer to the question whether or not such a path exists.

It is worth noting the variety of the answers given by the algorithms mentioned above, illustrating the variety of problems the DNA computing can approach. In Adleman's experiment we have a decidability problem, the answer is *yes* or *no* and it can be "read" by a **detect** operation (**select**, in [8]). Also in the graph isomorphism problem we look for a Boolean answer. This is not the case with the problem of generating all permutations of a set, where we have to construct a set of molecules with a specified property. However, these permutations are known in advance, we need them, in general, as an input for other algorithms (this is perfectly illustrated above with the algorithm given in [8] for solving the Hamiltonian path problem). Different from these is the case of the maximal clique problem, where we ask for a number. A number can be "read" from DNA molecules in various modes, the simplest one being probably gel electrophoresis, by carefully handling the length of the used molecules. Of course, by effectively sequencing DNA molecules we can get as an output of an algorithm any type of information. For instance, in the Hamiltonian path problem we might ask not only whether a path exists or not, but, in the affirmative case, to explicitly exhibit a path or all existing paths.

In [8], the maximal clique problem is reduced to the graph isomorphism problem, in the following manner: consider a graph G with n vertices; for each total graph K_i , with i vertices, $i \leq n$, check whether or not K_i is a subgraph of G ; the maximum i for which the answer is affirmative gives the answer to our problem.

A much more appealing procedure is considered in [203] (and effectively implemented; this was one of the earliest successful experiments dealing with NP complete problems reported after Adleman's breakthrough). The algorithm in [203] is a direct one and it consists of the following four steps:

1. Consider a graph $G = (V, E)$ with n vertices. The vertices are denoted by the digits $0, 1, \dots, n - 1$, while subsets of vertices (hence also the possible cliques) are represented by binary strings of length n : a digit set to 1 indicates that the corresponding vertex is present in the subset, a digit set to 0 indicates that the corresponding vertex is not present; we start counting from the right (the rightmost position is assigned to vertex 0, the next one is assigned to vertex 1, and so on until the leftmost position which is associated with vertex $n - 1$). For instance, in a graph with six vertices, the possible clique $\{1, 3, 4\}$ is represented by the binary string 011010.

In this way, the set of all possible sets of vertices is represented by the set of all binary strings (we may consider them numbers) of length n .

This set is called the *complete data pool*.

2. We construct the *complementary graph* with respect to G , that is, the graph with the same vertices, but containing an edge (i, j) if and only if (i, j) is not an edge in the graph G . Thus, any two vertices which are connected in the complementary graph are disconnected in G and therefore cannot be members of the same clique.
3. We eliminate from the complete data pools all strings containing connections in the complementary graph. The surviving strings correspond to all cliques in the original graph.
4. We sort the remaining data pool in order to find the strings containing the largest number of occurrences of 1. Each of these occurrences of 1 represents a vertex in the corresponding clique. The clique with the largest number of 1's indicates the size of the maximal clique.

The biochemical details of the way of implementing the two main phases of the procedure, constructing the complete data pool and filtering it until obtaining the maximal clique, can be found in [203]. The basic operations were again annealing, PCR with selective primers, digestion with restriction enzymes and length selection by gel electrophoresis. We do not enter into details, but we invite the reader to remember an important feature of all these experiments, irrespective whether or not they were actually done or they are only “mental experiments” in the *memmology* sense: one first generates a large set of candidate solutions, then one filters this set until only actual solutions, if any, remain. The first phase is crucially based on the huge parallelism made available by the DNA biochemistry. We will return to this observation in a subsequent section, when we will discuss the *computing by carving*. Many other experiments in DNA computing were carried out in various places. Some of them were successful, some of them are still in progress; some of them will be discussed in the subsequent sections, some of them not. For convenience, we mention some references, by no means trying to be exhaustive (this is simply impossible: on the one hand, not all experiments were reported in largely available places, on the other hand, many experiments are continuously reported and it is difficult to keep pace with all of them): [18], [120], [148], [151], [170], [200], [201], [217], [300].

2.6 A Two-Dimensional Generalization

The annealing operation does not suffice in order to capture the full power of Turing machines – we shall formally prove this in Section 2.8. Consequently, the computing models starting from the annealing operation and using operations of the form **separate**, **merge**, **amplify**, etc., are not enough in what concerns their computability *competence*. This was observed also in what

concerns the *performance* of such models: in [293] it is shown that these operations are not sufficiently powerful in order to invert functions defined by circuits in linear time.

This remark raises the question of looking for an additional feature/operation in such a way to increase the power of the obtained model. A theoretical solution is discussed in Section 2.8: considering some control on the annealing operation. Another solution, which was also practically checked, was developed in a series of papers by E. Winfree and his collaborators (see [294], [295], [298]): annealing plus ligation, in a two-dimensional framework.

Taking into account that one uses no explicit control on applying these operations, the fact that we obtain computational completeness is somewhat surprising. The “explanation” lies in the power of two-dimensionality. The link between DNA and computability is made in this case by cellular automata, in the variant known under the name of *partitioning cellular automata in 2D*, [287], which are known to characterize the computably enumerable languages. Following the mentioned papers by E. Winfree, we call them *blocked cellular automata*.

In short, such an automaton consists of a bi-infinite tape where symbols from a given alphabet, V , can be written; two symbols from V are distinguished; one is the blank symbol and the other one is the stop symbol. One gives a set of *instructions* (we also call them *rules*, as in a rewriting system) of the form $ab \rightarrow cd$, where a, b, c, d are symbols from V . These rules are used in parallel. At any step, all symbols written on the tape are rewritten by using such rules, with the following important restriction: from a step to the next one, the place of using the rules is shifted with one position. The computation starts from an initial configuration of the tape (a given string written on it) and stops at the first step when the stop symbol is introduced. (An equivalent variant is to stop in a halting configuration.) The contents of the tape after a completed computation is the result of the computation.

It is known that blocked cellular automata are computationally complete, they characterize the computably enumerable languages. (From the form of rules, they have both context-sensitivity and erasing possibilities; the shifted application of rules ensures the global control on the tape contents, that is, the possibility of sending information at an arbitrary distance. Such features are generally known to equal the power of type-0 Chomsky grammars, hence that of Turing machines.)

The “codification” of a blocked cellular automaton in terms of DNA is rather tricky and based on a DNA structure already known since several years, see [109], the so-called DAE (from “Double-crossover, Antiparallel helical strands with an Even number of half-turns between crossovers”). It is obtained as suggested in Figure 2.13.

We start from five single stranded sequences as indicated in the left side of the figure. By annealing, they will form the “domino” in the right hand

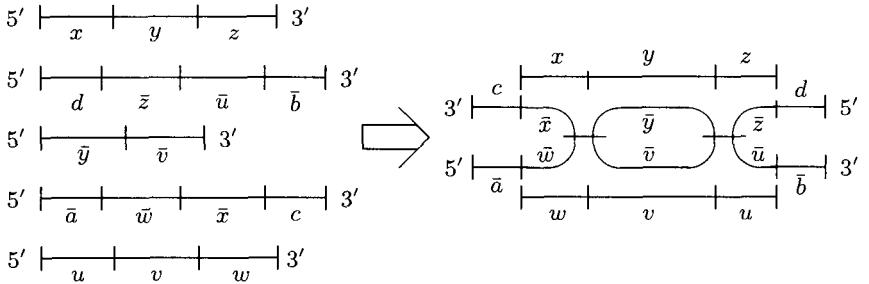


Figure 2.13: Constructing a DAE.

of the figure. It is known that this construct is rigid and planar and has four sticky ends. The subsequences $x, y, z, u, v, w, a, b, c, d$ should be carefully chosen; the barred sequences \bar{x}, \bar{y} , etc. are the complementary strands with respect to x, y , etc.

It is important to note that we have four sticky ends, \bar{a}, \bar{b}, c , and d . They are denoted as the four letters in an instruction $ab \rightarrow cd$ of a blocked cellular automaton and this is the key idea of E. Winfree. We shall present the way of simulating a blocked cellular automaton by using DAE constructs as sketched above in a more schematic manner, namely, by representing a DAE “domino” as a square – Figure 2.14. We only preserve the information concerning the four sticky ends (hence the instruction of the cellular automaton we want to simulate). This directly leads to another, related, characterization of computably enumerable languages, by means of W. Hao dominoes.

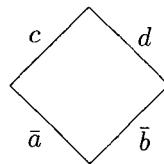


Figure 2.14: A DAE represented as a square.

The initial contents of the tape is represented as suggested in Figure 2.15 (# is the blank symbol). The string $aabab$ written on the tape (part (i) of the figure), is represented by means of DAE blocks as in part (ii), which corresponds to the geometric representation in part (iii). Note the presence of the sticky ends which encode the symbols $\#, a, a, b, a, b, \#$ and which correspond to the edges marked with the same symbols in the geometric

representation.

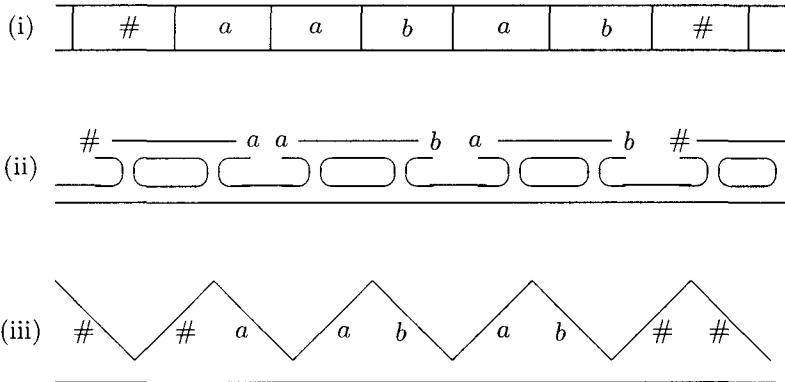


Figure 2.15: The initial contents of the tape.

The rules of the cellular automaton are given by means of DAE structures (hence squares, in our simplification). Such structures are left free in the same solution with the initial tape. Because of the complementarity, these structures will anneal to the corresponding places, which exactly means the simulation of the corresponding instructions. By ligation, these new blocks will start to construct a sort of carpet, where each row represents one stage of the computation (a configuration of the automaton tape). Figure 2.16 shows one step parallel annealing-ligation, starting from the tape described in Figure 2.15 (that is, from the string $\#aabab\#$) and using the following transitions

$$aa \rightarrow ba, \quad ba \rightarrow ab, \quad b\# \rightarrow aa,$$

as well as the “completion rule”

$$\#\# \rightarrow \#\#.$$

We obtain the string $\#baabaa\#$, exactly as in the cellular automaton.

Note the important fact that the next step will use pairs of symbols which are shifted with one position with respect to the previous places where we have used the transitions (this is automatically ensured by the shape of the DAE structures and the shape of the initial description of the tape).

The special stop symbol is encoded by a distinguished sticky end of a DAE structure, which can be immediately recognized during the annealing-ligation process. (Several possibilities are imagined in [294], [295], but we do not enter into details. Just imagine that when this special sticky end appears, it leads to a well prepared reaction which turns the whole solution

to a given colour, say blue. The reader can also find in the mentioned papers various biochemical solutions to several other problems which appear in this framework: how to promote the annealing and the ligation, how to get rid of mismatches, how to stop the reaction in the moment when the stop symbol is introduced, and so on and so forth.)

When the computation stops, the last row of DAE structures in our “carpet” represents the output. (Again, a biochemical problem is how to separate this last row and to read it; details can be found in [294], [295].)

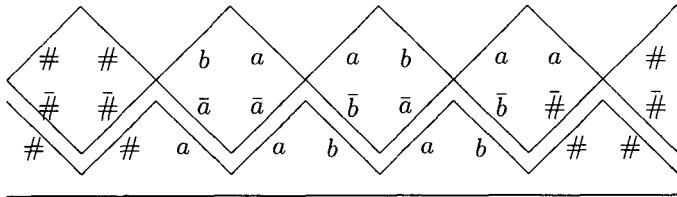


Figure 2.16: Example of a one step computation.

Following [294], we summarize the procedure as follows.

1. Express your problem in terms of blocked cellular automata.
2. Create the initial molecule, which represents the initial contents of the automaton tape.
3. Create the DAE structures associated with the instructions of the cellular automaton.
4. Mix the molecules created in steps 2 and 3 together in a test tube and keep under precise conditions such that the DNA “carpet” crystallize.
5. When the solution “turns blue”, extract the strand with the halting symbol.
6. Read the result by sequencing the strand obtained at step 5.

Since the time when this way of computing by means of annealing was proposed, several experiments were reported, practically proving that small computations can be performed in this manner. Still, we know no computation of a practically significant size done in this way (E. Winfree is quite optimistic in this sense; hopefully, before this book will be printed, a convincing application of DNA computing will be obtained).

Several characteristics of the previous procedure are worth mentioning:

- The computation is fully parallel, first because the cellular automata work in parallel, then because the annealing process develops in parallel. This is very promising from the time complexity point of view.
- However, the amount of DNA involved in a computation is large and not used in a very efficient way: at each moment, the whole history of the computation (all the previous descriptions of the tape) are preserved. In E. Winfree terms: Can some conditions be found such that the bottom of the lattice is dissolving while the top of the lattice is growing? We are aware of no answer to this question.
- Theoretically, the computation itself requires no energy at all, but melting the errors, producing the initial molecules, and, especially, reading the output are energy consuming. This illustrates Ch. Bennett's principle [23], that computation is free, but input and output are costly.

A quite promising – both theoretically and practically – generalization is to work with three dimensional structures. Some details can be found in [295], but we do not approach here this generalization.

2.7 Computing by Carving

The overall strategy of most, if not all, experiments in DNA computing reported or only imagined so far, of first generating a large set of candidate solutions and then removing non-solutions (see again the examples in the previous sections), suggests a mode of computation which is not very usual in computer science or in computability theory (it appears, however, in a few places in other areas, for instance in operations research): produce a “complete data pool” (a set of candidate solutions) and filter it iteratively in such a way that only the solutions of the problem remain. This seems to be a very adequate strategy for DNA computing, because the first phase can be carried out biochemically, in a single step, using the huge parallelism made possible by DNA. In this section, we try to exploit the power of this mode of *computing by carving* in a general framework.

In formal language theory terms, this idea can be formulated in the following quite new way of identifying a language L : start from a superlanguage of L , say¹ M , large and easy to be obtained, then remove from M strings or sets of strings, iteratively, until obtaining L . Contrast this strategy with the usual “positivistic” grammatical approach, where one produces the strings in L , at the end of successful derivations, discarding the “wrong derivations” (derivations not ending with a terminal string), and constantly ignoring/avoiding the complement of L .

In particular, M above can be V^* , the total language over the alphabet V of the language L . Therefore, as a particular case, we can try to produce V^* –

¹From “marble”, to fit the idea of “computing by carving”....

L , the complement of L , and to extract it from V^* . As one knows, the family of computably enumerable languages is not closed under complementation. Therefore, *by carving we can “compute” non-computably enumerable languages!*

This is just another formulation of the fact that the family RE is not closed under complementation. But this reformulation is very natural for DNA computing, because of the following two facts:

1. We can generate any computably enumerable language by using one of the many generative devices based on DNA-type of operations which were already investigated. The 2D annealing procedure presented in the previous section is one of the possibilities. In Sections 2.10 and 2.11 we will discuss several others. In particular, we can generate (in an easy way) the language V^* (the “complete data pool” for our case), or any needed regular language.
2. A procedure to implement the difference of two languages in DNA terms can be easily imagined (in menmnological terms, assuming that the operations are going in the ideal fashion, without errors, mismatching, etc.), and such operations were already used in the experiments mentioned in the previous sections.

Let us imagine a general procedure of this type, for “computing” the difference $L_1 - L_2$ of two given languages L_1, L_2 consisting of DNA double stranded molecules. More specifically, assume that both L_1, L_2 consist of molecules codifying some information in such a way that in the upper strands of molecules one uses only the nucleotides A and C and in the lower strands one uses the complementary nucleotides T and G. (This gives us the possibility to speak about the *upper* and the *lower* strands of a molecule also after denaturing the molecules and separating the strands.) Moreover, let us add an occurrence of T in between each two nucleotides in the upper strand, and hence A in between each two nucleotides in the lower strand. Let us also concatenate the molecules in the two languages with some blocks α and β which never appear as substrings of the strings in L_1, L_2 . (The alternating occurrences of T in the upper strands and of A in the lower strands make this possible.) The language L_1 bounded in this way is placed in a test tube. We separate the strands (by denaturing the DNA) and we remove all lower strand sequences (for instance, we can pour the solution over a solid support having the upper strand of α bound to it; the lower strands will anneal and will remain bound to the solid support, the upper strands can be washed out). We make a copy of the tube. We do the same with the language L_2 , in another test tube, preserving only the lower strands. We merge the two tubes and leave the strands to anneal, then we remove all molecules which are not complete duplexes (having sticky ends or loops, hence nucleotides not paired with a

complement); this can again be done by using solid support techniques as above. What remains is the intersection of the two languages. We melt again and remove the upper strands, then we merge the obtained solution with the copy of the first tube. After annealing, we remove all complete molecules and keep all upper strands which were not paired with their complementary lower strands. This is the difference $L_1 - L_2$ (if needed, by polymerisation/amplification we can return to double stranded molecules).

Of course, from a biochemical point of view, this is a completely idealized procedure (not to speak about the fact that we have tacitly dealt with infinite languages codified as DNA molecules and handled in test tubes, which, for a while are nothing else than finite objects ...). However, this is the same kind of idealization as when passing from a computer to a Turing machine as a computing model, with an infinite tape, working an unbounded number of steps, etc. Important is that the difference and the complementation are “DNA-like operations”.

In sum: we can produce V^* , we can produce any given computably enumerable language L and we can compute the difference $V^* - L$. In this way, we can step outside the family of computably enumerable languages!

The above described procedure can hardly be considered an algorithm, in any broad meaning of the term, hence we cannot infer that in this way we violate the Church–Turing Thesis. (We involve operations between infinite sets, considered as “one step operations”, some of these operations are performed in an analog rather than a symbolic way, etc. Analog super-Turing computations were described also in other places, see, e.g. [271], based on other techniques.) However, in some sense (because of the lack of precise definitions of the involved notions), this is a matter of terminology: what an algorithm is, what the Church–Turing Thesis says, when crossing the barrier of computable enumerability means violating the Church–Turing Thesis? We do not persist in this speculative direction, but we want to look for more precise definitions of the “computing by carving” in formal language theory terms.

In practice, we can always follow a finite number of steps (dealing with finite objects). This could be seen as a drastic decrease of the interest for computing by carving, but we advocate for the opposite conclusion. First, in practice we deal with *instances* of problems, which are always finite; still, in order to solve them in a uniform way, we need to study the general problem, the whole, possibly infinite, family of instances of it. Second, in practice we are often satisfied with an approximation of the solution. An initial sequence of steps from an infinite sequence of approximating steps which lead in a sharp manner to a solution can provide a satisfactory approximation². (Actually, we do not have a good notion of an approximation of a language in our

²There are Michelangelo unfinished statues which are equally impressive and valuable as those which are believed to be finished....

framework; we will discuss this question lately.)

Let us now start a “classic” approach to computing by carving in formal language theory terms. The main goal is to find a way to compute “difficult” languages (beyond Turing, as it was possible in the general framework of the previous discussion), by “as easy as possible” tools.

Identifying a language L by generating first its complement, $H = V^* - L$, and then computing the difference $V^* - H$ is quite a rough procedure. Let us consider a more subtle one, where L is obtained at the end of several difference operations. The “most subtle” case is that when we have an infinite sequence of such operations: we construct V^* , as well as certain languages L_1, L_2, \dots , and we iteratively compute $V^* - L_1, (V^* - L_1) - L_2, \dots$, such that L is obtained at the limit. In total, this means

$$L = V^* - (\cup_{i \geq 1} L_i).$$

We have returned again to the complement of only one language, the union $\cup_{i \geq 1} L_i$. Moreover, the languages $L_i, i \geq 1$, can be rather simple while their union can be arbitrarily complex. For example, take L_i as consisting of the i th string in the lexicographic ordering of a given language, which can be of any complexity in the Chomsky hierarchy. Each L_i is a singleton, the union of these languages can be non-computably enumerable.

The problem is still ill formulated. *The sequence of languages $L_i, i \geq 1$, must be defined in a regular, finite, way.* Here is a proposal:

A sequence L_1, L_2, \dots of languages over an alphabet V is called *regular*³ if L_1 is a regular language and there is a gsm g such that $L_{i+1} = g(L_i)$, for each $i \geq 1$.

Thus, L_1 is given, all other languages are iteratively constructed by applying the gsm g to L_1 . In this way, we can identify the sequence by its generating pair (L_1, g) .

Then, a language $L \subseteq V^*$ is said to be *C-REG computable* if there is a regular sequence of languages, L_1, L_2, \dots , and a regular language $M \subseteq V^*$ such that $L = M - (\cup_{i \geq 1} L_i)$.

Note that when defining C-REG computable languages we also allow the use of a specified initial (regular) language M , which is not necessarily V^* as in the previous discussion.

As in Section 1.6, we denote by $g^*(L_1)$ the iteration of the gsm g over L_1 . Then, we can write $L = M - g^*(L_1)$. We are again back to the complement with respect to M of a single language, $g^*(L_1)$, but this language is the union of a regular sequence of languages (the languages in the sequence are defined by finite tools: a regular grammar – or a finite automaton – for the language L_1 , and the gsm g).

³Please distinguish between “a sequence of regular language” and “a regular sequence of languages”, as defined here. Every regular sequence of languages is a sequence of regular languages, but the converse is not true.

The regular sequences of languages deserve a more detailed study. We present here only a few properties of them.

First, it is easy to see that the union of a regular sequence of languages can be of a high complexity, in spite of the fact that each *finite* union is a regular language. For instance, it is easy to see that the sequence

$$L_i = \{a^{2^i}\}, i \geq 1,$$

is regular: start from $L_1 = \{a^2\}$ and take the gsm which doubles each occurrence of a . However, the language $\{a^{2^i} \mid i \geq 1\}$ is non-context-free.

However, not all computably enumerable languages can be written as the union of a regular sequence. We can find counterexamples by using the following necessary condition, whose proof is omitted:

Lemma 2.1 *If (L_1, g) identifies an infinite language $L = g^*(L_1)$, then there is a constant k such that for each $x \in L$ we can find $y \in L$ such that $|x| < |y| \leq k|x|$.*

For instance, the language

$$L = \{a^{2^n} \mid n \geq 1\}$$

does not have the property in Lemma 2.1, hence it cannot be obtained as the union of a regular sequence of languages. Note that this language is context-sensitive.

On the other hand, the family of languages which can be obtained in this way is not at all a small one – remember Theorem 1.3 from Section 1.7:

Every computably enumerable language $L \subseteq T^$ can be written in the form $L = g^*(\{a_0\}) \cap T^*$, where g is a gsm – depending on L – and a_0 is a fixed symbol not in T .*

This result has an important consequence: because we have

$$V^* - (g^*(\{a_0\}) \cap T^*) = (V^* - g^*(\{a_0\})) \cup (V^* - T^*),$$

and $V^* - T^*$ is a regular language, it follows that $V^* - g^*(\{a_0\})$ is not necessarily a computably enumerable language: if $V^* - (g^*(\{a\}) \cap T^*)$ is not computably enumerable (and this can be the case, because $g^*(\{a_0\}) \cap T^*$ can be any computably enumerable language), then $V^* - g^*(\{a_0\})$ is not computably enumerable either. This statement is worth formulating as a theorem:

Theorem 2.3 *There are C-REG computable languages which are not computably enumerable languages.*

A question appears here in a natural way: How powerful is the computing by carving (when using regular sequences of languages)? The answer is provided by the following result.

Theorem 2.4 A language is C-REG computable if and only if it can be written as the complement of a computably enumerable language.

Proof. Consider a language $L \subseteq T^*$ such that the language $T^* - L$ is computably enumerable. Consider a type-0 Chomsky grammar $G = (N, T, S, P)$ for the language $T^* - L$.

We construct the regular sequence of languages starting with

$$L_1 = \{S\},$$

and using the gsm

$$g = (K, V, V, s_0, \{s_1\}, R),$$

with the following components:

$$\begin{aligned} K &= \{s_0, s_1\} \cup \{[x] \mid x \in (N \cup T)^+, xy \rightarrow v \in P, \\ &\quad \text{for some } y \in (N \cup T)^+\}, \\ V &= N \cup T, \\ R &= \{s_0\alpha \rightarrow \alpha s_0 \mid \alpha \in N \cup T\} \\ &\cup \{s_0\alpha_1 \rightarrow [\alpha_1], [\alpha_1]\alpha_2 \rightarrow [\alpha_1\alpha_2], \dots, \\ &\quad [\alpha_1 \dots \alpha_{i-2}]\alpha_{i-1} \rightarrow [\alpha_1 \dots \alpha_{i-2}\alpha_{i-1}], \\ &\quad [\alpha_1 \dots \alpha_{i-1}]\alpha_i \rightarrow vs_1 \mid \alpha_j \in N \cup T, 1 \leq j \leq i, \\ &\quad i \geq 2, \text{ and } \alpha_1 \dots \alpha_i \rightarrow v \in P\} \\ &\cup \{s_0\alpha \rightarrow vs_1 \mid \alpha \rightarrow v \in P, \alpha \in N\} \\ &\cup \{s_1\alpha \rightarrow \alpha s_1 \mid \alpha \in N \cup T\}. \end{aligned}$$

It is easy to see that $g^*(L_1) \cap T^* = L(G)$ (at each iteration of g one simulates the application of a rule in P).

Therefore, for the regular language $M = T^*$ we obtain $M - g^*(L_1) = T^* - L(G) = T^* - (T^* - L) = L$. In conclusion, L is a C-REG computable language.

Conversely, assume that $L \subseteq T^*$ is a C-REG computable language, hence L can be written in the form $L = M - g^*(L_1)$, for some regular sequence of languages defined by (L_1, g) and a regular language $M \subseteq T^*$. Let us consider the complement of L , that is $T^* - L$. We have

$$T^* - L = T^* - (M - g^*(L_1)) = (T^* - M) \cup g^*(L_1).$$

The language $T^* - M$ is regular (the family of regular languages is closed under difference) and $g^*(L_1)$ is computably enumerable. Consequently, $T^* - L$ is a computably enumerable language. \square

Corollary 2.2 (i) Every context-sensitive language is C-REG computable.
(ii) The family of C-REG computable languages is incomparable with the family of computably enumerable languages.

Proof. Assertion (i) follows from the fact that the family of context-sensitive languages is closed under the complementation.

Assertion (ii) is a consequence of the non-closure of the family of computably enumerable languages under the complementation. \square

We do not know whether or not Theorem 2.4 (and assertion (i) in Corollary 2.2) remains true for other, more restrictive, definitions of a regular sequence of languages. What about restricted forms of gsm mappings? In particular, what about deterministic gsm's?

A related question is dealt with (and solved) below: does the number of states of the gsm used in the definition of regular sequences of languages induce an infinite hierarchy of C-REG computable languages? (The answer will be negative.)

Let us denote by $CREG_n, n \geq 1$, the family of languages of the form $M - g^*(L_1)$, for M, L_1 regular languages and g a gsm with at most n states; by $CREG$ we denote the union of all these families, that is, the family of all C-REG computable languages.

Theorem 2.5 $CREG_1 \subset CREG_2 \subseteq CREG_3 = CREG$.

Proof. The inclusions \subseteq are obvious. According to Theorem 2.4, each language $L \subseteq T^*$, $L \in CREG$, has the complement in RE . Take such a language L and consider a grammar $G = (N, T, S, P)$ for the language $T^* - L$ in the Geffert normal form, that is with $N = \{S, A, B, C\}$ and with P containing context-free rules $S \rightarrow x, x \in (N \cup T)^*$, and the non-context-free rule $ABC \rightarrow \lambda$.

Consider $L_1 = \{S\}$ and the gsm

$$g = (K, V, V, s_0, \{s_0\}, R),$$

with the following components:

$$\begin{aligned} K &= \{s_0, s_A, s_B\} \\ V &= N \cup T, \\ R &= \{s_0\alpha \rightarrow \alpha s_0 \mid \alpha \in N \cup T\} \\ &\cup \{s_0S \rightarrow xs_0 \mid S \rightarrow x \in P\} \\ &\cup \{s_0A \rightarrow s_A, s_A B \rightarrow s_B, s_B C \rightarrow s_0\}. \end{aligned}$$

It is easy to see that $g^*(L_1) \cap T^* = L(G)$ (at each iteration of g one can simulate the application of a rule in P ; if several rules are simulated at the same iteration, then this does not change the generated language). We proceed now as in the proof of Theorem 2.4: we take $M = T^*$ and we obtain $M - g^*(L_1) = L$. In conclusion, $L \in CREG_3$.

The inclusion $CREG_1 \subset CREG_2$ is clearly proper, because by iterating a gsm with only one state we get a language in OL . \square

The previous theorem shows that in order to classify the C-REG computable languages we need other parameters describing the complexity of the regular sequences of languages, the number of states is not sufficiently sensitive. Such more sensitive parameters remain to be found.

Consider again a pair (L_1, g) , identifying a regular sequence of languages, used for computing a language $L \subseteq V^*$ as a difference $L = M - g^*(L_1)$, for some $M \subseteq V^*$. If we stop the carving at any stage n , that is, we compute the language $F_n = M - \bigcup_{i \geq 0}^n g^i(L_1)$, then we obtain a decreasing sequence of languages

$$M \supseteq F_0 \supseteq F_1 \supseteq \dots \supseteq F_n \supseteq F_{n+1} \supseteq \dots \supseteq L.$$

Therefore, F_n can be considered an approximation of L , of a better quality for bigger values of n . However, because M and L_1 are regular, and the family REG is closed under gsm mappings and difference, each language $F_n, n \geq 0$, is a regular one. Therefore, all these approximations are regular, irrespective of the type of the approximated language L . This seems to be a serious drawback to this way of approximating a language, but we have to mention that this happens frequently in grammar inference, when one tries to learn an approximate grammar for an approximation of a language.

From a mathematical point of view, this observation raises the question whether or not there are languages which can be approximated by regular languages in a satisfactory manner. We have to start by an adequate definition of the notion of an approximation.

Here is a natural proposal: for $L \subseteq V^*$ we say that $L' \subseteq V^*$ is a *good regular approximation* of L if $L \subseteq L'$ and there is no regular language $L'' \subseteq V^*$ such that $L \subseteq L'' \subseteq L'$ and $L' - L''$ is infinite.

Note that if we remove a finite number of strings from a regular language, then what we obtain is again regular. Thus, if we only remove finite sets from a given regular approximation L' of a non-regular language L (this clearly implies that $L' - L$ is infinite), then we can do this operation an infinite number of steps. No satisfactory approximation in this sense can exist, that is why we request that $L' - L''$ is infinite.

A good regular approximation of a language can be useful for computing by carving: instead of computing L , we try to compute a good regular approximation L' of L . Because L' is regular, it is possible to find two regular languages M, L_0 and a gsm g such that $L' = F_n$, where F_n is defined as above.

Several problems appear here: Given a language L and a regular approximation L' of it, construct (algorithmically) M, L_1, g such that $L' = F_n$, for some n . (Of course, we look for a nontrivial answer: because L' is regular, we can take $M = L'$ and $L_1 = \emptyset$. Moreover, we need triples M, L_1, g with some convenient properties, which have to be specified in each practical case separately.) Does a good regular approximation exist for each language? If a good

approximation exists for a given language, is it unique? Find algorithmically good regular approximations.

We attack here only one of these problems and prove that there are (linear) languages for which there is no good regular approximation in the sense of the previous definition. This suggests that the definition is not satisfactory (hence the other problems should be approached only after finding such a good definition – if any).

Theorem 2.6 *The language $\{a^n b^n \mid n \geq 1\}$ has no good regular approximation.*

Proof. Denote our language by L (we have $L \in LIN - REG$) and assume that a regular language L' includes this language, $L' \subseteq \{a, b\}^*$.

Consider the pumping lemma for regular languages, in the following form: for each regular language M there is a constant p such that each string $z \in M$ with $|z| \geq p$ can be written in the form $z = uvw$ with $v \neq \lambda$ and $uv^i w \in M$ for all $i \geq 0$; moreover, there are such decompositions of z with $|uv| \leq p$ and, other decompositions, with $|vw| \leq p$. (We can iterate either the first nonterminal symbol which appears for the second time when counting from the beginning of a derivation in a regular grammar for M , or the nonterminal symbol which appears for the second time when counting from the end of a derivation.)

All strings $a^n b^n$ are in L' . We apply the pumping lemma to such strings with $n \geq p$, iterating first a substring of a^n . As a consequence, we find that all strings of the form $a^{n+ik} b^n$, for $i \geq 0$, for a given $k \geq 1$, are in L' . Now, consider a string $a^{n+ik} b^n$ as above and apply the pumping lemma in such a way to iterate a substring of b^n . It follows that all strings of the form

$$a^{p+ik} b^{p+jl},$$

for all $i, j \geq 0$ and given $k, l \geq 1$, are in L' .

Consider now the following regular language:

$$L_{pkl} = \{a^{p+2lik} b^{p+(2j+1)l} \mid i, j \geq 1\}.$$

This language is clearly included in L' and infinite. The language

$$L'' = L' - L_{pkl}$$

is regular (it is the difference of two regular languages) and it includes the language L : no string $a^n b^n$ is in L_{pkl} , hence no string of this form has been removed from L' . Indeed, we cannot have $p + 2lik = p + (2j + 1)l$: after reducing p and simplifying with l we get $2ki = 2j + 1$, which is contradictory. Because $L' - L'' = L_{pkl}$, it follows that L' cannot be a good regular approximation. Because L' was an arbitrary regular language containing L , the theorem is proved. \square

We do not know whether there are non-regular languages for which good regular approximations exist; anyway, this is only a mathematical question. As we have said above, a more significant question is to find an adequate definition of the notion of a good regular approximation. This is a task for further investigations.

2.8 Sticker Systems

The aim of this section is twofold. First, we want to formalize and generalize the basic operation used in many of the experiments described in the previous sections, starting with Adleman's experiment: annealing. While in Adleman's experiment (as well as in other experiments) this operation is explicitly mentioned, in the algorithms described in terms of the "programming language" based on operations of the form of `merge`, `detect`, `select`, etc., the actual DNA operations are no longer mentioned. However, the basic operations are crucial for implementing these "second order operations" `merge`, `detect`, `select`, etc. For instance, in several of the experiments mentioned in the previous sections, the *initial data pool*, whatever this means in each case, was built up by using the annealing operation. A natural question appears here: how powerful this operation is, in terms of the Chomsky hierarchy? The filtering procedure is in all cases principally finite; otherwise stated, a finite number of squeezing operations are performed starting from an initial set of candidate solutions. In most cases, such an operation means a `separate` operation, that is, removing strings which contain certain substrings. Operations of this type preserve the place in the Chomsky hierarchy: the language of the strings which contain certain substrings is regular, hence also its complement – let us denote it by M – is regular; removing from a language L the strings which contain certain substrings means intersecting L with the language M ; the intersection with a regular language preserves the place in the Chomsky hierarchy. This implies that the language of the solutions to a given problem which are given by procedures as above cannot be more complex – in Chomsky's sense – than the initial complete data pool. This implies the need of theoretically investigating the power of annealing.

Another important reason to investigate this operation is related to its close connection with DNA and the new data structure suggested by DNA molecules, the double stranded sequences with the corresponding elements complementary in a given sense. We have seen in Section 2.2 that the double stranded sequences based on complementarity have a sort of intrinsic computational completeness (via characterizations of computably enumerable languages as images of the twin-shuffle language over $\{0, 1\}$ by deterministic sequential transducers).

It is easy to see that annealing itself is not very powerful: the correct construction of double stranded sequences starting from a finite set of single stranded sequences can be controlled with a finite memory (a "window" of

the length equal to the length of the longest single strand is sufficient to ensure the correctness of the process); this means that in this way we get only regular languages. Thus, on the one hand, we have a weak operation, the annealing; on the other hand, we deal with data structures having an in-built computational power. How can this power be used in order to increase the power of annealing? A more precise formulation of this question is: what kind of a computing framework can we devise, using annealing as its basic operation and the double strand as the basic data structure, in such a way that a large enough computational power is obtained, maybe even computational completeness and universality?

We will give below an answer to this question in the form of a generative machinery; also an automata counterpart will be briefly mentioned at the end of this section.

We do not give a complete formalization of the operation we use, but we prefer to introduce it mainly through pictures. Full technical details can be found in Chapter 4 of [224].

Consider an alphabet V and a relation $\rho \subseteq V \times V$ (of *complementarity*) over V .

Intuitively, the complementarity is a symmetric relation, so we assume that ρ is symmetric. (The Watson–Crick complementarity is also injective, but we do not impose here this property.)

Using the relation ρ we can define the set of all double stranded sequences over V as follows. Denote

$$\begin{aligned} \left[\begin{smallmatrix} V \\ V \end{smallmatrix} \right]_{\rho} &= \left\{ \left[\begin{smallmatrix} a \\ b \end{smallmatrix} \right] \mid a, b \in V, (a, b) \in \rho \right\}, \\ WK_{\rho}(V) &= \left[\begin{smallmatrix} V \\ V \end{smallmatrix} \right]_{\rho}^*. \end{aligned}$$

The set $WK_{\rho}(V)$ is called the *Watson–Crick domain* associated with the alphabet V and the complementarity relation ρ . The elements $\left[\begin{smallmatrix} a_1 \\ b_1 \end{smallmatrix} \right] \left[\begin{smallmatrix} a_2 \\ b_2 \end{smallmatrix} \right] \cdots \left[\begin{smallmatrix} a_n \\ b_n \end{smallmatrix} \right] \in WK_{\rho}(V)$ are also written in the form $\left[\begin{smallmatrix} w_1 \\ w_2 \end{smallmatrix} \right]$, for $w_1 = a_1 a_2 \dots a_n, w_2 = b_1 b_2 \dots b_n$. For a sequence (sometimes we also say *molecule*) $\left[\begin{smallmatrix} w_1 \\ w_2 \end{smallmatrix} \right] \in WK_{\rho}(V)$, we say that the two component strings, w_1, w_2 , are *strands*; w_1 is the *upper strand* and w_2 is the *lower strand*.

We emphasize the two properties characterizing the elements $\left[\begin{smallmatrix} w_1 \\ w_2 \end{smallmatrix} \right]$ of $WK_{\rho}(V)$, because they are essential for the models considered below:

- the two strands w_1, w_2 are of the same length,
- the corresponding symbols in the two strands are complementary in the sense of the relation ρ .

We shall also use below “incomplete molecules,” that is, elements of the set

$$W_\rho(V) = L_\rho(V) \cup R_\rho(V) \cup LR_\rho(V),$$

where

$$\begin{aligned} L_\rho(V) &= (\binom{\lambda}{V^*} \cup \binom{V^*}{\lambda}) \left[\begin{matrix} V \\ V \end{matrix} \right]_\rho^*, \\ R_\rho(V) &= \left[\begin{matrix} V \\ V \end{matrix} \right]_\rho^* (\binom{\lambda}{V^*} \cup \binom{V^*}{\lambda}), \\ LR_\rho(V) &= (\binom{\lambda}{V^*} \cup \binom{V^*}{\lambda}) \left[\begin{matrix} V \\ V \end{matrix} \right]_\rho^+ (\binom{\lambda}{V^*} \cup \binom{V^*}{\lambda}). \end{aligned}$$

(The notation $\binom{L_1}{L_2}$, for two languages L_1, L_2 , denotes the set $\{\binom{x}{y} \mid x \in L_1, y \in L_2\}$. When one of these languages is a singleton, say $L_1 = \{x\}$, then we write $\binom{x}{L_2}$ instead of $\binom{\{x\}}{L_2}$. Note the essential difference between $\left[\begin{matrix} x \\ y \end{matrix} \right]$ and $\binom{x}{y}$: in the latter case we have no reference to the complementarity relation, there is no relation between the symbols appearing in x and those appearing in y ; $\binom{x}{y}$ is just a pair of strings, written one over the other, while $\left[\begin{matrix} x \\ y \end{matrix} \right]$ is a molecule, with x, y having the two properties mentioned above.)

The possible shapes of elements in $W_\rho(V)$ are illustrated in Figure 2.17. In all cases, we have a well-formed double stranded sequence x and overhangs y, z in one or two sides of x . These overhangs (sticky ends) can be placed in the upper strand or in the lower one. Note that in the case of $L_\rho(V)$ and $R_\rho(V)$, the block x may be empty, but in the elements of $LR_\rho(V)$ the block $x \in$ contains at least one element $\left[\begin{matrix} a \\ b \end{matrix} \right]$ with $(a, b) \in \rho$. In turn, the overhangs can also be empty; what remains is then an element of $WK_\rho(V)$, therefore $WK_\rho(V)$ is included in each set $L_\rho(V), R_\rho(V), LR_\rho(V)$.

Any element of $W_\rho(V)$ which contains at least a position $\left[\begin{matrix} a \\ b \end{matrix} \right]$, $a \neq \lambda$, $b \neq \lambda$, is called a *well-started double stranded sequence*; of course, when several “columns” $\left[\begin{matrix} a \\ b \end{matrix} \right]$, with $(a, b) \in \rho$, appear, they appear consecutively. In general, the elements of $W_\rho(V)$ are also called *dominoes*.

Among the elements of $W_\rho(V)$ we can define a partial operation, modelling the annealing operation: a well-started molecule (hence a sequence having at least a position filled in both of the two strands) x can be prolonged to the right or to the left with a domino y , providing that the sticky ends match, that is, they are complementary in the corresponding positions. The result should always be a well-started molecule, hence a sequence which does not have empty places surrounded by symbols from V .

These eight possible cases when a molecule x is prolonged to the right with a molecule y are indicated in Figure 2.18. The operation is denoted by $\mu(x, y)$.

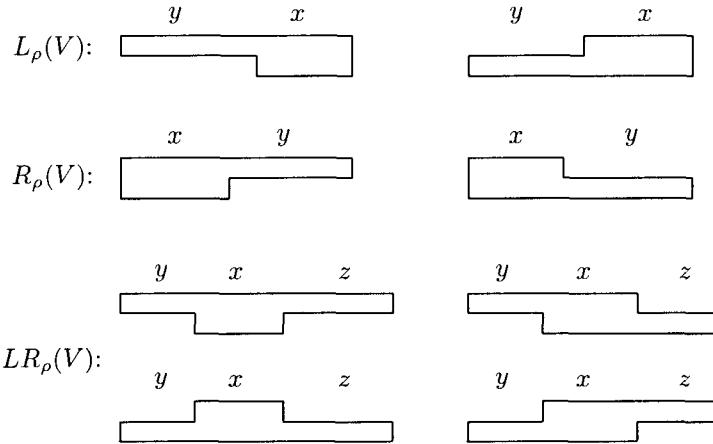


Figure 2.17: Possible shapes of “dominoes”.

In the symmetric way we can define $\mu(y, x)$, the prolongation of a well-started molecule x , by a sequence y , to the left. Note that we do not need to distinguish the “left prolongation” from the “right prolongation” by denoting them in different ways: in any case at least one of the terms of the operation must be a well-started molecule and the result – it is a well-started molecule, too – entirely depends on the order of the two sequences and on their sticky ends.

Note that in cases 3 and 6 we do not use annealing (hence the complementarity relation). The maximal length of an overhang in a sequence $z \in W_\rho(V)$ is also called the *delay* of z and it is denoted by $d(z)$; it represents the delay in completing the two strands with symbols in V . (Hence, in cases 1, 2 in Figure 2.18, the “right delay” of x and the “left delay” of y should coincide when y is also a well-started double stranded sequence.)

In the same way as rewriting is the underlying operation for Chomsky grammars, the sticking operation is the underlying one for *sticker systems* introduced in [149], and then investigated in [222], [105], [224]. The reader is advised to have in mind the discussion in Section 1.3 about a computability framework: we follow here the strategy discussed in Section 1.3, with particularizations to the sticking operation.

Free sticking (annealing) is presumably weak as it cannot produce more than regular languages, so we define here the sticker systems in a controlled variant: when building molecules, we start from well-started sequences and prolong them in both directions, to the left and to the right at the same time, synchronously. As we shall see, this restriction is able to lead to characteri-

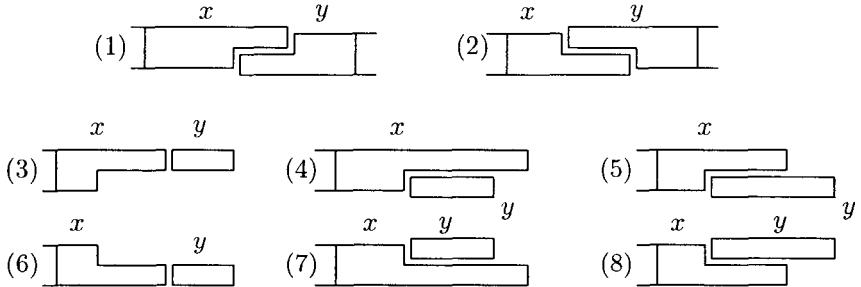


Figure 2.18: The sticking operation.

zations of computably enumerable languages.

A *sticker system* is a construct

$$\gamma = (V, \rho, A, D),$$

where V is an alphabet, $\rho \subseteq V \times V$ is a symmetric relation, A is a finite subset of $LR_\rho(V)$, and D is a finite subset of $W_\rho(V) \times W_\rho(V)$.

The elements of A are called *axioms*. Starting from these axioms and using the pairs (u, v) of dominoes in D , we can obtain a set of double stranded sequences in $WK_\rho(V)$, hence complete molecules, by using the operation μ of sticking.

Formally, for a given sticker system $\gamma = (V, \rho, A, D)$ and two sequences $x, y \in LR_\rho(V)$, we write

$$x \implies y \text{ iff } y = \mu(u, \mu(x, v)), \text{ for some } (u, v) \in D.$$

A sequence $x_1 \implies x_2 \implies \dots \implies x_k$, with $x_1 \in A$, is called a *computation* in γ . A computation $x_1 \implies^* x_k$ is *complete* when $x_k \in WK_\rho(V)$ (no sticky end is present in the last sequence).

The set of all molecules over V produced at the end of complete computations in γ is denoted by $LM(\gamma)$ (LM stands for “language of molecules”):

$$LM(\gamma) = \{w \in WK_\rho(V) \mid x \implies^* w, x \in A\}.$$

In what follows we consider the sticker systems as generating languages of strings. To this aim, we associate with $LM(\gamma)$ the language

$$L(\gamma) = \{x \in V^* \mid \begin{bmatrix} x \\ x' \end{bmatrix} \in LM(\gamma) \text{ for some } x' \in V^*\}.$$

We say that $L(\gamma)$ is the language *generated* by γ .

Several variants of sticker systems are of interest. A system $\gamma = (V, \rho, A, D)$ is said to be:

- *one-sided*, if for each pair $(u, v) \in D$ we have either $u = \lambda$ or $v = \lambda$,
- *regular*, if for each pair $(u, v) \in D$ we have $u = \lambda$,
- *simple*, if all pairs $(u, v) \in D$ have either $u, v \in \binom{V^*}{\lambda}$, or $u, v \in \binom{\lambda}{V^*}$.

In one-sided systems, the prolongation to the left is independent of the prolongation to the right; in regular systems we only prolong the sequences to the right (hence the axioms must be of the form x_1x_2 , with $x_1 \in WK_\rho(V)$ and $x_2 \in \binom{V^*}{\lambda} \cup \binom{\lambda}{V^*}$). In a computation in a simple sticker system we add symbols only to one of the two strands.

We denote by *ASL* the family of languages of the form $L(\gamma)$, for γ a sticker system of an arbitrary form (*SL* stands for “sticker language” and *A* indicates the use of sticker systems of “arbitrary forms”). When only sticker systems which are one-sided, regular, simple, simple and one-sided, or simple and regular are used, we replace *A* in front of *SL* by *O*, *R*, *S*, *SO*, *SR*, respectively. We stress the fact that these families contain *string languages*, not languages of molecules, hence we can discuss their relationships with families in the Chomsky hierarchy without further precautions.

We give here complete proofs for the two basic results about sticker systems: systems without a control on the operation (under the form of a synchronized prolongation to the left and to the right) generate only regular languages, while sticker systems of the general form (in fact, simple systems suffice) characterize the computably enumerable languages modulo a weak coding.

Theorem 2.7 $OSL = RSL = REG$.

Proof. ($OSL \subseteq REG$) Consider a one-sided sticker system $\gamma = (V, \rho, A, D)$. Let us denote by d the length of the longest sticky end or of the longest single stranded sequence appearing in A or in the pairs of D .

We construct the context-free grammar

$$G = (N, T, S, P),$$

where

$$\begin{aligned} N &= \{\langle \binom{u}{\lambda} \rangle_l, \langle \binom{u}{\lambda} \rangle_r, \langle \binom{\lambda}{u} \rangle_l, \langle \binom{\lambda}{u} \rangle_r \mid u \in V^*, 0 \leq |u| \leq d\} \\ &\quad \cup \{S\}, \\ T &= \left[\begin{matrix} V \\ V \end{matrix} \right]_\rho, \end{aligned}$$

and P contains the following rules:

1. $S \rightarrow \langle \binom{u_1}{u_2} \rangle_l \left[\begin{matrix} x_1 \\ x_2 \end{matrix} \right] \langle \binom{v_1}{v_2} \rangle_r$, for $\binom{u_1}{u_2} \left[\begin{matrix} x_1 \\ x_2 \end{matrix} \right] \binom{v_1}{v_2} \in A$, with $\binom{u_1}{u_2}, \binom{v_1}{v_2} \in \binom{\lambda}{V^*} \cup \binom{V^*}{\lambda}$ and $\left[\begin{matrix} x_1 \\ x_2 \end{matrix} \right] \in WK_\rho(V)$.

2. $\langle \left(\begin{smallmatrix} u_1 \\ u_2 \end{smallmatrix} \right) \rangle_l \rightarrow \langle \left(\begin{smallmatrix} u'_1 \\ u'_2 \end{smallmatrix} \right) \rangle_l \left[\begin{smallmatrix} w_1 \\ w_2 \end{smallmatrix} \right]$, where $\left(\begin{smallmatrix} u_1 \\ u_2 \end{smallmatrix} \right), \left(\begin{smallmatrix} u'_1 \\ u'_2 \end{smallmatrix} \right) \in \left(\begin{smallmatrix} \lambda \\ V^* \end{smallmatrix} \right) \cup \left(\begin{smallmatrix} V^* \\ \lambda \end{smallmatrix} \right)$,

$\left[\begin{smallmatrix} w_1 \\ w_2 \end{smallmatrix} \right] \in WK_\rho(V)$, and there is a pair in D of the form

$(\left(\begin{smallmatrix} u'_1 \\ u'_2 \end{smallmatrix} \right) \left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix} \right] \left(\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix} \right), \left(\begin{smallmatrix} \lambda \\ \lambda \end{smallmatrix} \right))$ with $\left(\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix} \right) \in \left(\begin{smallmatrix} V^* \\ \lambda \end{smallmatrix} \right) \cup \left(\begin{smallmatrix} \lambda \\ V^* \end{smallmatrix} \right)$,

and $\left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix} \right] \in WK_\rho(V)$, such that $\left[\begin{smallmatrix} x_1 y_1 u_1 \\ x_2 y_2 u_2 \end{smallmatrix} \right] = \left[\begin{smallmatrix} w_1 \\ w_2 \end{smallmatrix} \right]$.

(We prolong the sequence to the left, using the pairs with an empty right hand member, in accordance with the sticky end; we remember which is the sticky end by means of the nonterminal $\langle \left(\begin{smallmatrix} u_1 \\ u_2 \end{smallmatrix} \right) \rangle_l$; the subscript l stands for “left”. Note that $\left[\begin{smallmatrix} w_1 \\ w_2 \end{smallmatrix} \right]$ above can be equal to $\left[\begin{smallmatrix} \lambda \\ \lambda \end{smallmatrix} \right]$.)

3. $\langle \left(\begin{smallmatrix} u_1 \\ u_2 \end{smallmatrix} \right) \rangle_r \rightarrow \left[\begin{smallmatrix} w_1 \\ w_2 \end{smallmatrix} \right] \langle \left(\begin{smallmatrix} u'_1 \\ u'_2 \end{smallmatrix} \right) \rangle_r$, where $\left(\begin{smallmatrix} u_1 \\ u_2 \end{smallmatrix} \right), \left(\begin{smallmatrix} u'_1 \\ u'_2 \end{smallmatrix} \right) \in \left(\begin{smallmatrix} \lambda \\ V^* \end{smallmatrix} \right) \cup \left(\begin{smallmatrix} V^* \\ \lambda \end{smallmatrix} \right)$,

$\left[\begin{smallmatrix} w_1 \\ w_2 \end{smallmatrix} \right] \in WK_\rho(V)$, and there is a pair in D of the form

$(\left(\begin{smallmatrix} \lambda \\ \lambda \end{smallmatrix} \right), \left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix} \right] \left(\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix} \right) \left(\begin{smallmatrix} u'_1 \\ u'_2 \end{smallmatrix} \right))$, with $\left(\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix} \right) \in \left(\begin{smallmatrix} V^* \\ \lambda \end{smallmatrix} \right) \cup \left(\begin{smallmatrix} \lambda \\ V^* \end{smallmatrix} \right)$,

and $\left[\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix} \right] \in WK_\rho(V)$, such that $\left[\begin{smallmatrix} u_1 x_1 y_1 \\ u_2 x_2 y_2 \end{smallmatrix} \right] = \left[\begin{smallmatrix} w_1 \\ w_2 \end{smallmatrix} \right]$.

(The same idea as above, but prolonging the sequence to the right.)

4. $\langle \left(\begin{smallmatrix} \lambda \\ \lambda \end{smallmatrix} \right) \rangle_l \rightarrow \lambda$,

$\langle \left(\begin{smallmatrix} \lambda \\ \lambda \end{smallmatrix} \right) \rangle_r \rightarrow \lambda$.

(When no sticky end is present, we can finish the derivation.)

It is easy to see that $L(G) = LM(\gamma)$: because we only use one-sided pairs in order to build sequences, the operation of prolonging sequences to the left is independent of the operation of prolonging sequences to the right and conversely; consequently, we can always use that pair $(\left(\begin{smallmatrix} z_1 \\ z_2 \end{smallmatrix} \right), \left(\begin{smallmatrix} \lambda \\ \lambda \end{smallmatrix} \right))$ or $(\left(\begin{smallmatrix} \lambda \\ \lambda \end{smallmatrix} \right), \left(\begin{smallmatrix} z_1 \\ z_2 \end{smallmatrix} \right))$ from D which sticks to the existing overhanging ends, which means that the overhanging ends are not longer than those already existing in A or in D . Thus, the nonterminals in N can control the process in the same way as the sticky ends do this.

In the grammar G there is no derivation of the form $X \Rightarrow^* uXv$ with both u and v being non-empty strings. Consequently ([113], Exercise 9, page 55), the language $L(G)$ is regular. Because $L(G) = LM(\gamma)$ and $L(\gamma)$ is a coding of $LM(\gamma)$, we also have $L(\gamma) \in REG$.

($REG \subseteq RSL$) Consider a finite automaton $M = (K, V, s_0, F, \delta)$ with $K = \{s_0, s_1, \dots, s_k\}$, $n \geq 0$. We construct the regular sticker system

$$\gamma = (V, \rho, A, D),$$

with

$$\rho = \{(a, a) \mid a \in V\},$$

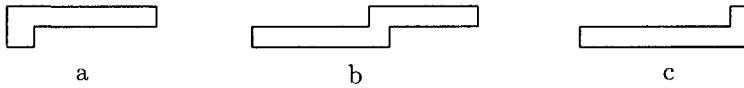


Figure 2.19: Dominoes used in the proof of inclusion $REG \subseteq RSL$.

$$\begin{aligned}
 A &= \left\{ \begin{bmatrix} x \\ x \end{bmatrix} \mid x \in L(M), |x| \leq k+2 \right\} \\
 &\cup \left\{ \begin{bmatrix} x \\ x \end{bmatrix} \binom{u}{\lambda} \mid |xu| = k+2, |x| \geq 1, |u| = i, \text{ for } \right. \\
 &\quad \left. 1 \leq i \leq k+1 \text{ such that } s_0 xu \xrightarrow{*} xus_{i-1} \right\}, \\
 D &= \left\{ \left(\binom{\lambda}{\lambda}, \binom{\lambda}{v} \begin{bmatrix} x \\ x \end{bmatrix} \binom{u}{\lambda} \right) \mid 1 \leq |v| \leq k+1, |xu| = k+2, |x| \geq 1, \right. \\
 &\quad \left| u \right| = i, \text{ for } 1 \leq i \leq k+1, \text{ such that } s_j xu \xrightarrow{*} xus_{i-1}, \right. \\
 &\quad \left. \text{and } j = |v| - 1 \right\} \\
 &\cup \left\{ \left(\binom{\lambda}{\lambda}, \binom{\lambda}{v} \begin{bmatrix} x \\ x \end{bmatrix} \right) \mid 1 \leq |v| \leq k+1, 1 \leq |x| \leq k, \text{ and } \right. \\
 &\quad \left. s_j x \xrightarrow{*} xs_f, s_f \in F, \text{ where } j = |v| - 1 \right\}.
 \end{aligned}$$

The idea is to start with a domino of the form shown in Figure 2.19a, to iteratively use dominoes of the form shown in Figure 2.19b, and to end the computation with a domino of the form shown in Figure 2.19c.

The overhangs codify the states of M by their lengths. The axioms in A which are not already in $WK_\rho(V)$ and the dominoes of the form in Figure 2.19b appearing in the right hand member of pairs in D have overhangs of lengths i , $1 \leq i \leq k+1$, which identify the state s_{i-1} by the length i . This state is reached by M when receiving the string in the upper strand of the well-started molecule which is obtained using the domino. All dominoes of the forms in Figure 2.19b and Figure 2.19c have a non-empty left overhang, hence a molecule in $WK_\rho(V)$ cannot be prolonged. Thus, after using a domino of type c, the computation must stop. Since the system γ has a delay of at most $k+1$, we have $L(\gamma) = L(M)$, which completes the proof of the inclusion $REG \subseteq RSL$.

The inclusion $RSL \subseteq OSL$ follows from the definitions. \square

Completing the previous result, it is interesting to note that $REG - SOSL \neq \emptyset$, but each regular language can be written as the coding of a language in the family $SRSL$.

In short, one sided sticker systems cannot go beyond the power of finite automata. The annealing operation in the “natural” form, without any further constraints than the complementarity, is rather weak from a computational point of view. In contrast, the pairing of dominoes increases the power at the level of Turing machines (modulo a weak coding).

Theorem 2.8 Every language $L \in RE$ can be written in the form $L = h(L')$, where h is a weak coding and $L' \in SSL$.

Proof. Consider a language $L \subseteq T^*$, $L \in RE$. According to Theorem 1.5, there exist two λ -free morphisms $h_1, h_2 : V_1^* \rightarrow V_2^*$, a regular language $R \subseteq V_2^*$ and a projection $pr_T : V_2^* \rightarrow T^*$ for $T \subseteq V_2$, such that $L = pr_T(h_1(EQ(h_1, h_2)) \cap R)$.

Consider a deterministic finite automaton $M = (K, V_2, s_0, F, \delta)$ recognizing the language R .

We construct the simple sticker system

$$\gamma = (V, \rho, A, D),$$

with

$$\begin{aligned} V &= V_2 \cup \bar{V}_2 \cup K \cup \{\$\$, E, E', C, Z\}, \\ \rho &= \{(X, X) \mid X \in V\}, \\ A &= \left\{ \binom{s_0}{\lambda} \left[\begin{matrix} \$ \\ \$ \end{matrix} \right] \left(\begin{matrix} Z \\ \lambda \end{matrix} \right) \right\}, \end{aligned}$$

and D contains the following pairs of dominoes:

- For every $a \in V_1$ such that $h_1(a) = b_1 \dots b_k, k \geq 1$, and $h_2(a) = c_1 \dots c_m, m \geq 1$, with $b_1, \dots, b_k, c_1, \dots, c_m \in V_2$, and for $s_{i_j} \in K$, $0 \leq j \leq m$, such that $\delta(s_{i_j}, c_i) = s_{i_{j+1}}, 0 \leq j \leq m$, we introduce in D the pair

$$\left(\binom{s_{i_m} \bar{c}_m s_{i_{m-1}} \dots s_{i_2} \bar{c}_2 s_{i_1} s_{i_0} \bar{c}_1}{\lambda} \right), \left(\binom{b_1 CZ b_2 CZ \dots CZ b_k CZ}{\lambda} \right).$$

(To the left of $\left[\begin{matrix} \$ \\ \$ \end{matrix} \right]$ we produce the reversed image of some $h_2(a)$, for $a \in V_1$, and at the same time we guess a valid path through M over $h_2(a)$: $s_{i_0} c_1 c_2 \dots c_m \Rightarrow^* s_{i_m}$. To the right we produce the image of a through h_1 , with the symbols of $h_1(a)$ separated by the auxiliary symbols CZ .)

- $\left(\binom{E' s_f}{\lambda} \right), \left(\binom{E}{\lambda} \right)$, for $s_f \in F$.

(The recognition of the string in the upper strand of the left part of the sequence by means of M is finished correctly.)

- $\left(\binom{\lambda}{ss} \right), \left(\binom{\lambda}{Z} \right)$, for all $s \in Q$.

(These rules check the correct continuation of the recognition path through M : if $s_1 x \Rightarrow^* x s_2$ is followed by $s_3 y \Rightarrow^* y s_4$, then we must have $s_2 = s_3$, otherwise the complementarity is not observed when using the block $\left(\binom{\lambda}{ss} \right)$.)

4. $(\left(\begin{array}{c} \lambda \\ b \end{array}\right), \left(\begin{array}{c} \lambda \\ bC \end{array}\right))$, for $b \in V_2$.

(The string of symbols \bar{b} generated to the left of $\left[\begin{array}{c} \$ \\ \$ \end{array}\right]$ in the upper strand is compared with the string of symbols b generated to the right of $\left[\begin{array}{c} \$ \\ \$ \end{array}\right]$ in the upper strand. Note that the symbols Z are “consumed” together with the pairs of states, by rules of type 3; now we also “consume” the symbols C introduced in the upper strand, to the right of $\left[\begin{array}{c} \$ \\ \$ \end{array}\right]$.)

5. $(\left(\begin{array}{c} \lambda \\ E' \end{array}\right), \left(\begin{array}{c} \lambda \\ E \end{array}\right))$.

(Only in this way we can get a complete molecule.)

From the explanations above, one can see that the complete molecules produced by γ are of the form

$$\left[\begin{array}{l} E's_f s_f \bar{c}_t s_t s_t \dots \bar{c}_2 s_1 s_1 \bar{c}_1 s_0 s_0 \\ E's_f s_f \bar{c}_t s_t s_t \dots \bar{c}_2 s_1 s_1 \bar{c}_1 s_0 s_0 \end{array} \right] \left[\begin{array}{c} \$ \\ \$ \end{array} \right] \left[\begin{array}{l} Zb_1 CZb_2 CZ \dots CZb_t CZE \\ Zb_1 CZb_2 CZ \dots CZb_t CZE \end{array} \right],$$

for

$$c_1 c_2 \dots c_t = h_1(w) = h_2(w) = b_1 b_2 \dots b_t,$$

for some $w \in V_1^*$, and $s_0 c_1 \dots c_{t+1} \Rightarrow^* c_1 \dots c_{t+1} s_f$ in M for $s_f \in F$, hence $h_1(w) \in R$.

No complete computation can be continued, because the upper strands of dominoes (in groups 1 and 2) have one state only in the left end of the left domino, whereas the lower strands of dominoes (in groups 3, 4, 5) have either two states or a symbol \bar{b} , $b \in T$, or the symbol E' in that position.

Consider now the weak coding (in fact, a projection) h defined by

$$\begin{aligned} h(a) &= a, \text{ for } a \in T, \\ h(\bar{a}) &= \lambda, \text{ for } a \in T, \\ h(s) &= \lambda, \text{ for } s \in Q, \\ h(E) &= h(E') = h(\$) = h(C) = h(Z) = \lambda. \end{aligned}$$

Clearly, we get $L = h(L(\gamma))$, which completes the proof. \square

The use of the weak coding in the previous characterization cannot be avoided: in a sticker system of any type we always increase the current (incomplete) molecule, that is, the obtained language is context-sensitive (the work of a sticker system can be simulated by a monotonous grammar). In order to obtain more than context-sensitive languages, we need erasing.

The situation exhibited by the previous two theorems illustrates a very important point: a natural operation (the annealing here), used in a free manner (as in one sided sticker systems), can lead only to regular languages; when a control on this operation is imposed (in our case, the pairing of dominoes used at each step), the power of the considered mechanism is spectacularly

increased. Badly enough, the control is not “natural”, it should be implemented in a way which is not known from nature. In the case above, there is no realistic way of imposing that a molecule is prolonged simultaneously to the right and to the left with dominoes in a pair from a given set of pairs. The biochemistry ensures computation at the level of finite automata, but in order to get more computational power we have to wait for a new technology, able to implement our restrictions on the operations we use. For a while, we have to look for problems which can be solved at the level of finite automata, by *ad hoc* methods (remember that at this level we do not have good universality results), a programmable DNA computer based on the annealing operation (in the form of a sticker system) does not seem realizable in this moment.

A similar conclusion can be drawn also for other computability frameworks inspired by DNA biochemistry, in particular, for those based on the splicing operation (see Sections 2.9 – 2.11).

We close this presentation of sticker systems by pointing out that there is one more way of characterizing the computably enumerable languages (modulo a weak coding) by means of sticker systems, namely, by considering *coherent* computations. In fact, simple regular sticker systems are used. The idea is the following. Let us write a simple regular sticker system in the form $\gamma = (V, \rho, A, D_l, D_u)$, where V, ρ, A are as above, D_l is the set of lower dominoes (they are single strands which are placed in the lower strand of the molecules we build), and D_u are upper dominoes. Consider that both the dominoes in D_l and those in D_u are labelled with elements in the same set M . Assume that in a computation in γ one uses the sequence $x_{i_1}, x_{i_2}, \dots, x_{i_k}$ of dominoes from D_u and the sequence $y_{i_1}, y_{i_2}, \dots, y_{i_r}$ of dominoes from D_l . If $k = r$ and the labels of $x_{i_1}, x_{i_2}, \dots, x_{i_k}$ are the same as the labels of $y_{i_1}, y_{i_2}, \dots, y_{i_k}$, then we say that the computation is *coherent*.

A theorem from [149] (see also Theorem 4.12 in [224]) says that each computably enumerable language is the weak coding of a language generated by a simple regular sticker system by means of coherent computations.

Again, the control we consider increases considerably the power in comparison to the power of non-controlled systems (remember that the systems are at the same time simple and regular, hence of the most restricted type of sticker systems), but the control is rather similar to that involved in bidirectional systems: dominoes used at a possibly large distance (actually, arbitrarily large, otherwise again a finite state machine can supervise the process) should have identical labels, which is not at all very realistic at this moment.

Let us now remember the way of “reading” a DNA molecule as proposed in Section 2.2, in such a way to obtain the twin-shuffle language over $\{0, 1\}$, having also in mind the characterization of computably enumerable languages as images of $TS_{\{0,1\}}$ by (deterministic) sequential transducers. This immediately suggests considering a kind of automata, with the tape of the form

of a double stranded sequence, with two reading heads (playing the role of the two “insects” mentioned in Section 2.2) which scan the two strands from left to right, controlled by a finite memory. We obtain a variant of a finite automaton as illustrated in Figure 2.20, which are the automata counterpart of simple regular sticker systems.

We call such an automaton a *Watson–Crick finite automaton*. Formally, such a machinery is defined as a construct

$$M = (K, V, \rho, s_0, F, \delta),$$

where K and V are disjoint alphabets, $\rho \subseteq V \times V$ is a symmetric relation, $s_0 \in K$, $F \subseteq K$, and $\delta : K \times \binom{V^*}{V^*} \rightarrow \mathcal{P}(K)$ is a mapping such that $\delta(s, \binom{x}{y}) \neq \emptyset$ only for finitely many triples $(s, x, y) \in K \times V^* \times V^*$.

The elements of K are called *states*, V is the (input) alphabet, ρ is a complementarity relation on V , s_0 is the initial state, F is the set of final states, and δ is the transition mapping. The interpretation of $s' \in \delta(s, \binom{x_1}{x_2})$ is: in state s , the automaton passes over x_1 in the upper level strand and over x_2 in the lower level strand of a double stranded sequence, and enters the state s' .

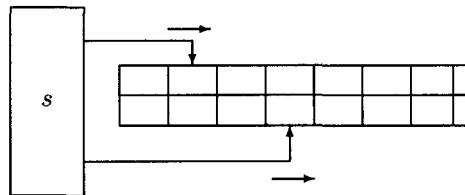


Figure 2.20: A Watson–Crick finite automaton.

A transition in a Watson–Crick finite automaton can be defined as follows: For $\binom{x_1}{x_2}, \binom{u_1}{u_2}, \binom{w_1}{w_2} \in \binom{V^*}{V^*}$ such that $\binom{x_1 u_1 w_1}{x_2 u_2 w_2} \in WK_\rho(V)$ and $s, s' \in K$, we write

$$\binom{x_1}{x_2} s \binom{u_1}{u_2} \binom{w_1}{w_2} \Rightarrow \binom{x_1}{x_2} \binom{u_1}{u_2} s' \binom{w_1}{w_2} \text{ iff } s' \in \delta(s, \binom{u_1}{u_2}).$$

We denote by \Rightarrow^* the reflexive and transitive closure of the relation \Rightarrow .

As in the case of sticker systems, we consider here the language of strings appearing in the upper strands of Watson–Crick tapes recognized by our automata, that is the language

$$L(M) = \{w_1 \in V^* \mid s_0 \left[\begin{matrix} w_1 \\ w_2 \end{matrix} \right] \Rightarrow^* \left[\begin{matrix} w_1 \\ w_2 \end{matrix} \right] s_f, \text{ for } s_f \in F\},$$

$$\text{and } w_2 \in V^*, \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \in WK_\rho(V)\}.$$

Informally speaking, we start with the two heads positioned on the left-most symbols in each strand, in the initial state, and we move the heads to the right according to the transitions specified by δ . The string in the upper strand is recognized if the two heads reach the rightmost symbols in the corresponding strands and the automaton enters a final state. We stress the fact that we deal with complete molecules; the two strings written in the two strands of the tape have equal lengths.

From the construction above and the discussion we have started with, the following result is rather expected:

Theorem 2.9 *Each computably enumerable language is the projection of a language recognized by a Watson–Crick finite automaton.*

Characterizations of RE can be obtained also by using restricted classes of Watson–Crick finite automata, for instance, with all moves of the form $s \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} s'$ where $|x_1 x_2| = 1$ (at each move we scan a symbol, either in the upper strand or in the lower one), with only one state, with all states being final, etc. In these restricted cases, more powerful squeezing devices are necessary (morphisms and/or intersections with regular languages), a projection is no longer sufficient.

Taking into account the opposite directionality of the two strands of a DNA molecule, we can consider a variant of a Watson–Crick finite automaton which starts from a configuration as that in Figure 2.21: at the beginning of a computation, the upper reading head is placed in the first cell of the upper strand while the lower head is placed in the rightmost cell of the lower strand. These automata are called *reverse Watson–Crick finite automata*. Using such automata, we get characterizations of RE of the same form as in the case of the basic form of automata.

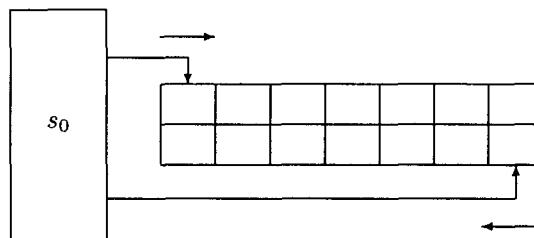


Figure 2.21: A reverse Watson–Crick finite automaton.

Finally, we observe that the projection involved in the statement of Theorem 2.9 (more generally, any morphism) can be simulated by means of an

output mapping, which leads to considering Watson–Crick sequential transducers, of the form illustrated in Figure 2.22. They are nothing else than a mixture of a Watson–Crick automaton and a gsm: the transition mapping δ also specifies an output (as in gsm’s).

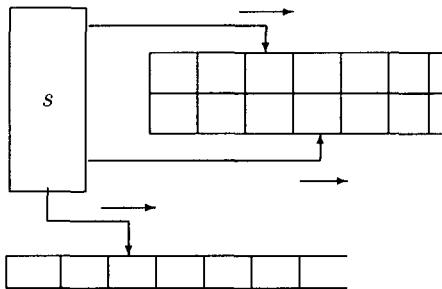


Figure 2.22: A Watson–Crick transducer.

It is easy to see that for each computably enumerable language $L \subseteq V^*$ there is an alphabet V' , a complementarity relation ρ over V' , and a Watson–Crick transducer g_L such that $L = g_L(WK_\rho(V'))$.

Further details about Watson–Crick automata can be found in [107], [106], as well as in the chapter of [224] devoted to this topic.

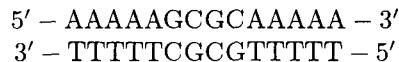
2.9 Extended H Systems

We have mentioned in Section 2.3 the recombination operation, where two DNA molecules are cut by restriction enzymes in such a way that matching sticking ends are produced and the obtained fragments are pasted together (under the effect of a ligase) such that possibly new molecules are formed. This operation can be used as a starting point of a computability framework. After [127], where a formal model of the recombination was considered, we call this operation *splicing* and the computing models based on it are called *H systems*.

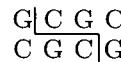
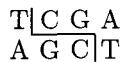
In this section we introduce the uncontrolled H systems and we give the two basic results about their generative power, the regularity lemma and the universality lemma. Controlled and distributed H systems will be discussed in the subsequent sections (all of them characterize the computably enumerable languages, so they are possible models for DNA “computers” based on the splicing operation).

We first trace the abstraction process from the cut and paste operation carried out by restriction enzymes and ligases to the formal operation of splicing (we follow the style of [129] and [224], Section 7.1).

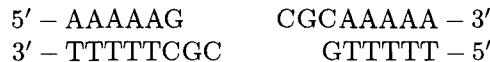
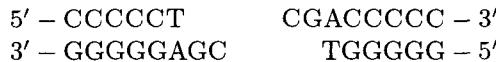
We start from an example. Consider the following two DNA molecules



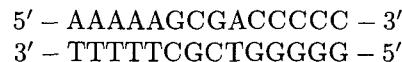
as well as the restriction enzymes *TaqI* and *SciNI*, which recognize, respectively, the following patterns



We have also indicated the way of cutting the DNA molecules. Thus, when acting on the two molecules mentioned above, these enzymes will produce the following four fragments:



We have obtained molecules with identical sticky ends, therefore the four fragments can be bound together, either restoring the initial molecules, or producing new molecules by recombination. The recombination gives the following new molecules



Because of the precise Watson–Crick complementarity, we can consider the operation above as acting on single stranded sequences (hence on strings). For instance, the two molecules we have started with are precisely identified by the strings

CCCCCTCGACCCCC ,

AAAAAGCGCAAAAAA ,

respectively, with the convention that they represent a strand of a DNA molecule read in the 5' to 3' direction.

In what concerns the pattern recognized by a restriction enzyme, it can be described by a triple (u, x, v) , of strings over the alphabet $\{\text{A, C, G, T}\}$, with

the following meaning: (u, v) is the context where the cutting takes place and x is the overhanging sequence. In the case of the two enzymes above we have the triples:

$$(T, CG, A), \quad (G, CG, C).$$

Thus, the operation can be formalized as follows: having two strings w_1, w_2 and two triples $(u_1, x_1, v_1), (u_1, x_2, v_2)$, such that

$$\begin{aligned} w_1 &= w'_1 u_1 x_1 v_1 w''_1, \\ w_2 &= w'_2 u_2 x_2 v_2 w''_2, \end{aligned}$$

we allow the recombination operation only when $x_1 = x_2$, and the strings obtained in this way are

$$\begin{aligned} z_1 &= w'_1 u_1 x v_2 w''_2, \\ z_2 &= w'_2 u_2 x v_1 w''_1, \end{aligned}$$

where $x = x_1 = x_2$.

Tacitly, we have made here one more generalizing step, by working with strings over an arbitrary alphabet.

In order to get the most general operation with strings, modelling the previously described one, we have to advance three more steps.

First, we put together pairs of triples (u, x, v) and we start directly from pairs $((u_1, x, v_1), (u_2, x, v_2))$ (which will be considered “splicing rules”, governing the operation).

Secondly, when having a pair $((u_1, x, v_1), (u_2, x, v_2))$ and two strings w_1, w_2 as above, $w_1 = w'_1 u_1 x v_1 w''_1$ and $w_2 = w'_2 u_2 x v_2 w''_2$, we can consider only the string $z_1 = w'_1 u_1 x v_2 w''_2$ as a result of the recombination, because the string $z_2 = w'_2 u_2 x v_1 w''_1$ is the result of the one-output-recombination with respect to the symmetric pair, $((u_2, x, v_2), (u_1, x, v_1))$.

Thirdly, instead of pairs of triples $((u_1, x, v_1), (u_2, x, v_2))$ as above, we can consider pairs of pairs: the passing from $w_1 = w'_1 u_1 x v_1 w''_1$, $w_2 = w'_2 u_2 x v_2 w''_2$ to $z_1 = w'_1 u_1 x v_2 w''_2$ with respect to $((u_1, x, v_1), (u_2, x, v_2))$ is equivalent to the passing from $w_1 = w'_1 u'_1 v_1 w''_1$, $w_2 = w'_2 u'_2 v_2 w''_2$ to $z_1 = w'_1 u'_1 v_2 w''_2$ with respect to $((u'_1, v_1), (u'_2, v_2))$, where $u'_1 = u_1 x$ and $u'_2 = u_2 x$. Similarly, we can consider the quadruple $((u_1, x v_1), (u_2, x v_2))$.

Altogether, we are led to the following operation with strings over an alphabet V : a quadruple $(u_1, u_2; u_3, u_4)$, of strings over V , is called a *splicing rule*; with respect to such a rule r , for $x, y, z \in V^*$ we write

$$\begin{aligned} (x, y) \vdash_r z &\quad \text{iff} \quad x = x_1 u_1 u_2 x_2, \\ &\quad y = y_1 u_3 u_4 y_2, \\ &\quad z = x_1 u_1 u_4 y_2, \\ &\quad \text{for some } x_1, x_2, y_1, y_2 \in V^*. \end{aligned}$$

We say that we *splice* x, y at the *sites* u_1u_2, u_3u_4 , respectively, and the result is z . This is the basic operation we shall deal with in this chapter.

When understood from the context, we omit the specification of r and write \vdash instead of \vdash_r .

Of course, if we want to bring our models back to laboratory, we have to renounce to all the aforementioned abstraction steps, going back to considering specified enzymes, with specified recognition patterns, but we will not enter here such details.

Because we place our discussion at the most general level, we will formalize the splicing rules as strings, in the following natural way. Consider an alphabet V and two special symbols, $\#, \$$, not in V . A *splicing rule* (over V) is a string of the form

$$r = u_1 \# u_2 \$ u_3 \# u_4,$$

where $u_1, u_2, u_3, u_4 \in V^*$.

The maximal length of strings u_1, u_2, u_3, u_4 is called the *radius* of the splicing rule $r = u_1 \# u_2 \$ u_3 \# u_4$.

The passing from x, y to z , via \vdash_r , can be represented as shown in Figure 2.23.

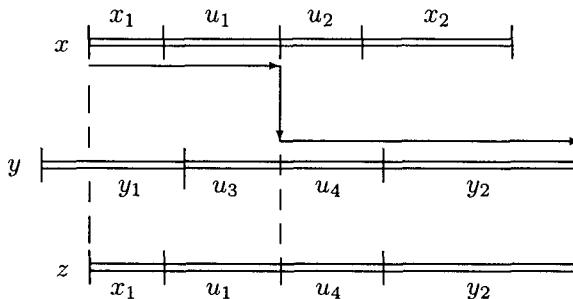


Figure 2.23: The splicing operation.

For a better readability, in many cases we shall indicate by a vertical bar the place where the terms of the splicing are cut, in the style:

$$(x_1 u_1 | u_2 x_2, y_1 u_3 | u_4 y_2) \vdash_r x_1 u_1 u_4 y_2,$$

for $r = u_1 \# u_2 \$ u_3 \# u_4$.

We step now from this operation to building a computability device.

An *H scheme* is a pair

$$\sigma = (V, R),$$

where V is an alphabet and $R \subseteq V^* \# V^* \$ V^* \# V^*$ is a set of splicing rules.

Note that R can be infinite and that we can consider its place in the Chomsky hierarchy, or in another classification of languages. In general, if

$R \in FL$, for a given family of languages, FL , then we say that the H scheme σ is of FL type.

For a given H scheme $\sigma = (V, R)$ and a language $L \subseteq V^*$, we define

$$\sigma(L) = \{z \in V^* \mid (x, y) \vdash_r z, \text{ for some } x, y \in L, r \in R\}.$$

When some restriction enzymes and a ligase are present in a test tube, they do not stop acting after one cut and paste operation, but they act iteratively. Accordingly, for an H scheme $\sigma = (V, R)$ and a language $L \subseteq V^*$ we define

$$\begin{aligned}\sigma^0(L) &= L, \\ \sigma^{i+1}(L) &= \sigma^i(L) \cup \sigma(\sigma^i(L)), \quad i \geq 0,\end{aligned}$$

and

$$\sigma^*(L) = \bigcup_{i \geq 0} \sigma^i(L).$$

Consequently, $\sigma^*(L)$ is the closure of L under the splicing with respect to σ , i.e. the smallest language L' which contains L , and is closed under the splicing with respect to σ , that is to say, $\sigma(L') \subseteq L'$.

For two families of languages, FL_1, FL_2 , we define

$$H(FL_1, FL_2) = \{\sigma^*(L) \mid L \in FL_1 \text{ and } \sigma = (V, R) \text{ with } R \in FL_2\}.$$

The basic results in this area, of crucial importance for DNA computing based on splicing, are the following two. Because of their importance, we give (almost) complete proofs.

Lemma 2.2 (The Regularity Preserving Lemma) $H(REG, FIN) \subseteq REG$.

Proof. Let $L \subseteq V^*$ be a regular language recognized by a finite automaton $M = (K, V, s_0, F, \delta)$. Consider also an H scheme $\sigma = (V, R)$ with a finite set $R \subseteq V^* \# V^* \$ V^* \# V^*$. Assume that $R = \{r_1, \dots, r_n\}$ with $r_i = u_{i,1} \# u_{i,2} \$ u_{i,3} \# u_{i,4}, 1 \leq i \leq n, n \geq 1$. Moreover, assume that $u_{i,1} u_{i,4} = a_{i,1} a_{i,2} \dots a_{i,t_i}$, for $a_{i,j} \in V, 1 \leq j \leq t_i, t_i \geq 0, 1 \leq i \leq n$. For each $i, 1 \leq i \leq n$, consider the new states $q_{i,1}, q_{i,2}, \dots, q_{i,t_i}, q_{i,t_i+1}$. Denote their set by K' and consider the finite automaton

$$M_0 = (K \cup K', V, s_0, F, \delta_0),$$

where

$$\begin{aligned}\delta_0(s, a) &= \delta(s, a), \text{ for } s \in K, a \in V, \\ \delta_0(q_{i,j}, a_{i,j}) &= \{q_{i,j+1}\}, \quad 1 \leq j \leq t_i, 1 \leq i \leq n.\end{aligned}$$

We construct a sequence of finite automata (with λ transitions) $M_k = (K \cup K', V, s_0, F, \delta_k)$, $k \geq 1$, starting from M_0 , by passing from M_k to M_{k+1} , $k \geq 0$, in the following way.

Consider each splicing rule $r_i = u_{i,1}\#u_{i,2}\$u_{i,3}\#u_{i,4}$, $1 \leq i \leq n$.

If s is a state in $K \cup K'$ such that

1. $q_{i,1} \notin \delta_k(s, \lambda)$,
2. there is $s_1 \in K \cup K'$ and $x_1, x_2 \in V^*$ such that

$$\begin{aligned}s &\in \delta_k(s_0, x_1), \\ s_1 &\in \delta_k(s, u_{i,1}u_{i,2}), \\ \delta_k(s_1, x_2) \cap F &\neq \emptyset,\end{aligned}$$

(therefore, $x_1u_{i,1}u_{i,2}x_2 \in L(M_k)$), then we put

$$\delta_{k+1}(s, \lambda) = \{q_{i,1}\}.$$

We say that this is an *initial transition* of level $k + 1$.

Moreover, if s' is a state in $K \cup K'$ such that

1. $s' \notin \delta_k(q_{i,t_i+1}, \lambda)$,
2. there is $s_1 \in K \cup K'$ and $y_1, y_2 \in V^*$ such that

$$\begin{aligned}s_1 &\in \delta_k(s_0, y_1), \\ s' &\in \delta_k(s_1, u_{i,3}u_{i,4}), \\ \delta_k(s', y_2) \cap F &\neq \emptyset,\end{aligned}$$

(therefore, $y_1u_{i,3}u_{i,4}y_2 \in L(M_k)$), then we put

$$\delta_{k+1}(q_{i,t_i+1}, \lambda) = \{s'\}.$$

We say that this is a *final transition* of level $k + 1$.

Then, δ_{k+1} is the extension of δ_k with the initial and final transitions of level $k + 1$, with respect to all splicing rules in R and all states s, s' in $K \cup K'$.

As the set of states is fixed, the above procedure stops after at most $2 \cdot n \cdot \text{card}(K \cup K')$ steps, that is, there is an integer m such that $M_{m+1} = M_m$.

We shall prove that $L(M_m) = \sigma^*(L)$.

Since $\sigma^*(L)$ is the smallest language containing L and closed under the splicing with respect to σ , it is enough to prove that

- (i) $L \subseteq L(M_m)$,
- (ii) $L(M_m)$ is closed under the splicing with respect to σ ,
- (iii) $L(M_m) \subseteq \sigma^*(L)$.

Point (i) is obvious from the construction of the automaton M_m .

In order to prove point (ii), let us consider a splicing rule $r_i = u_{i,1}\#u_{i,2}\$u_{i,3}\#u_{i,4}$ in R and two strings $x, y \in L(M_m)$ such that $x = x_1u_{i,1}u_{i,2}x_2, y = y_1u_{i,3}u_{i,4}y_2$. There are two states $s_1, s_2 \in K \cup K'$ such that

$$\begin{aligned}s_1 &\in \delta_m(s_0, x_1), \quad \delta_m(s_1, u_{i,1}u_{i,2}x_2) \cap F \neq \emptyset, \\ s_2 &\in \delta_m(s_0, y_1u_{i,3}u_{i,4}), \quad \delta_m(s_2, y_2) \cap F \neq \emptyset.\end{aligned}$$

From the construction of M_m we have

$$q_{i,1} \in \delta_m(s_1, \lambda) \text{ and } s_2 \in \delta_m(q_{i,t_i+1}, \lambda).$$

This implies that $x_1u_{i,1}u_{i,4}y_2 \in L(M_m)$. The situation is illustrated in Figure 2.24. Consequently, $\sigma(L(M_m)) \subseteq L(M_m)$.

By an induction argument which we omit here, one can prove that each string recognized by M_m can be produced by iterated splicing with respect to σ starting from strings in L , and this concludes the proof. \square

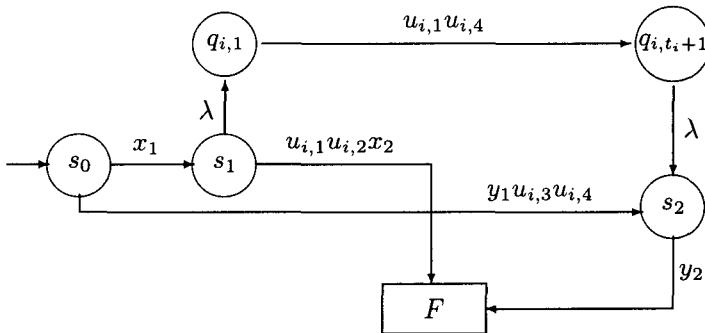


Figure 2.24: Simulating a splicing in M_m .

Thus, by using a finite set of splicing rules we get regular languages only. Making the smallest step forward (in the Chomsky hierarchy), that is, using a regular set of splicing rules, leads to a maximal jump in generative power: we obtain a characterization of computably enumerable languages, modulo a filtering operation in the form of an intersection with a regular language:

Lemma 2.3 (The Basic Universality Lemma) *Every language $L \in CE, L \subseteq T^*$, can be written in the form $L = L' \cap T^*$ for some $L' \in H(FIN, REG)$.*

Proof. Consider a type-0 grammar $G = (N, T, S, P)$, denote $U = N \cup T \cup \{B\}$, where B is a new symbol, and construct the H scheme

$$\sigma = (V, R),$$

where

$$V = N \cup T \cup \{X, X', B, Y, Z\} \cup \{Y_\alpha \mid \alpha \in U\},$$

and R contains the following groups of rules:

- | | | |
|--------------------|---------------------------------|---|
| <i>Simulate</i> : | 1. $Xw\#uY\$Z\#vY,$ | for $u \rightarrow v \in P, w \in U^*,$ |
| <i>Rotate</i> : | 2. $Xw\#\alpha Y\$Z\#Y_\alpha,$ | for $\alpha \in U, w \in U^*,$ |
| | 3. $X'\alpha\#Z\$X\#wY_\alpha,$ | for $\alpha \in U, w \in U^*,$ |
| | 4. $X'w\#Y_\alpha\$Z\#Y,$ | for $\alpha \in U, w \in U^*,$ |
| | 5. $X\#Z\$X'\#wY,$ | for $w \in U^*,$ |
| <i>Terminate</i> : | 6. $\#ZY\$XB\#wY,$ | for $w \in U^*,$ |
| | 7. $\#Y\$XZ\#.$ | |

Consider also the language

$$\begin{aligned} L_0 = & \{XBSY, ZY, XZ\} \\ & \cup \{ZvY \mid u \rightarrow v \in P\} \\ & \cup \{ZY_\alpha, X'\alpha Z \mid \alpha \in U\}. \end{aligned}$$

We obtain $L = \sigma^*(L_0) \cap T^*$.

Let us examine the work of σ , namely the possibilities to obtain a string in T^* .

No string in L_0 is in T^* . All rules in R involve a string containing the symbol Z , but this symbol will not appear in the string produced by splicing. Therefore, at each step we have to use a string in L_0 and, excepting the case of using the string $XBSY$ in L_0 , a string produced at a previous step.

The symbol B is a marker for the beginning of the sentential forms of G simulated by σ .

By rules in group 1 we can simulate the rules in P . Rules in groups 2–5 move symbols from the right hand end of the current string to the left hand end, thus making possible the simulation of rules in P at the right hand end of the string produced by σ . However, because B is always present and marks the place where the string of G begins, we know at each moment which is that string. Namely, if the current string in σ is of the form $\beta_1 w_1 B w_2 \beta_2$, for some β_1, β_2 markers of types X, X', Y, Y_α with $\alpha \in U$, and $w_1, w_2 \in (N \cup T)^*$, then $w_2 w_1$ is a sentential form of G .

We start from $XBSY$, hence from the axiom of G , marked to the left hand with B and bracketed by X, Y .

Let us see how the rules of types 2–5 work. Take a string $Xw\alpha Y$, for some $\alpha \in U, w \in U^*$. By a rule of type 2 we get

$$(Xw|\alpha Y, Z|Y_\alpha) \vdash XwY_\alpha.$$

The symbol Y_α memorizes the fact that α has been erased from the right hand end of $w\alpha$. No rule in R can be applied to XwY_α , excepting the rules of type 3:

$$(X'\alpha|Z, X|wY_\alpha) \vdash X'\alpha wY_\alpha.$$

Note that the same symbol α removed at the previous step is now added in the front of w . Again we have only one way to continue, namely by using a rule of type 4. We get

$$(X'\alpha w|Y_\alpha, Z|Y) \vdash X'\alpha w Y.$$

If we use now a rule of type 7, removing Y , then X' (and B) can never be removed, the string cannot be turned to a terminal one. We have to use a rule of type 5:

$$(X|Z, X'|\alpha w Y) \vdash X\alpha w Y.$$

We have started from $Xw\alpha Y$ and have obtained $X\alpha w Y$, a string with the same end markers. We can iterate these steps as long as we want, so any circular permutation of the string between X and Y can be produced. Moreover, what we obtain are exactly the circular permutations and nothing more (for instance, at every step we still have one and only one occurrence of B).

To every string XwY we can also apply a rule of type 1, providing w ends with the left hand member of a rule in P . Any rule of P can be simulated in this way, at any place we want in the corresponding sentential form of G , by preparing the string as above, using rules in groups 2–5.

Consequently, for every sentential form w of G there is a string $XBwY$, produced by σ , and, conversely, if Xw_1Bw_2Y is produced by σ , then w_2w_1 is a sentential form of G .

The only way to remove the symbols not in U from the strings produced by σ is by using rules in groups 6, 7. More precisely, the symbols XB can be removed only if Y is present (hence the work is blocked if we use first rule 7, removing Y : the string cannot participate to any further splicing, and it is not terminal) and the symbol B is in the left hand position. After removing X and B we can remove Y , too, and what we obtain is a string in U^* . From the previous discussion, it is clear that if such a string is from T^* , then it is in $L(G)$, hence $\sigma^*(L_0) \cap T^* \subseteq L(G)$. Conversely, each string in $L(G)$ can be produced in this way, hence $L(G) \subseteq \sigma^*(L_0) \cap T^*$. We have the equality $L(G) = \sigma^*(L_0) \cap T^*$, which completes the proof. \square

Many variants of the rotate-and-simulate procedure from the previous proof are used in the proofs of the results in the following sections.

The intersection with T^* in the Basic Universality Lemma suggests the definition of *extended H systems*, which are quadruples

$$\gamma = (V, T, A, R),$$

where V is an alphabet, $T \subseteq V$, $A \subseteq V^*$, and $R \subseteq V^* \# V^* \$ V^* \# V^*$, where $\#, \$$ are special symbols not in V .

We call V the alphabet of γ , T is the *terminal* alphabet, A is the set of *axioms*, and R the set of splicing rules.

The *language generated* by γ is defined by

$$L(\gamma) = \sigma^*(A) \cap T^*,$$

where $\sigma = (V, R)$ is the underlying H scheme of γ .

For two families of languages, FL_1, FL_2 , we denote by $EH(FL_1, FL_2)$ the family of languages $L(\gamma)$ generated by extended H systems $\gamma = (V, T, A, R)$, with $A \in FL_1, R \in FL_2$.

The following counterpart of the Regularity Preserving Lemma can be easily proved.

Lemma 2.4 $REG \subseteq EH(FIN, FIN)$.

The known results about the generative power of extended H systems are summarized in Table 2.1, where at the intersection of the row marked with FL_1 with the column marked with FL_2 there appear either the family $EH(FL_1, FL_2)$, or two families FL_3, FL_4 such that $FL_3 \subset EH(FL_1, FL_2) \subseteq FL_4$. These families FL_3, FL_4 are the best possible estimations among the six families considered here.

Table 2.1: The generative power of extended H systems.

	<i>FIN</i>	<i>REG</i>	<i>LIN</i>	<i>CF</i>	<i>CS</i>	<i>CE</i>
<i>FIN</i>	<i>REG</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>
<i>REG</i>	<i>REG</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>
<i>LIN</i>	<i>LIN, CF</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>
<i>CF</i>	<i>CF</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>
<i>CS</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>
<i>CE</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>	<i>CE</i>

Thus, the only family which is not equal to a family in the Chomsky hierarchy is $EH(LIN, FIN)$.

2.10 Controlled H Systems

Two of the relations summarized in Table 2.1 are central for the DNA computability based on splicing:

1. $EH(FIN, FIN) = REG$,
2. $EH(FIN, REG) = CE$.

The first one says that finite “computers” based on splicing are weak, they only equal the power of finite automata; remember that at this level we do not have satisfactory notions and results of universality. On the other hand, using a non-finite regular set of splicing rules is not practically possible. The price we pay for getting computational completeness (and universality: because the proof of the Basic Universality Lemma is constructive, if we start from

a universal type-0 grammar G , then we get an equivalent universal extended H system γ) is too large. We have to look for ways of improving the power of H systems with finite components. A suggestion comes from the regulated rewriting area in formal language theory, where the power of context-free grammars is increased by imposing restrictions on the use of rules. This is exactly what we will do in this section and the result is rather promising: “weak” controls on using the splicing rules are enough in order to obtain characterizations of computably enumerable languages by means of finite H systems.

Actually, suggestions on how to obtain controls of this type comes from the very proof of Lemma 2.3. Indeed, examining this proof, one can see that the set of splicing rules is infinite because of the appearance of substrings w in rules of types 1, 2, 3, 4, 5, 6. However, these substrings contain no information, they are arbitrary strings over the alphabet $N \cup T \cup \{B\}$. The role of these substrings w is to allow information to be obtained about the symbol appearing behind them, namely X, X' in the left hand end of a term of the splicing and $Y, Y_\alpha, \alpha \in N \cup T \cup \{B\}$, in the right hand end of the other term of the splicing. Otherwise stated, we have in fact a finite set of splicing rules, applied only to strings containing (at their ends) certain symbols, from well specified sets. This suggests considering the following type of H systems with controlled splicing.

An extended H system *with permitting contexts* is a quadruple

$$\gamma = (V, T, A, R),$$

where V is an alphabet, $T \subseteq V$, A is a finite language over V , and R is a finite set of triples of the form $p = (r; C_1, C_2)$, with $r = u_1 \# u_2 \$ u_3 \# u_4$ being a splicing rule over V and C_1, C_2 being subsets of V .

For $x, y, z \in V^*$ and $p \in R, p = (r; C_1, C_2)$, we define $(x, y) \vdash_p z$ if and only if $(x, y) \vdash_r z$, every element of C_1 appears in x and every element of C_2 appears in y ; when $C_1 = \emptyset$ or $C_2 = \emptyset$, then no condition on x , respectively y , is imposed.

The language generated by γ is defined in the natural way:

$$L(\gamma) = \sigma^*(A) \cap T^*,$$

where $\sigma = (V, R)$ is the underlying H scheme with permitting context rules.

We denote by $EH([n], p[m])$, $n, m \geq 1$, the family of languages $L(\gamma)$ generated by extended H systems with permitting contexts, $\gamma = (V, T, A, R)$, with $\text{card}(A) \leq n$ and $\text{rad}(R) \leq m$, where $\text{rad}(R)$ is the maximal radius of splicing rules r in triples $(r; C_1, C_2)$ from R . When no restriction on the number of axioms or on the maximal radius is considered (but, of course, these numbers are still finite), we replace $[n]$ or $[m]$, respectively, by FIN .

The following lemma can be obtained by rephrasing the proof of Lemma 2.3 in terms of extended H systems with permitting contexts; the details are left to the reader.

Lemma 2.5 $CE \subseteq EH(FIN, pFIN)$.

Actually, a stronger result can be proved, with either the number of axioms or the radius of the splicing rules bounded (by small values):

Theorem 2.10 $CE = EH([1], pFIN) = EH(FIN, p[2])$.

A proof can be found in [224]. It is not known whether or not the two parameters can be simultaneously bounded. (Note that the usual argument leading to a bound on the parameters describing an H system – starting from a universal type-0 grammar we get a fixed equivalent H system, also universal – does not work in the cases when axioms are counted, because the axioms are modified according to the particular system simulated by the universal one.)

Consider now again the proof of Lemma 2.3. The symbols whose presence is checked are elements of the set of control symbols

$$Q = \{X, X', Y\} \cup \{Y_\alpha \mid \alpha \in N \cup T \cup \{B\}\}.$$

The *presence* of a symbol is equivalent with the *absence* of all other symbols, because the control symbols are always present at the ends of the strings – except when finishing the generation of a terminal string. Thus, we can consider a dual variant of extended H systems with permitting contexts, that is, systems with forbidding contexts.

An extended H system *with forbidding contexts* is a quadruple $\gamma = (V, T, A, R)$, where V is an alphabet, $T \subseteq V$ (the terminal alphabet), A is a finite language over V (axioms), and R is a finite set of triples (we call them rules with forbidding contexts) of the form $p = (r; D_1, D_2)$, where $r = u_1 \# u_2 \$ u_3 \# u_4$ is a splicing rule over V and D_1, D_2 are subsets of V .

For $x, y, z, w \in V^*$ and $p \in R, p = (r; D_1, D_2)$, we define $(x, y) \vdash_p z$ if and only if $(x, y) \vdash_r z$, no element of D_1 appears in x and no element of D_2 appears in y ; when $D_1 = \emptyset$ or $D_2 = \emptyset$, then no condition on x , respectively y , is imposed.

From a biochemical point of view, the permitting contexts can be interpreted as catalysts or promoters, favouring the splicing by the associated splicing rule, while the forbidding contexts can be interpreted as inhibitors, suppressing the associated splicing rule.

The pair $\sigma = (V, R)$ is called an (underlying) H scheme with forbidding contexts rules. The language generated by γ is defined in the usual way:

$$L(\gamma) = \sigma^*(A) \cap T^*.$$

We denote by $EH([n], f[m])$, $n, m \geq 1$, the family of languages $L(\gamma)$ generated by extended H systems with forbidding contexts, $\gamma = (V, T, A, R)$, with $\text{card}(A) \leq n$ and $\text{rad}(R) \leq m$, where $\text{rad}(R)$ is the maximal radius of splicing rules r in triples $(r; D_1, D_2)$ in R .

As expected from the previous discussion, we have the following result, stronger than that obtained in the case of permitting contexts: both parameters are bounded.

Theorem 2.11 $EH([1], f[2]) = CE$.

The control through forbidding symbols, can also be implemented by considering a *priority relation* on the set of splicing rules.

Specifically, an *ordered* extended H system is a construct $\gamma = (V, T, A, R, >)$, where V is an alphabet, $T \subseteq V$ (the terminal alphabet), A is a finite language over V (axioms), R is a finite set of splicing rules over V , and $>$ is a partial order relation on R .

For $x, y, z \in V^*$ and $r \in R$ we allow the relation $(x, y) \vdash_r z$ only if we do not have $(x, y') \vdash_{r'} z'$ for some $y', z' \in V^*$ and $r' \in R$ such that $r' > r$. (A splicing is performed by a rule which is maximal among all splicing rules which can be applied to the first string and any other second string.)

We denote by $EH([n], ord[m])$, $n, m \geq 1$, the family of languages generated by ordered extended H systems with at most n axioms and of radius at most m . We get the following counterpart of the previous theorem:

Theorem 2.12 $CE = EH([1], ord[2])$.

The order relation in systems as above can be interpreted as modelling the difference between the reactivity of the enzymes involved in the splicing rules: when two different enzymes can cut the same string, the more reactive one will actually work.

Another way of regulating the splicing can be achieved by *target languages*, which correspond to another biochemical aspect, encountered *in vivo*: nature selects the offsprings of the evolutionary process in a rather dramatic manner, not allowing the perpetuation of “unsuitable” forms of life.

An extended H system with *local targets* is a construct $\gamma = (V, T, A, R)$, where V is an alphabet, $T \subseteq V$ (the terminal alphabet), A is a finite language over V (axioms), and R is a finite set of pairs $p = (r, Q_p)$, where $r = u_1 \# u_2 \$ u_3 \# u_4$ is a splicing rule over V and Q_p is a regular language over V . For $x, y, z \in V^*$ and $p = (r, Q_p)$ in R we write $(x, y) \vdash_p z$ if and only if $(x, y) \vdash_r z$ and $z \in Q_p$ (the result of the splicing with respect to r belongs to Q_p).

If, for such an extended H system with local targets $\gamma = (V, T, A, R)$, we have $Q_{p_1} = Q_{p_2}$ for all $p_1 = (r_1, Q_{p_1}), p_2 = (r_2, Q_{p_2})$ in R , then we say that γ is a system with a *global target*. If Q is the common target language of rules in R , then we write the system in the form $\gamma = (V, T, A, R', Q)$, with R' consisting of the splicing rules in R .

As usual, we denote by $EH([n], lt[m])$, $n, m \geq 1$, the family of languages generated by extended H systems with local targets having at most n axioms

and splicing rules of radius at most m ; in the case of global targets we replace lt by gt .

As expected, we have:

Theorem 2.13 $CE = EH([1], gt[2]) = EH([1], lt[2]).$

We can relate the target language control also to the style of genetic algorithms area, formulating the conditions about splicing operations by means of *fitness mappings*: consider a mapping assessing the quality (fitness, reactivity) of the strings and let us control the process by asking that strings with a low degree of fitness are not used in further splicings.

Proofs of all the results mentioned above and further results of this type (including other ways of controlling the splicing operation in order to increase the power of H systems) can be found in [224].

We now pass to considering another idea, which is not of a type investigated in the “classic” formal language theory, that of using multisets for controlling the splicing operation.

Again a characterization of computably enumerable languages can be obtained. The initial proof of this result (see also Lemma 8.13 in [224]) is not of a rotate-and-simulate type, but a proof of this type is possible also in this case. We give here such a proof – following [206] –, so we present this variant of H systems in a more detailed manner.

In the definition of splicing operations used in this and in the previous section, after splicing two strings x, y we may use again x or y as a term of a splicing, these strings are not consumed by splicing; moreover, we may splice strings from one “generation” with strings from another “generation”. Also the new strings obtained by splicing are supposed to appear in an arbitrary number of copies each.

This assumption, that whenever a string is available then arbitrarily many copies of it are available, is realistic in the sense that, usually, a large number of copies of each string are used whenever a string is used. Moreover, producing a large number of copies of a DNA sequence is easily feasible by amplification through PCR techniques. However, the existence of several copies of each string raises the difficult problem of controlling the splicing so as to prevent “wrong” operations. For instance, after cutting several copies of a string x into fragments x_1, x_2 and modifying (part of the copies of) x_1, x_2 to some x'_1, x'_2 , the test tube will contain strings of all four forms, x_1, x_2, x'_1, x'_2 ; it might be possible to recombine x_1 with x'_2 or x'_1 with x_2 in such a way as to obtain illegal strings which “look like” legal strings x_1x_2 or $x'_1x'_2$.

A possibility to avoid this difficulty is to use at least some of the strings in a specified number of copies and to keep track of these numbers during the work of the system. This leads to the idea of an H system working with multisets.

An *extended μ H system* is a quadruple

$$\gamma = (V, T, A, R),$$

where V is an alphabet, $T \subseteq V$ (the terminal alphabet), A is a multiset over V^+ with $\text{supp}(A)$ finite (axioms), and R is a finite set of splicing rules over V .

For such a μ H system and two multisets M_1, M_2 over V^* we define

$$\begin{aligned} M_1 \implies_{\gamma} M_2 \quad \text{iff} \quad & \text{there are } x, y, z, w \in V^* \text{ such that} \\ (i) \quad & M_1(x) \geq 1, (M_1 - \{(x, 1)\})(y) \geq 1, \\ (ii) \quad & x = x_1 u_1 u_2 x_2, y = y_1 u_3 u_4 y_2, \\ & z = x_1 u_1 u_4 y_2, w = y_1 u_3 u_2 x_2, \\ & \text{for } x_1, x_2, y_1, y_2 \in V^*, u_1 \# u_2 \$ u_3 \# u_4 \in R, \\ (iii) \quad & M_2 = (((M_1 - \{(x, 1)\}) - \{(y, 1)\}) \\ & \cup \{(z, 1)\}) \cup \{(w, 1)\}. \end{aligned}$$

At point (iii) we have operations with multisets. The writing above is meant to also cover the case when $x = y$ (then we must have $M_1(x) \geq 2$ and we must subtract 2 from $M_1(x)$), or $z = w$ (then we must add 2 to $M_2(z)$). When γ is understood, we write \implies instead of \implies_{γ} .

In plain words, when passing from a multiset M_1 to a multiset M_2 , according to γ , the multiplicity of two elements of M_1 , x and y , is diminished by one, and the multiplicity of the words which can be obtained by recombination, z and w , is augmented by one. The multiplicity of all other elements in $\text{supp}(M_1)$ is not changed. The obtained multiset is M_2 .

The language generated by an extended μ H system γ consists of all words containing only terminal symbols and whose multiplicity is at least once greater than or equal to one during the work of γ . Formally, we define this language by

$$L(\gamma) = \{w \in T^* \mid w \in \text{supp}(M) \text{ for some } M \text{ such that } A \implies_{\gamma}^* M\}.$$

An extended H system $\gamma = (V, T, A, R)$, as defined in Section 2.9, can be interpreted as an extended μ H system with $A(x) = \infty$ for all $x \in A$ and with $M(x) = \infty$ for all multisets M whose support is composed of strings x derived from A . Such multisets (with $M(x) = \infty$, if and only if $M(x) > 0$) are called ω -multisets, hence the corresponding H systems can be called ω H systems.

The family of languages generated by extended μ H systems $\gamma = (V, T, A, R)$ with $\text{card}(\text{supp}(A)) \leq n$ and $\text{rad}(R) \leq m$, $n, m \geq 1$, is denoted by $EH(\mu[n], [m])$; when n or m are not bounded, then we replace $[n], [m]$ by FIN .

Similarly, we may write the families $EH(FL_1, FL_2)$ as $EH(\omega FL_1, FL_2)$ in order to stress the fact that we work with ω -multisets.

It is important to point out here the fact that writing $M(x) = \infty$ for a string in $\text{supp}(M)$ does not necessarily mean that we actually have at our disposal infinitely many copies of x . It only means that we do not count the number of copies of x : at any moment when we need a copy of x we have it. In the DNA framework, this means that when we need further copies of a given sequence, we can produce them (for instance, by amplification).

Using multisets, hence counting the number of occurrences (of some) of the strings used, provides once again the tools for controlling the work of H systems in such a way as to characterize the family CE .

Lemma 2.6 $CE \subseteq EH(\mu FIN, [3])$.

Proof. Consider a type-0 Chomsky grammar $G = (N, T, S, P)$, with the rules in P of the form $u \rightarrow v$ with $1 \leq |u| \leq 2$, $0 \leq |v| \leq 2$, $u \neq v$ (for instance, we can take G in the Kuroda normal form). Also assume that the rules in P are labelled in a one-to-one manner. By U we denote the set $N \cup T \cup \{B\}$, where B is a new symbol, and we construct the extended μ H system

$$\gamma = (V, T, A, R),$$

where

$$V = N \cup T \cup \{B, X, Y, Z\} \cup \{Y_\alpha, Z_\alpha \mid \alpha \in U\} \cup \{Z_r \mid r \in P\},$$

the multiset A contains the word

$$w_0 = XSBBY,$$

with the multiplicity $A(w_0) = 1$, and the following words with an infinite multiplicity:

$$\begin{aligned} w_r &= Z_r v Y, \text{ for } r : u \rightarrow v \in P, \\ w_\alpha &= Z_\alpha X Y_\alpha, \text{ for } \alpha \in U, \\ w_t &= Z. \end{aligned}$$

The set R contains the following splicing rules:

- | | | |
|----------------------|---|---|
| $\text{Simulate} :$ | 1. $\beta_1 \beta_2 \# u Y \$ Z_p \# v Y,$ | for $p : u \rightarrow v \in P$ and $\beta_1, \beta_2 \in U,$ |
| $\text{Rotate} :$ | 2. $\beta_1 \beta_2 \# \alpha Y \$ Z_\alpha X \# Y_\alpha,$ | for $\alpha, \beta_1, \beta_2 \in U,$ |
| | 3. $Z_\alpha X \alpha \# Y \$ X \# \beta,$ | for $\alpha, \beta \in U,$ |
| | 4. $\beta \# Y_\alpha \$ X \# Y,$ | for $\alpha, \beta \in U,$ |
| | 5. $Z_\alpha \# X \alpha \$ \# X Y_\alpha,$ | for $\alpha, \beta \in U,$ |
| $\text{Terminate} :$ | 6. $X \# \beta \$ \# Z,$ | for $\beta \in U,$ |
| | 7. $\# BBY \$ X Z \#.$ | |

The system works according to the rotate-and-simulate procedure, with the peculiarity of the multiset control.

The sentential forms of the grammar G are reproduced in the system γ in a circularly permuted form (with the beginning marked by two copies of the new symbol B) and with the ends marked with X and Y . Consider that we have such a string XwY (initially, $w = SBB$), in exactly one copy.

The simulation of rules in P is done by splicing rules of type 1:

$$(Xw'|uY, Z_p|vY) \vdash (Xw'vY, Z_puY).$$

Note that such rules cannot be applied to an axiom Z_pvY , because to the left of Y we need at least three symbols from U : β_1, β_2 and at least one symbol in u . The by-product string Z_puY cannot enter new splicings (because for each rule $u \rightarrow v \in P$ we have $u \neq v$, the string Z_puY cannot be an axiom).

The rotation is performed in the following way. Take a string $Xw\alpha Y$, for some $\alpha \in U, w \in U^*$, which is present in exactly one copy. Actually, $|w| \geq 3$, because always we have at least two copies of B and at least one further symbol (remember that we ignore the empty string, hence we do not need to generate it by γ).

By the corresponding rule of type 2 we get

$$(Xw|\alpha Y, Z_\alpha X|Y_\alpha) \vdash (XwY_\alpha, Z_\alpha X\alpha Y).$$

The obtained strings are present in exactly one copy each, the string $Xw\alpha Y$ is no longer present.

The strings obtained in this way can enter the unique splicing

$$(Z_\alpha X\alpha|Y, X|wY_\alpha) \vdash (Z_\alpha X\alpha wY_\alpha, XY),$$

by using a rule of type 3. Again, the input strings are consumed, the output strings are present in exactly one copy each.

We continue by splicing the resulting strings by means a rule of type 4:

$$(Z_\alpha X\alpha w|Y_\alpha, X|Y) \vdash (Z_\alpha X\alpha wY, XY_\alpha).$$

Now, the string $Z_\alpha X\alpha wY$ (present in one copy only) can be spliced by using a rule of type 5 either with an axiom $Z_\alpha XY_\alpha$ (and nothing new is produced), or with the current string XY_α (present in one copy), and we get:

$$(Z_\alpha |X\alpha wY, |XY_\alpha) \vdash (Z_\alpha XY_\alpha, X\alpha wY).$$

The axiom $Z_\alpha XY_\alpha$ is reproduced, the string $Xw\alpha Y$ was permuted with one symbol.

The process can be iterated. Thus, each derivation step in the grammar G can be simulated in the system γ .

In order to terminate, we use the rules of types 6 and 7. First, we remove the symbol X ,

$$(X|wY, |Z) \vdash (XZ, wY),$$

and we produce in this way the string XZ , in a unique copy; then, using the string XZ , we can remove Y , in the presence of the two copies of B :

$$(w|BBY, XZ|) \vdash (w, XZBBY).$$

If the string w is terminal, then it is accepted in the language $L(\gamma)$, if not, then it is “lost”, because no splicing can be applied to it.

It is easy to see that no “illegal” splicing can be done, because of the control ensured by the multiplicity of strings and by the witness symbols β, β_1, β_2 (which prevent splicing axioms instead of strings XwY). In conclusion, $L(G) = L(\gamma)$.

Note that the system γ has the radius equal to three. □

By putting together the axioms of finite multiplicity and, separately, the axioms of infinite multiplicity, we can get the following result, whose proof is left to the reader (it can also be found in [224]):

Lemma 2.7 $EH(\mu FIN, [m]) \subseteq EH(\mu[2], [m])$, for all $m \geq 1$.

It is also easy to see that $EH(\mu[1], FIN) \subseteq REG$ (take an extended μH system $\gamma = (V, T, A, R)$ with $supp(A) = \{w\}$; if $A(w) < \infty$, then $L(\gamma)$ is obviously a finite language, and if $A(w) = \infty$, then $L(\gamma) \in EH([1], FIN) \subseteq EH(FIN, FIN) = REG$) and that $REG \subseteq EH(\omega[1], [2])$.

Combining these relations, we obtain:

Theorem 2.14 $REG = EH(\mu[1], [2]) = EH(\mu[1], FIN) \subset EH(\mu[2], FIN) = EH(\mu[2], [m]) = CE$, for all $m \geq 3$.

We have mentioned that the restrictions on the splicing operation in extended H systems we have considered in the first part of this section are mainly inspired from the regulated rewriting area. These restrictions are of a non-biochemical nature, hence they raise serious difficulties for present day laboratory techniques if we want to implement them. Unfortunately, also the multiset approach has a serious drawback: having two strings, each of them with multiplicity one, and splicing them is an event with a very low probability.

The results above show that the extended H systems controlled in various ways are computationally complete and, because all proofs are constructive, they also provide universal H systems of the considered types: starting the constructions from universal type-0 grammars, we get equivalent universal H systems. This is a proof that, from a theoretical point of view, the programmable DNA computer based on splicing is possible. Unfortunately, as we have mentioned above, it is a long way from the proofs to laboratory. Significant progress in biochemistry and genetic engineering is necessary before even hoping that the universal programmable DNA computer (based on splicing) can take its place near the silicon computer.

2.11 Distributed H Systems

One of the important drawbacks of the models considered in the previous section is the fact that in the constructions involved in the proofs we need many splicing rules. Each rule corresponds to two restriction enzymes. However, each enzyme needs specific reaction conditions, which means that several enzymes cannot work together, simultaneously. A suggestion how to cope with this difficulty comes from the grammar systems area: using distributed architectures, separating parts of an H system which are as small as possible and able to work independently, cooperating in a specified way, and synthesizing the result of the computation from the partial results produced by the parts.

Several types of distributed H systems were considered so far (see a survey in Chapter 10 of [224]). We present here only four variants, in general with sketched proofs. We give details only when they cannot be found in [224].

We will take into account two types of problems. One is of a mathematical interest: which is the minimum number of components of a distributed H system of a given type sufficient for characterizing the family of computably enumerable languages? From a practical point of view (having in mind the motivation we have started with), more interesting is the “orthogonal” problem: without imposing a bound on the number of components, which is the minimal size of the components (as the number of splicing rules) of a distributed H system which is able to characterize the computably enumerable languages? Of course, a trade-off is expected among these two parameters, the number of components and their size.

The first model we consider here is rather similar to a parallel communicating grammar system: the components are usual context-free grammars, working separately, synchronously, on their own sentential forms, and splicing their sentential forms according to a given set of splicing rules. Thus, we have a hybrid model, involving both rewriting operations and splicing operations.

A *splicing grammar system* (of degree n , $n \geq 1$) is a construct

$$\Gamma = (N, T, (S_1, P_1), (S_2, P_2), \dots, (S_n, P_n), R),$$

where

- (i) N, T are disjoint alphabets, $S_i \in N$, and $P_i, 1 \leq i \leq n$, are finite sets of rewriting rules over $N \cup T$,
- (ii) R is a finite subset of $(N \cup T)^* \# (N \cup T)^* \$ (N \cup T)^* \# (N \cup T)^*$, with $\#, \$$ two distinct symbols not in $N \cup T$.

The sets P_i are called the *components* of Γ .

For two n -tuples (we call them *configurations*) $x = (x_1, x_2, \dots, x_n)$, and $y = (y_1, y_2, \dots, y_n)$, $x_i, y_i \in (N \cup T)^*$, $1 \leq i \leq n$, we write $x \Rightarrow y$ if and only

if one of the following two conditions holds:

- (i) for each $1 \leq i \leq n$, either $x_i \Rightarrow_{P_i} y_i$, or $x_i \in T^*$;
- (ii) there exist $1 \leq i, j \leq n$ such that $x_i = x'_i u_1 u_2 x''_i$, $x_j = x'_j u_3 u_4 x''_j$, and $y_i = x'_i u_1 u_4 x''_j$, $y_j = x'_j u_3 u_2 x''_i$, for $u_1 \# u_2 \$ u_3 \# u_4 \in R$; for all $k \neq i, j$, we have $y_k = x_k$.

In the above definition, point (i) defines a rewriting step, whereas point (ii) defines a splicing step, corresponding to a communication step in parallel communicating grammar systems. Note that no priority of any of these operations over the other one is assumed and that as the result of a splicing operation we consider both strings obtained by recombination. In case (ii) we denote the passing from (x_i, x_j) to (y_i, y_j) by $(x_i, x_j) \vdash (y_i, y_j)$.

The language generated by the system Γ is the language generated by its first component, that is,

$$L(\Gamma) = \{x_1 \in T^* \mid (S_1, \dots, S_n) \Rightarrow^* (x_1, \dots, x_n), x_j \in (N \cup T)^*, 2 \leq j \leq n\}.$$

We denote by $SGS_n(X)$ the families of languages $L(\Gamma)$, generated by splicing grammar systems of degree at most n , $n \geq 1$, with components of type X . When no restriction is imposed on the number of components, then we replace the subscript n with $*$.

Note that in view of the definition of the relation \Rightarrow between configurations of a splicing grammar system, we implicitly use multiplicities of strings, because each component of the system has in every moment – both after a rewriting and a splicing step – exactly one current string; no string of arbitrary multiplicity is used. Thus, as expected, such systems can characterize CE ; somewhat unexpected is the fact that two λ -free context-free components suffice.

Theorem 2.15 $CE = SGS_n(CF) = SGS_*(CF)$, for all $n \geq 2$.

Proof. The inclusions $SGS_n(CF) \subseteq SGS_{n+1}(CF) \subseteq SGS_*(CF) \subseteq CE$, $n \geq 1$, are obvious. We have only to prove the inclusion $CE \subseteq SGS_2(CF)$. Here is the core construction involved in this proof.

Consider a language $L \subseteq T^*$, $L \in CE$, and take a grammar $G = (N, T, S, P)$ in the Geffert normal form as specified in Theorem 1.2: $N = \{S, A, B, C\}$ and P contains context-free rules of the form $S \rightarrow x, x \in (\{S, A, B, C\} \cup T)^+$, as well as the rule $ABC \rightarrow \lambda$. Denote by P' the set of context-free rules in P . We construct the splicing grammar system

$$\Gamma = (N \cup \{X, Y, Z, S_2\}, T, (S, P_1), (S_2, P_2), R),$$

with

$$\begin{aligned} P_1 &= P' \cup \{A \rightarrow A\}, \\ P_2 &= \{S_2 \rightarrow YXXABCZ, X \rightarrow XX, X \rightarrow X\}, \\ R &= \{\#ABC\$YX\#XABC, \#XABC\$YXABC\#\}. \end{aligned}$$

We obtain $L(G) = L(\Gamma)$ (the details are left to the reader). \square

One sees that the “main” component of the system is P_1 , which contains a number of rules which depends on the grammar G , and that we need only two splicing rules. We do not know whether or not only one splicing rule suffices, but the result is optimal as the number of components (in systems with only one component we have no communication, hence they are equivalent to context-free grammars).

Much more interesting is the following class of distributed H systems, which are “purely biochemical” (they correspond to parallel communicating grammar systems which communicate by command, in a way similar to the WAVE paradigm in distributed computing, where messages are broadcast to all components of a system and accepted by these components according to certain filters associated with them).

A *communicating distributed H system* (of degree $n, n \geq 1$) is a construct

$$\Gamma = (V, T, (A_1, R_1, V_1), \dots, (A_n, R_n, V_n)),$$

noindent where V is an alphabet, $T \subseteq V$, A_i are finite languages over V , R_i are finite sets of splicing rules over V , and $V_i \subseteq V$, $1 \leq i \leq n$.

Each triple (A_i, R_i, V_i) , $1 \leq i \leq n$, is called a *component* of Γ ; A_i, R_i, V_i are the set of *axioms*, the set of *splicing rules*, and the *selector* (or *filter*) of the component i , respectively; T is the terminal alphabet of the system.

We denote

$$B = V^* - \bigcup_{i=1}^n V_i^*.$$

The pair $\sigma_i = (V, R_i)$ is the underlying H scheme associated with the component i of the system.

An n -tuple (L_1, \dots, L_n) , $L_i \subseteq V^*$, $1 \leq i \leq n$, is called a *configuration* of the system; L_i is also called the *contents* of the i th component, understanding the components as test tubes where the splicing operations are carried out.

For two configurations $(L_1, \dots, L_n), (L'_1, \dots, L'_n)$, we define

$$(L_1, \dots, L_n) \Rightarrow (L'_1, \dots, L'_n) \text{ iff}$$

$$L'_i = \bigcup_{j=1}^n (\sigma_j^*(L_j) \cap V_i^*) \cup (\sigma_i^*(L_i) \cap B),$$

for each $i, 1 \leq i \leq n$.

In words, the contents of each component are spliced according to the associated set of rules (we pass from L_j to $\sigma_j^*(L_j)$, $1 \leq j \leq n$), and the result is redistributed among the n components according to the selectors V_1, \dots, V_n ; the part which cannot be redistributed (which does not belong to some V_i^* , $1 \leq i \leq n$) remains in the component. As we have imposed no

restriction over the alphabets V_i , for example, we did not suppose that they are pairwise disjoint, when a string in $\sigma_j^*(L_j)$ belongs to several languages V_i^* , then copies of this string will be distributed to all components i with this property.

The language generated by Γ is defined by

$$L(\Gamma) = \{w \in T^* \mid w \in L_1 \text{ for some } L_1, \dots, L_n \subseteq V^* \text{ such that } (A_1, \dots, A_n) \Rightarrow^* (L_1, \dots, L_n)\}.$$

That is, the first component of the system is designated as its *master* and the language of Γ is the set of all terminal strings generated (or collected by communications) by the master.

We denote by CDH_n the family of languages generated by communicating distributed H systems of degree at most n , $n \geq 1$. When n is not specified, we replace the subscript n with $*$.

Without a proof, we give here the basic results about these systems:

Theorem 2.16 $CE = CDH_n = CDH_*$, for all $n \geq 3$.

Communicating distributed H systems of degree 1 do not use communication, hence they are extended finite H systems, that is, $CDH_1 = REG \subseteq CDH_2$. In fact, it is known that the inclusion $REG \subset CDH_2$ is proper.

It is an *open problem* whether or not the inclusion $CDH_2 \subseteq CDH_3$ is also proper, hence whether or not the result in Theorem 2.16 can be strengthened, to $n = 2$. We expect a negative answer. (We even *conjecture* that $CDH_2 \subset CF$.)

In what concerns the problem of using components of a small size, we have:

Theorem 2.17 For each type-0 grammar $G = (N, T, S, P)$ we can construct a communicating distributed H system Γ such that $L(G) = L(\Gamma)$, the degree of Γ is $2(\text{card}(N \cup T) + 1) + \text{card}(P) + 9$, and each component of Γ contains only one splicing rule.

Therefore, at the expense of working with a large number of tubes, the size of each tube is minimal: only one splicing rule.

The work of enzymes can be controlled by “environment” conditions, in particular, by the temperature. By cyclically varying the temperature, we cyclically use the enzymes in a test tube. This leads to the notion of a time-varying H system.

A *time-varying H system* (of degree n , $n \geq 1$) is a construct

$$\Gamma = (V, T, A, R_1, R_2, \dots, R_n),$$

where V is an alphabet, $T \subseteq V$ (terminal alphabet), A is a finite subset of V^* (axioms), and R_i are finite sets of splicing rules over V , $1 \leq i \leq n$.

At each moment $k = n \cdot j + i, j \geq 0, 1 \leq i \leq n$, the component R_i is used for splicing the currently available strings. Formally, we define

$$\begin{aligned} L_0 &= A, \\ L_k &= \sigma_i(L_{k-1}), \text{ for } i \equiv k(\text{mod } n), k \geq 1, \end{aligned}$$

where $\sigma_i = (V, R_i), 1 \leq i \leq n$.

The language generated by Γ is defined by

$$L(\Gamma) = (\bigcup_{k \geq 0} L_k) \cap T^*.$$

We denote by $TVH_n, n \geq 1$, the family of languages generated by time-varying distributed H systems of degree at most n and by TVH_* the family of all languages generated by time-varying H systems.

Theorem 2.18 $CE = TVH_n = TVH_*, n \geq 4$.

Proof. Clearly, we have to prove only the inclusion $CE \subseteq TVH_4$.

Consider a type-0 grammar $G = (N, T, S, P)$. Assume that $N \cup T = \{\alpha_1, \dots, \alpha_{n-1}\}$. Let $\alpha_n = B$ be a new symbol. We label the rules in P by $r_j : u_j \rightarrow v_j$, for $n+1 \leq j \leq m$. We also consider that $u_i = v_i = \alpha_i$, for $1 \leq i \leq n$.

We construct the time-varying H system

$$\Gamma = (V, T, A, R_1, R_2, R_3, R_4),$$

with

$$\begin{aligned} V &= N \cup T \cup \{X, Y, Z, Z', Q, B\} \cup \{X_i, Y_i \mid 0 \leq i \leq m\}, \\ A &= \{XBSY, ZY, ZQ, XZ', QZ'\} \cup \{X_j Z', ZY_j \mid 0 \leq j \leq m\} \\ &\quad \cup \{X_j v_j Z' \mid 1 \leq j \leq m\}, \\ R_1 &= \{\#Y \$ \# Y, Z \$ \$ Z \#, \# Z' \$ \# Z', X_0 \$ \$ X_0 \#\} \\ &\quad \cup \{\#Y_j \$ \# Y_j \mid 1 \leq j \leq m\}, \\ R_2 &= \{X \$ \$ X \#, Z \$ \$ Z \#, \# Z' \$ \# Z', \# Y_0 \$ \$ Y_0 \} \\ &\quad \cup \{X_j \$ \$ X_j \# \mid 1 \leq j \leq m\}, \\ R_3 &= \{Z \$ \$ Z \#, \# Z' \$ \# Z', \# Y_0 \$ Z \# Y, \# Y_0 \$ ZQ \#\} \\ &\quad \cup \{\# u_j Y \$ Z \# Y_j, \# Y_j \$ Z \# Y_{j-1} \mid 1 \leq j \leq m\}, \\ R_4 &= \{Z \$ \$ Z \#, \# Z' \$ \# Z', X \$ Z' \$ X_0 \#, \# Z' Q \$ X_0 B \#\} \\ &\quad \cup \{X_j v_j \$ Z' \$ X \#, X_j \$ \$ X_{j-1} \$ Z' \mid 1 \leq j \leq m\}. \end{aligned}$$

We will prove that $L(\Gamma) = L(G)$.

The idea of this proof is again “rotate-and-simulate” as introduced in the proof of the Basic Universality Lemma.

Here both the simulation of the rules in G and the circular permutation of strings are performed in Γ in the same way: a suffix u of the current string is removed and the corresponding string, v , is added in the left end of the string. For $u \rightarrow v$ a rule in P we simulate a derivation step in G . For $u = v$ a symbol in $N \cup T \cup \{B\}$ we have one symbol “rotation” of the current string.

The simulation and “rotation” are performed by the components R_3 and R_4 ; the components R_1, R_2 have the role of checking the correctness of these operations (see below).

$L(G) \subseteq L(\Gamma)$. Consider a string of the form XwY (initially, $w = BS$) when R_1 is active. Using the rule $\#Y\$#Y$, the string is passed unchanged to R_2 . From R_2 the string is passed to R_3 by using the rule $X\#\$X\#$. Now, if $w = w'u_i$ for some $n+1 \leq i \leq m$ (that is, $u_i \rightarrow v_i \in P$), then in R_3 we can perform:

$$(Xw'|u_iY, Z|Y_i) \vdash Xw'Y_i.$$

(The axioms are passed from a component to the next one by using the rules $Z\#\$Z\#$, $\#Z'\$\#Z'$, which are present in all components.) The string $Xw'Y_i$ is passed to R_4 where we perform:

$$(X_iv_i|Z', X|w'Y_i) \vdash X_iv_iw'Y_i.$$

This string arrives in R_1 , it is passed unchanged to R_2 (using the rule $\#Y_i\$#Y_i \in R_1$), then to R_3 (by $X_j\#\$X_j\# \in R_2$). Here, the subscript of Y is decreased by one; the resulting string, $X_iv_iw'Y_{i-1}$, is passed to R_4 , where the subscript of X is decreased by one. The process can be iterated, so eventually we get the string $X_0v_iw'Y_0$. This string passes through R_1, R_2 , in R_3 we can substitute Y_0 by Y , and in R_4 we substitute X_0 by X . In this way we have passed from $Xw'u_iY$ to $Xv_iw'Y$, which corresponds to simulating the rule $u_i \rightarrow v_i$ from P .

If $1 \leq i \leq n$, that is, $u_i = v_i = \alpha_i \in N \cup T \cup \{B\}$, then in this way we circularly permute the string with one symbol.

By iterating this procedure, we can simulate any rule of P in any desired position. Thus, for every derivation $S \implies^* x_1x_2$ in G , we can produce the string Xx_2Bx_1Y in Γ . By a circular permutation, we can also produce the string $X_0Bx_1x_2Y_0$. When we get a string of the form X_0BwY_0 , we can remove Y_0 by the rule $\#Y_0\$ZQ\# \in R_3$ and X_0B by the rule $\#Z'Q\$X_0B\# \in R_4$. If the obtained string, w , is a terminal one, then $w \in L(\Gamma)$. Consequently $L(G) \subseteq L(\Gamma)$.

$L(G) \supseteq L(\Gamma)$. Let us examine the form of the strings which can be passed from a component of Γ to the next one.

First of all, we notice that all the axioms (except $XSBY$) can pass through all components by using the rules $Z\#\$Z\#$, $\#Z'\$\#Z'$.

Let us assume that we have performed the operation

$$(Xw'|u_iY, Z|Y_i) \vdash Xw'Y_i$$

in R_3 and in R_4 we performed

$$(X_j v_j | Z', X | w' Y_i) \vdash X_j v_j w' Y_i,$$

for some $1 \leq i, j \leq m$. A string of the form $X_j v_j w' Y_i$ with $i \geq 1$ can pass through R_1 by using the rule $\#Y_i\$ \#Y_i$. Then, the string can pass through R_2 using the rule $X_j\#\#X_j\#$, providing that $j \geq 1$.

In R_3 we decrement by one the subscript of Y and this is the only possible operation in this component. Similarly, in R_4 we can only decrement the subscript of X . We repeat this process until at least one subscript reaches zero.

Suppose that after R_4 we have a string $X_0 w Y_k$, for $k \geq 1$. Because R_1 contains the rules $X_0\#\$X_0\#$, $\#Y_k\#\#Y_k$, the string can be passed unchanged to R_2 . There is no rule in R_2 which can be applied to this string, so the string is “lost”.

If after R_4 we have a string $X_k w Y_0$, for $k \geq 1$, then it cannot pass through R_1 .

So, the only strings having at least a subscript equal to zero which “survive” are those of the form $X_0 w Y_0$. This implies that the string $X_j v_j w' Y_i$ from which we have started to decrement the subscripts has $i = j$. This means that we have correctly simulated the use of a rule $u_i \rightarrow v_i$ in P , or we have circularly permuted the string by one symbol.

Consequently, the only terminal strings which can be generated by Γ correspond to strings in $L(G)$; because X_0 can be removed only when B is adjacent to it, the string is in the same circular permutation as the associated string in $L(G)$, that is, the two strings are identical. \square

It is highly possible that the result above can be improved: a number of components smaller than four could be sufficient, but we do not know a convincing proof of such a result. In what concerns the question about the size of components, the best known result is the following one:

Theorem 2.19 *Each computably enumerable language can be generated by a time-varying distributed H system whose components contain at most three splicing rules.*

Proof. We give only the core construction of the proof. Consider a type-0 grammar $G = (N, T, S, P)$ with $N \cup T = \{\alpha_1, \dots, \alpha_{n-1}\}$, $n \geq 3$, and $P = \{u_i \rightarrow v_i \mid 1 \leq i \leq m\}$. Let $\alpha_n = B$ be a new symbol. We construct the time-varying distributed H system

$$\Gamma = (V, T, A, R_1, \dots, R_{2n+m+2}),$$

with

$$V = N \cup T \cup \{X, Y, Y', Z, Z', B\},$$

$$\begin{aligned} A = & \{XBSY, ZY, ZY', ZZ'\} \\ & \cup \{ZvY \mid u \rightarrow v \in P\} \\ & \cup \{X\alpha_i Z \mid 1 \leq i \leq n\}, \end{aligned}$$

and the following sets of splicing rules

$$\begin{aligned} R_i &= \{\#u_i Y \$ Z \# v_i Y, \# Y \$ Z \# Y, Z \$ \$ Z \#\}, 1 \leq i \leq m, \\ R_{m+2j-1} &= \{\# \alpha_j Y \$ Z \# Y, \# Y \$ Z \# Y', Z \$ \$ Z \#\}, 1 \leq j \leq n, \\ R_{m+2j} &= \{X\alpha_j \# Z \$ X \#, \# Y' \$ Z \# Y, Z \$ \$ Z \#\}, 1 \leq j \leq n, \\ R_{m+2n+1} &= \{\# Z Z' \$ X B \#, \# Y \$ Z \# Y', Z \$ \$ Z \#\}, \\ R_{m+2n+2} &= \{\# Y \$ Z Z' \#, \# Y' \$ Z \# Y, Z \$ \$ Z \#\}. \end{aligned}$$

The equality $L(G) = L(\Gamma)$ can be proved in a way similar to that in the proof of the previous theorem. \square

We have mentioned that in grammar systems theory there also exists a large class of systems whose components work in a sequential manner (one component is active in each time unit, the other ones are just waiting). Also a counterpart of such a distributed system was considered for the case of H systems.

A *sequential distributed H system* (of degree $n, n \geq 1$) is a construct

$$\Gamma = (V, T, w, (A_1, R_1), \dots, (A_n, R_n)),$$

where V is an alphabet, $T \subseteq V$, $w \in V^*$, A_i is a finite subset of V^* , and R_i is a finite subset of $V^* \# V^* \$ V^* \# V^*$, $1 \leq i \leq n$.

V is the alphabet of Γ , T is the terminal alphabet, w is the axiom, (A_i, R_i) is called a *component* of the system; A_i is the set of axioms of the component i , R_i is the set of splicing rules of the component i , $1 \leq i \leq n$.

The work of Γ consists of iterated applications of the splicing schemes $\sigma_i = (V, R_i)$ to pairs $(x, z), (z, x)$ for $x \in A_i$, where z is the string obtained at the previous step, taking w as the starting string; moreover, the use of σ_i is *maximal* in the sense that we stop using it only if no further splicing is possible. Formally, for $x, y \in V^*$ and $i \in \{1, 2, \dots, n\}$, we define

$$\begin{aligned} x \rightarrow_i y &\quad \text{iff } (x, z) \vdash_r y \text{ or } (z, x) \vdash_r y, \text{ for some } z \in A_i, r \in R_i, \\ x \rightarrow_i^* y &\quad \text{iff } x = y, \text{ or } x = x_0 \rightarrow_i x_1 \rightarrow_i \dots \rightarrow_i x_k = y, k \geq 1, \\ &\quad x_j \in V^*, 1 \leq j \leq k, \\ x \Rightarrow_i y &\quad \text{iff } x \rightarrow_i^* y \text{ and there is no } u \in V^* \text{ such that } y \rightarrow_i u. \end{aligned}$$

The language generated by Γ is defined by

$$\begin{aligned} L(\Gamma) = & \{x \in T^* \mid w \Rightarrow_{i_1} w_1 \Rightarrow_{i_2} \dots \Rightarrow_{i_m} w_m, \\ & x = w_m, m \geq 1, w_j \in V^*, 1 \leq i_j \leq n, 1 \leq j \leq m\}. \end{aligned}$$

Note that, although we work systematically on strings which are obtained from the axiom w , we do not have here multisets; each string is assumed to appear in an arbitrary number of copies. This is especially important for the sets $A_i, 1 \leq i \leq n$, which are continuously available in their initial form.

We denote by SDH_n the family of languages generated by sequential distributed H systems of degree at most $n, n \geq 1$, and by SDH_* the union of all these families.

Theorem 2.20 $CE = SDH_n = SDH_*, n \geq 3$.

Proof. We prove only the inclusion $CE \subseteq SDH_3$.

Consider a type-0 Chomsky grammar $G = (N, T, S, P)$. Consider a new symbol B and assume that

$$N \cup T \cup \{B\} = \{\alpha_1, \dots, \alpha_n\},$$

with $B = \alpha_1$; because both N and T are non-empty, we have $n \geq 3$.

We construct the sequential distributed H system

$$\Gamma = (V, T, w, (A_1, R_1), (A_2, R_2), (A_3, R_3)),$$

where

$$\begin{aligned} V &= N \cup T \cup \{X, X', \overline{X}, Y, Y', Y'', \overline{Y}, Z, B, C\}, \\ w &= XBSY, \\ A_1 &= \{ZvY \mid u \rightarrow v \in P\} \\ &\cup \{ZC^iY' \mid 1 \leq i \leq n\} \\ &\cup \{ZY'', XZ, \overline{X}Z\}, \\ R_1 &= \{\#uY\$Z\#vY \mid u \rightarrow v \in P\} \\ &\cup \{\#\alpha_i Y \$ Z \# C^i Y' \mid 1 \leq i \leq n\} \\ &\cup \{\#CY\$Z\#Y'', X'\#\$X\#Z, \overline{X}\#\$\overline{X}\#Z, \#\overline{Y}\$XZ\#\}, \\ A_2 &= \{X'CZ, ZY', ZY\}, \\ R_2 &= \{X'C\#Z\$X\#, \#Y''\$Z\#Y', \overline{X}B\#\$\#ZY, \#Y\$Z\#Y\}, \\ A_3 &= \{X\alpha_i Z, X\alpha_i CZ \mid 1 \leq i \leq n\} \\ &\cup \{ZY, \overline{X}BZ, Z\overline{Y}, ZCY, XCZ, ZY''\}, \\ R_3 &= \{X\alpha_i \#Z\$X'C^i \# \mid 1 \leq i \leq n\} \\ &\cup \{\#Y'\$Z\#Y, \overline{X}B\#\$Z\$XC\#, \#Y'\$Z\#\overline{Y}, \\ &\quad \#CY\$Z\#CY, XC\#\$XC\#Z, \#Y''\$Z\#Y''\} \\ &\cup \{X\alpha_i C\#\$X\alpha_i C\#Z \mid 1 \leq i \leq n\} \cup \{\overline{X}BC\#\$\overline{X}BC\#Z\}. \end{aligned}$$

Let us examine the work of the system Γ .

Each component contains certain splicing rules which are meant to be trap rules: if they can be applied once, they can be applied for ever, hence

the corresponding component cannot stop correctly its work. Such rules are $\overline{X} \# \$ \overline{X} \# Z$ in the first component, $\# Y \$ Z \# Y$ in the second one, and $\# C Y \$ Z \# C Y$, $X C \# \$ X C \# Z$, $\# Y'' \$ Z \# Y''$, $X \alpha_i C \# \$ X \alpha_i C \# Z$, $1 \leq i \leq m$ and $\overline{X} B C \# \$ \overline{X} B C \# Z$ in the third component.

Thus, if a string of the form XxY , $x \in (N \cup T \cup \{B\})^*$ (initially, we have $x = BS$) is processed by the second component, then the system is blocked. The third component cannot modify such a string, hence it is passed away unchanged.

In the first component, a string XxY is processed as follows. At the end of x we can simulate rules of G $((Xx_1|uY, Z|vY) \vdash Xx_1vY$, for $x = x_1u$ and $u \rightarrow v \in P$. Also, a symbol α_i can be cut from the end of x $((Xx_1|\alpha_iY, Z|C^iY') \vdash Xx_1C^iY'$, when $x = x_1\alpha_i$); the removed symbol is replaced by i occurrences of the symbol C and the replacement is also indicated by replacing Y with Y' . We have to continue with $(Xx_1C^{i-1}|CY', Z|Y'') \vdash Xx_1C^{i-1}Y''$. Note that one occurrence of C has been removed and Y' has been replaced with Y'' .

No further splicing can be done in the first component involving the obtained string $Xx_1C^{i-1}Y''$, so it has to be passed to another component. The only possibility which does not block the system is to start working in the second component. Two splicings can be done here, at the ends of the string: $(X'C|Z, X|x_1C^{i-1}Y') \vdash X'Cx_1C^{i-1}Y''$ and $(X'Cx_1C^{i-1}|Y'', Z|Y') \vdash X'Cx_1C^{i-1}Y'$. In this way, one occurrence of C has been introduced in the left end of the string and the markers at the ends of the string have become X', Y' . No further splicing can be done here. We have two cases:

(1) If the string is passed to the first component, then, after using the rule $X' \# \$ X \# Z$, we obtain the string $X C x_1 C^{i-1} Y'$ and again we have to cut one occurrence of C from its right hand and to replace Y' with Y'' . As above, the string must be passed to the second component and the process can be iterated. In this way, any number of occurrences of the symbol C can be removed from the right hand end of the string and reintroduced in the left hand end.

(2) If the string is passed to the third component, then again two splicings will be done here, at the ends of the string. Assume that we start working in (A_3, R_3) on a string of the form $X'C^j x_1 C^k Y'$ for some $j, k \geq 0$. The symbol Y' is replaced by Y . If $k \geq 1$, then the rule $\# C Y \$ Z \# C Y$ can be used forever. Therefore, we must start from a string $X'C^j x_1 Y'$. If we perform a splicing $(X\alpha_s|Z, X'C^s|C^{j-s}x_1Y) \vdash X\alpha_s C^{j-s} x_1 Y$ with $j > s$, then again the system is blocked: the rule $X\alpha_s C \# \$ X\alpha_s C \# Z$ can be used indefinitely. A correct continuation is possible only when we have $s = j$; this leads to the string $X\alpha_s x_1 Y$. In this way, all occurrences of C were replaced by the corresponding symbol in $N \cup T \cup \{B\}$. This means that exactly the symbol α_i which has been removed from the right hand end of the string has been reintroduced in the left hand end (with the notation above, $i = j = s$). Thus,

any circular permutation of the string can be obtained.

The string produced by the third component is of the form XzY , hence it can only be processed by the first component, where it is possible to simulate other rules in P in the end of z .

By iterating this rotate-and-simulate procedure, we can simulate in Γ any derivation in G .

Note that B is circulated as any other symbol. Initially, B is introduced in the left hand of S , the axiom of G . Because at every moment exactly one occurrence of B is present, it indicates the actual beginning of the sentential form of G simulated by Γ in a permuted form: if the string produced by Γ is Xz_1Bz_2Y , possibly with X, Y replaced with some primed versions of them, then z_2z_1 is a sentential form of G .

After receiving a string $X'C^jx_1Y'$, $j \geq 1$, the third component can also use the rules of the form $\overline{X}B\#Z\$XC\#\#, \#Y'\$Z\#\overline{Y}$. If both these rules are used, then we get the string $\overline{X}BC^{j-1}x_1\overline{Y}$. If $j - 1 \geq 1$, then the system is blocked by the rule $\overline{X}BC\#\overline{X}BC\#Z$. Therefore, the string must be $\overline{X}Bx_1\overline{Y}$. The first component is blocked if receiving this string; the second one can work one step, removing the prefix $\overline{X}B$. The obtained string has to be taken by another component. The third one cannot modify it, the first one will remove the symbol \overline{Y} from its end. If the obtained string is terminal, then it is an element of the generated language. Because the unique occurrence of the symbol B has been removed when placed in the leftmost position, it follows that the string is in the same circular permutation as the corresponding sentential form of G .

Finally, assume that the third component produces a string with only one barred symbol. If this string is of the form $\overline{X}BxY$, then no component can process it without blocking the system: the first component contains the rule $\overline{X}\#\overline{X}\#Z$, the second one contains the rule $\#Y\$Z\#\overline{Y}$. If the string is of the form $XBx\overline{Y}$, then it can reach both the first and the second components.

In the first component, the string $XBx\overline{Y}$ will lose the symbol \overline{Y} , then it has to be moved to the second component. Here, a symbol C is added in the left end, producing $X'CBx$. This string can go back to the first component; iterating these steps, we can produce $X'C^iBx$ for some $i \geq 1$. Eventually, the string will be processed by the third component, and a string $X\alpha_iBx$ is produced. If $\alpha_i \neq B$, then B can never be removed: it cannot be moved near X , because no rotation phase is possible (the symbol Y is no longer present). If $\alpha_i = B$ (hence $i = 1$), then XB can be removed, but we get Bx , which is not a terminal string.

If the string $XBx\overline{Y}$ arrives in the second component, then again we can introduce an occurrence of C in its left end, the obtained string $X'CBx\overline{Y}$ is then processed by the first component, which removes \overline{Y} , and we have returned to a situation as above: no terminal string can be produced, because the symbol B cannot be removed.

In conclusion, every derivation in G can be simulated in Γ and, conversely,

if a terminal string is produced by the system Γ , then it is an element of $L(G)$. That is, $L(G) = L(\Gamma)$. \square

We do not know which of the inclusions $SDH_1 \subseteq SDH_2 \subseteq SDH_3$ are proper. In what concerns the number of splicing rules in each component, the following result, whose proof is omitted, holds:

Theorem 2.21 *For each type-0 grammar $G = (N, T, S, P)$ we can construct a sequential distributed H system Γ of degree $2 \cdot \text{card}(N \cup T) + \text{card}(P) + 10$, with each component containing at most three splicing rules, and such that $L(G) = L(\Gamma)$.*

2.12 Bibliographical Notes

From the many good books in molecular biology and genetic engineering which are available we only mention [6], [81], [274]. Good introductions to the biochemistry of DNA are [142] and [266]. A presentation of DNA structure and operations with implicit and explicit orientation to computer science and DNA computing can be found in the first chapter of [224].

Theorem 2.2 is from [92]. The connection between DNA structure and the twin-shuffle language was first observed in [257] and then discussed in various places: [224], [225], [264], etc.

In Sections 2.4–2.8 we have given a series of references, which we do not recall here.

The idea of computing by carving was proposed in [214] and then presented in [181]. Iterated gsm's were investigated in [207], [255], [299].

The splicing operation was introduced in [127], while the extended H systems were first considered in the explicit form in [223]. The Regularity Preserving Lemma is due to [240]; the complete proof also appears in [224]. A more general result is given in [242]: each abstract family of languages (AFL) is closed under iterated splicing with respect to a finite set of splicing rules. This proof is also given in [129]. The Basic Universality Lemma is from [209].

In Section 2.10 we have given some references about the mentioned results; further details can be found in [224].

A chapter about distributed H systems can be found in [224], where also a further class of systems is presented (while the sequential distributed H systems are not discussed in [224]), the so-called two-level distributed H systems, introduced in [213] (systems of this type with three components characterize CE , but no result is known about the size of the components).

Splicing grammar systems were introduced in [74]; Theorem 2.15 is from [210]. Further results can be found in [111].

Communicating distributed H systems were introduced in [70] and investigated in [211], [306], [243]; the proof of Theorem 2.16 can be found in [216]. Theorem 2.17 is from [215].

The time-varying distributed H systems are introduced in [211]; Theorem 2.18 is from [205]. Theorem 2.19 is from [216]. In [183] it is announced that time-varying H systems with two components generate each computably enumerable languages.

The sequential distributed H systems were introduced in [184]; Theorems 2.20 and 2.21 are from this paper.

Another computability model based on the operation of annealing is proposed in [303] and further elaborated in [304]; it is based on a careful analysis of the proof of Geffert normal form (Theorem 1.2 (ii)), [110].

Chapter 3

Membrane Computing

*If any entity should be thought of
as a governor of cellular activity,
then this should certainly be the membrane.*

*All major activities of cells
are topologically connected to membrane.*

J. Hoffmeyer, 1998

In this chapter we introduce a new computability model which is biochemically inspired, but not necessarily using DNA molecules and DNA operations. It is a general distributed model, highly parallel, based on the notion of a *membrane structure*. Such a structure consists of several cell-like membranes, recurrently placed inside a unique “skin” membrane. A plane representation is a Venn diagram without intersected sets and with a unique superset. In the regions delimited by the membranes there are placed *objects*. These objects are assumed to evolve: each object can be transformed into other objects, can pass through a membrane, or can dissolve the membrane in which it is placed. A priority relation between evolution rules can be considered. The evolution is done in parallel for all objects able to evolve. In this way, we obtain a computing device (we call it a *P system*): start with a certain number of objects in certain membranes and let the system evolve; if it will halt (no object can further evolve), then the computation is finished, with the result given as the number of objects in a specified membrane. If the development of the system goes for ever, then the computation fails to have an output.

Also membrane division can be considered. In this way, an enhanced parallelism is obtained, which can be very useful from the point of view of computational complexity (we will see that SAT can be solved in linear time in such a framework).

We will consider here several variants of P systems, giving proofs only for some basic results (in general, dealing with the power of these systems).

We enter into some technical details, because we want to let the reader to be acquainted with this very recently developed topic.

3.1 P Systems with Labelled Membranes

We start from the observation that life “computes” not only at the genetic level, but also at the cellular level. More generally, any non-trivial biological system is a hierarchical construct, composed of several “organs” which are well defined and delimited from the neighbouring organs, which evolve internally and also cooperate with the other organs in order to keep alive the system as a whole; an intricate flow of materials, energy, and information underlies the functioning of such a system.

At a more specific level with respect to the models we are going to define, important to us is the fact that the parts of a biological system are well delimited by various types of *membranes*, in the broad sense of the term, starting from the membranes which delimit the various intra-cell components, going to the cell membrane and then to the skin of organisms, and ending with more or less virtual “membranes” which delimit, for instance, parts of an ecosystem. In very practical terms, in biology and chemistry one knows membranes which keep together certain chemicals and allow other chemicals to pass, in a selective manner, sometimes only in one direction (for instance, through protein channels placed in membranes). Membranes delimiting subsystems of a symbol manipulating system are also considered in the logical framework to the so-called metabolic systems, as defined in [180], or in the so-called chemical abstract machine, introduced in [32].

Formalizing the previous intuitive ideas, we now introduce the basic structural ingredient of the computing devices we will define later: membrane structures.

Let us consider first the language MS over the alphabet $\{[,]\}$, whose strings are recurrently defined as follows:

1. $[] \in MS$;
2. if $\mu_1, \dots, \mu_n \in MS$, $n \geq 1$, then $[\mu_1 \dots \mu_n] \in MS$;
3. nothing else is in MS .

Consider now the following relation over the elements of MS : $x \sim y$ if and only if we can write the two strings in the form $x = \mu_1\mu_2\mu_3\mu_4$, $y = \mu_1\mu_3\mu_2\mu_4$, for $\mu_1\mu_4 \in MS$ and $\mu_2, \mu_3 \in MS$ (two pairs of parentheses which are neighbours at the same level are interchanged, together with their contents). We also denote by \sim the reflexive and transitive closure of the relation \sim . This is clearly an equivalence relation. We denote by \overline{MS} the set of equivalence classes of MS with respect to this relation. The elements of \overline{MS} are called *membrane structures*.

Each matching pair of parentheses $[,]$ appearing in a membrane structure is called a *membrane*. The number of membranes in a membrane structure μ is called the *degree* of μ and denoted by $\deg(\mu)$. The external membrane of a membrane structure μ is called the *skin membrane* of μ . A membrane which appears in $\mu \in \overline{MS}$ in the form $[]$ (no other membrane appears inside the two parentheses) is called an *elementary membrane*.

The *depth* of a membrane structure μ , denoted by $\text{dep}(\mu)$, is defined recurrently as follows:

1. if $\mu = []$, then $\text{dep}(\mu) = 1$;
2. if $\mu = [\mu_1 \dots \mu_n]$, for some $\mu_1, \dots, \mu_n \in MS$,
then $\text{dep}(\mu) = \max\{\text{dep}(\mu_i) \mid 1 \leq i \leq n\} + 1$.

A membrane structure can be represented in a natural way as a Venn diagram. This makes clear the fact that the order of membrane structures of the same level in a larger membrane structure is irrelevant; what matters is the topological structure, the relationships between membranes. In the sequel we will make an extensive use of such a representation.

The Venn representation of a membrane structure μ also makes clear the notion of a *region* in μ : any closed space delimited by membranes is called a region of μ . It is clear that a membrane structure of degree n contains n internal regions, one associated with each membrane. We also use to speak about the *outer* region, the whole space outside the skin membrane.

Figure 3.1 illustrates some of the notions mentioned above.

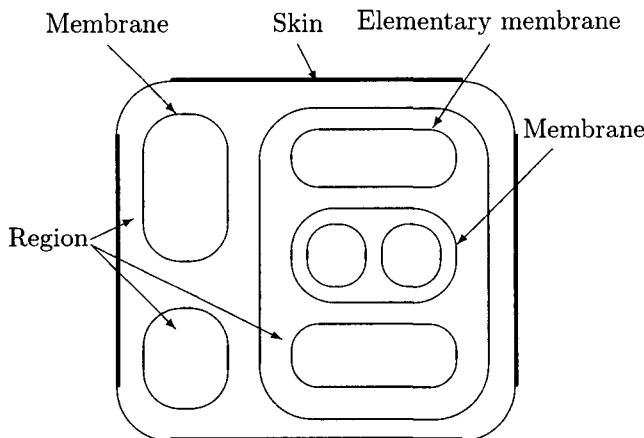


Figure 3.1: A membrane structure.

We now make one more step towards the definition of a computing device, by adding objects to a membrane structure.

Let U be a finite set whose elements are called *objects*.

Consider a membrane structure μ of degree n , $n \geq 1$, with the membranes labelled in a one-to-one manner, for instance, with the numbers from 1 to n . In this way, also the regions of μ are identified by the numbers from 1 to n . If a multiset $M_i : U \longrightarrow \mathbf{N}$ is associated with each region i of μ , $1 \leq i \leq n$, then we say that we have a *super-cell* (note that we do not allow infinite multiplicities of objects in U).

Any multiset M_i mentioned above can be empty. In particular, all of them can be empty, that is, any membrane structure is a super-cell. On the other hand, each individual object can appear in several regions, in several copies in each of them.

Several notions defined for membrane structures are extended in the natural way to super-cells: degree, depth, region, etc.

The multiset corresponding to a region of a super-cell (in particular, it can be an elementary membrane) is called *the contents* of this region. The total multiplicities of the elements in an elementary membrane m (the sum of their multiplicities) is called *the size* of m and is denoted by $\text{size}(m)$.

If a membrane m' is placed in a membrane m such that m and m' contribute to delimiting the same region (namely, the region associated with m), then all objects placed in the region associated with m are said to be *adjacent* to membrane m' (so, they are immediately “outside” membrane m' and “inside” membrane m).

A super-cell can be described by a Venn diagram where both the membranes and the objects are represented (in the case of the objects, taking care of multiplicities).

We are now ready to introduce the subject of our investigation, a computing mechanism essentially designed as a distributed parallel machinery, having as the underlying structure a super-cell. The basic additional feature is the possibility of objects to evolve, according to certain rules. Another feature refers to the definition of the input and the output (the result) of a computation.

A *P system* is a super-cell provided with evolution rules for its objects and with a designated output region.

More formally, a *P system* of degree m , $m \geq 1$, is a construct

$$\Pi = (V, T, C, \mu, w_1, \dots, w_m, (R_1, \rho_1), \dots, (R_m, \rho_m), i_0),$$

where:

- (i) V is an alphabet; its elements are called *objects*;
- (ii) $T \subseteq V$ (the *output* alphabet);
- (iii) $C \subseteq V, C \cap T = \emptyset$ (*catalysts*);

- (iv) μ is a membrane structure consisting of m membranes, with the membranes and the regions labelled in a one-to-one manner with elements of a given set Λ ; in this section we use the labels $1, 2, \dots, m$;
- (v) $w_i, 1 \leq i \leq m$, are strings representing multisets over V associated with the regions $1, 2, \dots, m$ of μ ;
- (vi) $R_i, 1 \leq i \leq m$, are finite sets of *evolution rules* over V associated with the regions $1, 2, \dots, m$ of μ ; ρ_i is a partial order relation over $R_i, 1 \leq i \leq m$, specifying a *priority* relation among rules of R_i .
An evolution rule is a pair (u, v) , which we will usually write in the form $u \rightarrow v$, where u is a string over V and $v = v'$ or $v = v'\delta$, where v' is a string over

$$\{a_{\text{here}}, a_{\text{out}}, a_{in_j} \mid a \in V, 1 \leq j \leq m\},$$

and δ is a special symbol not in V . The length of u is called *the radius* of the rule $u \rightarrow v$. (The strings u, v are understood as representations of multisets over V , in the natural sense.)

- (vii) i_0 is either a number between 1 and m and then it specifies the *output* membrane of Π , or it is equal to ∞ , and then it indicates that the output is read in the outer region.

When presenting the evolution rules, the indication “here” is in general omitted. Remember that the multiset associated with a string w is denoted by $M(w)$.

If Π contains rules of radius greater than one, then we say that Π is a system *with cooperation*. Otherwise, it is a *non-cooperative* system. A particular class of cooperative systems is that of *catalytic* systems: the only rules of a radius greater than one are of the form $ca \rightarrow cv$, where $c \in C, a \in V - C$, and v contains no catalyst; moreover, no other evolution rules contain catalysts (there is no rule of the form $c \rightarrow v$ or $a \rightarrow v_1cv_2$, for $c \in C$). A system is said to be *propagating* if there is no rule which diminishes the number of objects in the system (note that this can be done by “erasing” rules, but also by sending objects out of the skin membrane).

Of course, any of the multisets $M(w_1), \dots, M(w_n)$ can be empty (that is, any w_i can be equal to λ) and the same is valid for the sets R_1, \dots, R_n and their associated priority relations ρ_i .

The components μ and w_1, \dots, w_m of a P system define a super-cell. Graphically, we will draw a P system by representing its underlying super-cell, and also adding the rules to each region, together with the corresponding priority relation. In this way, we can have a complete picture of a P system, much easier to understand than a symbolic description.

The $(m + 1)$ -tuple (μ, w_1, \dots, w_m) constitutes the *initial configuration* of Π . In general, any sequence $(\mu', w'_{i_1}, \dots, w'_{i_k})$, with μ' a membrane structure obtained by removing from μ all membranes different from i_1, \dots, i_k (of

course, the skin membrane is not removed), with w'_j strings over V , $1 \leq j \leq k$, and $\{i_1, \dots, i_k\} \subseteq \{1, 2, \dots, m\}$, is called a *configuration* of Π .

An important detail that should be noted is that the membranes preserve the initial labeling in all subsequent configurations; in this way, the correspondence between membranes, multisets of objects, and sets of evolution rules is well specified by the subscripts of these elements.

A more compact and easy to read writing of a configuration, avoiding the use of subscripts for multisets and sets above is that where the objects of the multisets are written (using multisets or in the form of a string) directly in the region to which they belong; similarly, the rules are written in the region where they can act. This is in good correspondence with the graphical representation of a P system and we will use it especially for configurations where many components are empty.

For two configurations

$$C_1 = (\mu', w'_{i_1}, \dots, w'_{i_k}), \quad C_2 = (\mu'', w''_{j_1}, \dots, w''_{j_l}),$$

of Π we write $C_1 \Rightarrow C_2$, and we say that we have a *transition* from C_1 to C_2 , if we can pass from C_1 to C_2 by using the evolution rules appearing in R_{i_1}, \dots, R_{i_k} in the following manner (rather than a completely cumbersome formal definition we prefer an informal one, explained by examples).

Consider a rule $u \rightarrow v$ in a set R_{i_t} . We look to the region of μ' associated with the membrane i_t . If the objects mentioned by u , with the multiplicities specified by u , appear in w'_{i_t} (that is, the multiset $M(u)$ is included in $M(w'_{i_t})$), then these objects can evolve according to the rule $u \rightarrow v$. The rule can be used only if no rule of a higher priority exists in R_{i_t} and can be applied at the same time with $u \rightarrow v$. More precisely, we start to examine the rules in the decreasing order of their priority and assign objects to them. A rule can be used only when there are copies of the objects whose evolution it describes and which were not “consumed” by rules of a higher priority and, moreover, there is no rule of a higher priority, irrespective which objects it involves, which is applicable at the same step. Therefore, all objects to which a rule *can* be applied *must* be the subject of a rule application. All objects in u are “consumed” by using the rule $u \rightarrow v$, that is, the multiset $M(u)$ is subtracted from $M(w'_{i_t})$.

The result of using the rule is determined by v . If an object appears in v in the form a_{here} , then it will remain in the same region i_t . If an object appears in v in the form a_{out} , then a will exit the membrane i_t and will become an element of the region immediately outside it (thus, it will be adjacent to the membrane i_t from which it was expelled). In this way, it is possible that an object leaves the system: if it goes outside the skin of the system, then it never comes back. If an object appears in the form a_{in_q} , then a will be added to the multiset $M(w'_q)$, providing that a is adjacent to the membrane q . If a_{in_q} appears in v and membrane q is not one of the membranes delimiting “from below” the region i_t , then the application of the rule is not allowed.

If the symbol δ appears in v , then membrane i_t is removed (we say *dissolved*) and at the same time the set of rules R_{i_t} (and its associated priority relation) is removed. The multiset $M(w'_{i_t})$ is added (in the sense of multisets union) to the multiset associated with the region which was immediately external to membrane i_t . We do not allow the dissolving of the skin, because this means that the super-cell is lost and we no longer have a correct configuration of the system.

All these operations are performed in parallel, for all possible applicable rules $u \rightarrow v$, for all occurrences of multisets u in the region associated with the rules, for all regions at the same time. No contradiction appears because of multiple membrane dissolving, or because simultaneous appearance of symbols of the form a_{out} and δ . If at the same step we have a_{in_i} outside a membrane i and δ inside this membrane, then, because of the simultaneity of performing these operations, again no contradiction appears: we assume that a is introduced in membrane i at the same time when it is dissolved, thus a will remain in the region placed outside membrane i ; that is, from the point of view of a , the effect of a_{in_i} in the region outside membrane i and δ in membrane i is a_{here} .

Remark 3.1 The mode of evolving of objects in a super-cell provided with evolution rules as described above can be interpreted in the following – idealized – biochemical way. We have an organism, delimited by a skin (the skin membrane). Inside, there are organs and free molecules, organized hierarchically. The molecules and the organs float randomly in the “cytoplasmic liquid” of each membrane. Under specific conditions, the molecules evolve, alone or with the help of certain catalysts; these, of course, are not modified by the reactions. This is done in parallel, synchronously for all molecules (a universal clock is assumed to exist). The new molecules can remain in the same region where they have appeared, or can pass through the membranes delimiting this space, selectively. Some reactions not only modify molecules, but also break membranes. (We may imagine that certain chemicals are produced which break/dissolve the membrane.) When a membrane is broken, the molecules previously placed inside it will remain free in the larger space newly created, but the evolution rules of the former membrane are lost. The assumption is that the reaction conditions from the previous membrane are modified by the disappearance of the membrane and in the newly created space only the rules specific to this space can act. Of course, when the external membrane is broken, then the organism ceases to exist, its organs fall apart.

The priority among evolution rules can be interpreted as corresponding to different reactives of the involved chemicals. Moreover, we assume that the reactions also consume some resources different from objects (for example, energy); that is why when a rule of a higher priority is used, no rule of a lower priority can be used. \square

A sequence of transitions between configurations of a given P system

Π is called a *computation* with respect to Π . A computation is *successful* if and only if it halts, that is, there is no rule applicable to the objects present in the last configuration, and, if the system is provided with an output region $i_0 \neq \infty$, then the membrane i_0 is present as an elementary one in the last configuration of the computation. (Note that the output membrane was not necessarily an elementary one in the initial configuration.) The result of a successful computation can be the total number of objects present in the output membrane of the halting configuration, or $\Psi_T(w)$, where w describes the multiset of objects from T present in the output membrane in a halting configuration (Ψ_T is the Parikh mapping associated with T), or a language, as it will be defined immediately. The set of vectors $\Psi_T(w)$ for w describing the multiset present in the output membrane of a system Π in a halting configuration is denoted by $Ps(\Pi)$ (from “Parikh set”) and we say that it is *generated* by Π in the *internal mode*. When we are interested only in the number of objects present in the output membrane in the halting configurations of Π , we denote by $N(\Pi)$ the set of numbers “generated” in this way.

When no output membrane is specified ($i_0 = \infty$), that is, we observe the system from outside, then we collect the objects ejected from the skin membrane, in the order they are ejected. Using these objects, we form a string. When several objects are ejected at the same time, any permutation of them is considered. In this way, a string or a set of strings is associated with each computation, that is, a language is associated with the system.

We denote by $L(\Pi)$ the language computed by Π in the way described above. We say that it is *generated* by Π in the *external mode*.

Remark 3.2 A natural representation of the P systems architecture can be obtained in terms of trees, a fact also suggested by the way of writing a membrane structure by means of parentheses. Each membrane is associated with a node. If membrane i is placed inside membrane j and no intermediate membrane exists, then an arc from membrane j to membrane i is considered. In this way, the skin membrane is associated with the root of the tree, while the elementary membranes are associated with the leaves of the tree. To each node we also associate a multiset of objects, a set of evolution rules, and a priority relation. Thus, a node is fully described by a quadruple (i, M_i, R_i, ρ_i) . Using a rule means changing the multiset M_i and, possibly, the multisets placed in the parent and the children nodes of node i : a_{here} means adding a to M_i (after subtracting the multiset in the left hand side of the rule from the current multiset), a_{out} means to send a to the parent node, a_{in_j} means to send a to the child j , providing that it exists (otherwise the rule cannot be applied). Dissolving a membrane i means now removing node i ; its rules are removed, its objects are passed to the parent node; all the children of i become children of the parent node of i . The root cannot be removed.

In this new framework, a computation means the circulation of objects in the tree, with the change of the tree itself due to dissolving actions. In the

internal style, the result of a computation is collected in a leaf node. In the external style, the result is observed in the root (and it is a partially ordered sequence of objects which leave the P system through its skin).

In this book we do not make use of this tree representation; although mathematically appealing, it loses the biochemical intuition behind P systems. \square

3.2 Examples

We will consider a series of examples, in order to clarify the way of working of P systems.

Example 3.1 Consider the P system of degree 4

$$\begin{aligned}\Pi_1 &= (V, T, C, \mu, w_1, w_2, w_3, w_4, (R_1, \rho_1), (R_2, \rho_2), (R_3, \rho_3), (R_4, \rho_4), 4), \\ V &= \{a, b, b', c, f\}, \\ T &= \{c\}, \\ C &= \emptyset, \\ \mu &= [1[2[3]_3[4]_4]_2]_1, \\ w_1 &= \lambda, R_1 = \emptyset, \rho_1 = \emptyset, \\ w_2 &= \lambda, R_2 = \{b' \rightarrow b, b \rightarrow bc_{in_4}, r_1 : ff \rightarrow af, r_2 : f \rightarrow a\delta\}, \\ \rho_2 &= \{r_1 > r_2\}, \\ w_3 &= af, R_3 = \{a \rightarrow ab', a \rightarrow b'\delta, f \rightarrow ff\}, \rho_3 = \emptyset, \\ w_4 &= \emptyset, R_4 = \emptyset, \rho_4 = \emptyset.\end{aligned}$$

The initial configuration of the system is presented in Figure 3.2.

No object is present in membrane 2, hence no rule can be applied here. The only possibility is to start in membrane 3, using the objects a, f , present in one copy each. Using the rules $a \rightarrow ab', f \rightarrow ff$, in parallel for all occurrences of a and f currently available, after n steps, $n \geq 0$, we get n occurrences of b' and 2^n occurrences of f . In any moment, instead of $a \rightarrow ab'$ we can use $a \rightarrow b'\delta$ (note that we always have only one copy of a). In that moment we have $n+1$ occurrences of b' and 2^{n+1} occurrences of f and we dissolve membrane 3. The obtained configuration is

$$\begin{aligned}[1[2[b'^{n+1}f^{2^{n+1}}, b' \rightarrow b, b \rightarrow b(c, in_4), \\ r_1 : ff \rightarrow af, r_2 : f \rightarrow a\delta, r_1 > r_2, [4]_4]_2]_1.\end{aligned}$$

The rules of the former active membrane 3 are lost, the rules of membrane 2 are now active. Due to the priority relation, we have to use the rule $ff \rightarrow af$ as much as possible. In one step, we pass from b'^{n+1} to b^{n+1} , while the number of f occurrences is divided by two. In the next step, from b^{n+1} , $n+1$ occurrences of c are introduced in membrane 4 (each occurrence of the

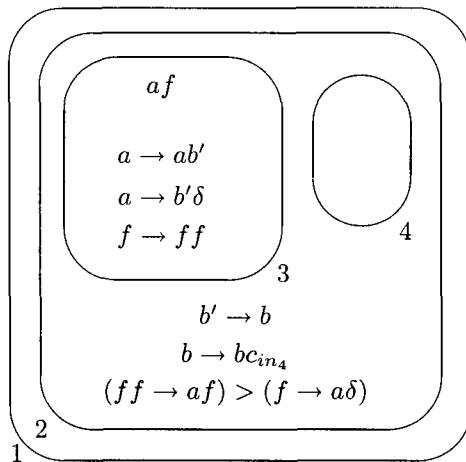


Figure 3.2: A P system generating $n^2, n \geq 1$.

symbol b introduces one occurrence of c). At the same time, the number of f occurrences is divided again by two. We can continue. At each step, further $n + 1$ occurrences of c are introduced in the output membrane. This can be done $n + 1$ steps: n times when the rule $ff \rightarrow af$ is used (thus diminishing the number of f occurrences to one), and one when using the rule $f \rightarrow a\delta$ (it may now be used). In this moment, membrane 2 is dissolved, which entails the fact that its rules are removed. No further step is possible. The obtained configuration is

$$[1 a^{2^{n+1}} b^{n+1}, [4 c^{(n+1)^2}]_4]_1.$$

Consequently, $N(\Pi_1) = \{m^2 \mid m \geq 1\}$.

Note that the system Π_1 is *propagating* (no object is removed by a rule or “lost” in the outer region) and it has only one cooperative rule.

The previous P system is a *generative* one: it starts from a unique initial configuration and, because of the non-deterministic evolution, it collects in its output membrane different values of $n^2, n \geq 1$. A variant of interest could be a system just *computing* n^2 for a given n . We leave to the reader the task of constructing such a system.

Example 3.2 In order to illustrate the possibly intricate work of a P system, we now consider a system which has a *decidability* task: we introduce in the input configuration two numbers, n and k , and ask whether or not n is a multiple of k . In the affirmative case, we will finish with one object in the output membrane; in the negative case we will have two objects in the output membrane.

The system is the following (of degree 3):

$$\begin{aligned}\Pi_2 &= (\{a, c, c', d, \dagger\}, \{a\}, \emptyset, \mu, \lambda, a^n c^k d, a, (R_1, \emptyset), (R_2, \emptyset), (\emptyset, \emptyset), 3), \\ \mu &= [{}_1[{}_2] {}_2[{}_3] {}_3] {}_1, \\ R_1 &= \{a \rightarrow a, d c' \rightarrow a_{in_3}\}, \\ R_2 &= \{ac \rightarrow c', ac' \rightarrow c, d \rightarrow d, d \rightarrow d\delta\}.\end{aligned}$$

The structure of Π_2 is better seen in Figure 3.3. In membrane 2 we subtract k from n , repeatedly (by the rules $ac \rightarrow c'$, $ac' \rightarrow c$: at each step, k copies of a disappear, while c is reproduced, primed or not primed, alternating the priming from a step to another one). At any time, the symbol d will introduce δ , and this membrane is dissolved.

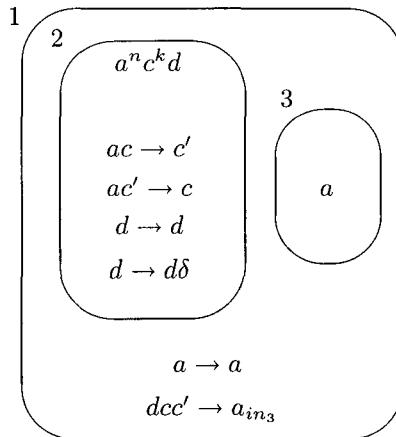


Figure 3.3: A P system deciding whether k divides n .

If an occurrence of a arrives in membrane 1, then the computation never ends. Therefore, we have to dissolve membrane 2 only after exhausting the n occurrences of a . If n is a multiple of k – and only in this case – then we never have both occurrences of c and of c' present simultaneously in membrane 2 (or in membrane 1, after dissolving membrane 2). Therefore, the rule $d c' \rightarrow a_{in_3}$ is used in membrane 1 if and only if n is not a multiple of k .

Note that this rule can be used at most once, because we have only one occurrence of d , and that the computation stops after using the rule $d c' \rightarrow a_{in_3}$. Similarly, the computation stops after using the rule $d \rightarrow d\delta$ in membrane 2 and we do not have both copies of c and of c' (no further step can be performed).

In conclusion, if the computation is correctly finished, then the output membrane contains two objects if and only if n is not a multiple of k (in the opposite case, we have here only one object).

We now pass to considering P systems with an external output.

Example 3.3 Consider the system with catalysts

$$\begin{aligned}\Pi_3 &= (\{a, c, d, e\}, \{a, d, e\}, \{c\}, [{}_1[{}_2]{}_2]{}_1, ca, \lambda, (R_1, \emptyset), (R_2, \emptyset), \infty), \\ R_1 &= \{a \rightarrow aa_{out}d_{in_2}, a \rightarrow a_{in_2}, cd \rightarrow cd_{out}e_{out}\}, \\ R_2 &= \{a \rightarrow \delta\}.\end{aligned}$$

For each copy of a which is sent outside the system, an occurrence of d is sent to the inner membrane, where it waits for the symbol a which determines the membrane dissolution. When the copies of d are released from the inner membrane, the catalyst c sends out one by one pairs de, ed . Thus, in our string we get consecutive substrings of the form de, ed , that is, the generated language is

$$L(\Pi_3) = \{a^n x_1 x_2 \dots x_n \mid n \geq 1, x_i \in \{de, ed\}, 1 \leq i \leq n\}.$$

The use of the catalyst is essential. If we would use non-cooperative rules of the form $d \rightarrow d'd_{out}, d' \rightarrow e_{out}$, where d' is a new symbol, then they must be used at the same time for all occurrences of d , hence we will obtain a string of the form $a^n d^n e^n$.

Example 3.4 The system

$$\begin{aligned}\Pi_4 &= (\{a, b\}, \{a, b\}, \emptyset, [{}_1]{}_1, a, (R_1, \emptyset), \infty), \\ R_1 &= \{a \rightarrow aa_{out}b, a \rightarrow a, b \rightarrow b, a \rightarrow a_{out}b, b \rightarrow b_{out}\},\end{aligned}$$

generates the Dyck language over $\{a, b\}$, that is, the language generated by the context-free grammar $G = (\{S\}, \{a, b\}, S, \{S \rightarrow \lambda, S \rightarrow SS, S \rightarrow aSb\})$. (Remember that when comparing the power of two mechanisms, the empty string is ignored.) We leave to the reader the task of checking this assertion.

Example 3.5 The system

$$\begin{aligned}\Pi_5 &= (\{a, b\}, \{a, b\}, \emptyset, [{}_1[{}_2]{}_2]{}_1, \lambda, ab, (R_1, \emptyset), (R_2, \emptyset), \infty), \\ R_1 &= \{a \rightarrow a_{out}, b \rightarrow b_{out}\}, \\ R_2 &= \{a \rightarrow aa, b \rightarrow bbb, a \rightarrow aad\},\end{aligned}$$

generates the language

$$L(\Pi_5) = \{x \in \{a, b\}^* \mid |x|_a = 2^n, |x|_b = 3^n, n \geq 0\}.$$

In the inner membrane one produces 2^n copies of a and 3^n copies of b , for any $n \geq 0$. When membrane 2 is dissolved, all these symbols are left free in membrane 1 and sent out at the same time. Thus, any sequence of them is a valid output string. (Note that this language is not an ETOL language.)

Example 3.6 Let p_1, p_2, \dots, p_m be different prime numbers and consider the system (of degree $m+1$):

$$\Pi_6 = (\{a\}, \{a\}, \emptyset, [1[2[3 \dots [_{m+1}]_{m+1} \dots]_3]_2]_1, \lambda, \dots, \lambda, a, (R_1, \emptyset), \dots, (R_{m+1}, \emptyset), \infty),$$

$$R_1 = \{a \rightarrow a_{out}\},$$

$$R_{i+1} = \{a \rightarrow a^{p_i}, a \rightarrow a^{p_i} \delta\}, \quad 1 \leq i \leq m.$$

The reader can easily verify that we obtain

$$L(\Pi_6) = \{a^{p_1^{i_1} p_2^{i_2} \dots p_m^{i_m}} \mid i_j \geq 1, 1 \leq j \leq m\}.$$

3.3 The Power of P Systems

First, let us fix some notations. The set of Parikh images of languages in a family FL is denoted by $PsFL$, while the family of permutation closures of languages in a family FL is denoted by pFL .

The family of languages $L(\Pi)$ generated (in the external mode) by P systems with priority, catalysts, and the membrane dissolving action, and of degree at most $m, m \geq 1$, is denoted by $LP_m(Pri, Cat, \delta)$; when one of the features $\alpha \in \{Pri, Cat, \delta\}$ is not present, we replace it with $n\alpha$. The union of all families $LP_m(\alpha, \beta, \gamma), m \geq 1$, is denoted by $LP_*(\alpha, \beta, \gamma)$, $\alpha \in \{Pri, nPri\}, \beta \in \{Cat, nCat\}, \gamma \in \{\delta, n\delta\}$. When collecting the result of a computation in an internal output membrane, we denote by $PsP_m(\alpha, \beta, \gamma)$ the family of vectors $Ps(\Pi)$ as defined in Section 3.1, with the same meaning for the involved parameters. When the output membrane is already elementary in the initial configuration, we write PsP' instead of PsP .

The following relations directly follow from the definitions (note that when we have only one membrane, the dissolving action is useless):

Lemma 3.1 (i) $XP_m(\alpha, \beta, \gamma) \subseteq XP_{m+1}(\alpha, \beta, \gamma)$, for all $m \geq 1$ and for all possible $\alpha, \beta, \gamma, X \in \{Ps, L\}$. Moreover, $XP_1(\alpha, \beta, \delta) = XP_1(\alpha, \beta, n\delta)$.

(ii) $XP_m(\alpha', \beta', \gamma') \subseteq XP_m(\alpha, \beta, \gamma)$, for all $m \geq 1$ and all possible α, β, γ such that α', β', γ' are either equal to α, β, γ or to $n\alpha, n\beta, n\gamma$, respectively, when n is not already present, $X \in \{Ps, L\}$.

(iii) $PsP'_m(\alpha, \beta, \gamma) \subseteq PsP_m(\alpha, \beta, \gamma)$, for all possible m and α, β, γ .

It is interesting to note that there are values of m, α, β, γ for which the inclusion in (iii) is proper: it was pointed out in [76] that $PsE0L \subseteq PsP_2(nPri, nCat, \delta)$, but we do not have $PsE0L \subseteq PsP'_2(nPri, nCat, \delta)$. Thus, at least for this case we do not have a normal form result, stating that it is enough to have the output membrane elementary in the initial configuration. We do not know whether or not this is true also in other cases. In view of the following results, this problem is worth investigating.

We are mainly interested in characterizations of computably enumerable languages, so most of the results mentioned below which are not of this type are given without proofs.

Lemma 3.2 $pLP_m(\alpha, \beta, \gamma) \subseteq LP_m(\alpha, \beta, \gamma)$, for all α, β, γ and $m \geq 1$.

Consider first the relationships between internal and external outputs.

Theorem 3.1 (The Bridge Theorem) (i) $PsP'_m(\alpha, \beta, \gamma) \subseteq PsLP_m(\alpha, \beta, \gamma)$, for all $m \geq 1$ and all possible α, β, γ .

(ii) $PsP_m(\alpha, \beta, \gamma) \subseteq PsLP_m(Pri, \beta, \gamma)$, for all $m \geq 1$ and all possible α, β, γ .

(iii) For all $m \geq 1$ and all possible α, β, γ , we have $PsLP_m(\alpha, \beta, \gamma) \subseteq PsP'_{m'}(\alpha, \beta, \gamma)$, for $m' = m + 1$. If $\gamma = n\delta$, then $m' = m$.

The reason why we need a further membrane in the general case mentioned in (iii) is the fact that the membrane dissolving is a dynamic event, from one computation to another one different membranes can be dissolved. In this way, we cannot fix in advance a membrane as an output membrane.

We will also see later that the question whether or not a given membrane is ever dissolved during the computations of a P system is not algorithmically solvable in the case of general systems (but it can be solved algorithmically in the case of systems of type $(nPri, nCat, \delta)$).

When “powerful enough” ingredients are used, characterizations of computably enumerable relations are obtained by using P systems of a simple structure. In view of the bridge between the internal and the external mode of collecting the output given by Theorem 3.1, we state this basic result only for the internal mode. Because this is an important result, we also give its proof.

This proof will use the characterization of computably enumerable languages by means of *matrix grammars with appearance checking*.

Such a grammar is a construct $G = (N, T, S, M, F)$, where N, T are disjoint alphabets, $S \in N$, M is a finite set of sequences of the form $(A_1 \rightarrow x_1, \dots, A_n \rightarrow x_n)$, $n \geq 1$, of context-free rules over $N \cup T$ (with $A_i \in N, x_i \in (N \cup T)^*$, in all cases), and F is a set of occurrences of rules in M (we say that N is the nonterminal alphabet, T is the terminal alphabet, S is the axiom, while the elements of P are called matrices).

For $w, z \in (N \cup T)^*$ we write $w \Rightarrow z$ if there is a matrix $(A_1 \rightarrow x_1, \dots, A_n \rightarrow x_n)$ in M and the strings $w_i \in (N \cup T)^*$, $1 \leq i \leq n+1$, such that $w = w_1, z = w_{n+1}$, and, for all $1 \leq i \leq n$, either $w_i = w'_i A_i w''_i$, $w_{i+1} = w'_i x_i w''_i$, for some $w'_i, w''_i \in (N \cup T)^*$, or $w_i = w_{i+1}$, A_i does not appear in w_i , and the rule $A_i \rightarrow x_i$ appears in F . (The rules of a matrix are applied in order, possibly skipping the rules in F if they cannot be applied; we say that these rules are applied in the *appearance checking* mode.) If $F = \emptyset$, then the grammar is said to be without appearance checking (and F is no longer mentioned).

We denote by \Rightarrow^* the reflexive and transitive closure of the relation \Rightarrow . The language generated by G is defined by $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$. The family of languages of this form is denoted by MAT_{ac} . When we use only grammars without appearance checking, then the obtained family is denoted by MAT .

It is known that $CF \subset MAT \subset MAT_{ac} = CE$. Further details about matrix grammars can be found in [75] and in [258].

A matrix grammar $G = (N, T, S, M, F)$ is said to be in the *binary normal form* if $N = N_1 \cup N_2 \cup \{S, \dagger\}$, with these three sets mutually disjoint, and that the matrices in M are of one of the following forms:

1. $(S \rightarrow XA)$, with $X \in N_1, A \in N_2$,
2. $(X \rightarrow Y, A \rightarrow x)$, with $X, Y \in N_1, A \in N_2, x \in (N_2 \cup T)^*$,
3. $(X \rightarrow Y, A \rightarrow \dagger)$, with $X, Y \in N_1, A \in N_2$,
4. $(X \rightarrow \lambda, A \rightarrow x)$, with $X \in N_1, A \in N_2$, and $x \in T^*$.

Moreover, there is only one matrix of type 1 and F consists exactly of all rules $A \rightarrow \dagger$ appearing in matrices of type 3. The symbols in N_1 are mainly used to control the use of rules of the form $A \rightarrow x$ with $A \in N_2$, while \dagger is a trap-symbol; once introduced, it is never removed. A matrix of type 4 is used only once, at the last step of a derivation.

According to Lemma 1.3.7 in [75], for each matrix grammar there is an equivalent matrix grammar in the binary normal form.

The following result describes the power of P systems with labelled membranes.

Theorem 3.2 $PsLP_m(Pri, Cat, n\delta) = PsLP_m(Pri, Cat, \delta) = PsLP_*(Pri, Cat, n\delta) = PsLP_*(Pri, Cat, \delta) = PsCE$, for all $m \geq 2$.

The only relation which needs a proof is the following one:

Lemma 3.3 (The Computational Completeness Lemma for P Systems) $PsCE \subseteq PsLP_2(Pri, Cat, n\delta)$.

Proof. Clearly, each set $Q \subseteq \mathbf{N}^k$, for some $k \geq 1$, can be identified with the language $L(Q) = \{a_1^{n_1} \dots a_k^{n_k} \mid (n_1, \dots, n_k) \in Q\}$ and Q is computably enumerable if and only if $L(Q)$ is computably enumerable. Take a matrix grammar with appearance checking, $G = (N, T, S, M, F)$, in the binary normal form, generating the language $L(Q)$, for a given computably enumerable set Q of vectors of natural numbers.

Assume that all matrices of forms 2, 3, 4 are labelled in a one-to-one manner, by m_1, m_2, \dots, m_s , for some $s \geq 1$.

We construct the P system with catalysts

$$\Pi = (V, T, \{c\}, [{}_1[{}_2]{}_2]_1, w_1, \lambda, (R_1, \rho_1), (\emptyset, \emptyset), 2),$$

where

$$w_1 = XAcZ, \text{ for } (S \rightarrow XA) \text{ the initial matrix in } M,$$

$$V = N_1 \cup N_2 \cup T \cup \{c, D, \dagger, Z\} \cup \{X_i, X'_i, X''_i \mid X \in N_1, 1 \leq i \leq s\},$$

and the set R_1 contains the following rules (h is the morphism defined by $h(\alpha) = \alpha$, $\alpha \in N_2$, and $h(a) = a_{in_2}$, for $a \in T$):

1. $X \rightarrow X_i$, for all $X \in N_1$ and $1 \leq i \leq s$.
2. $X_i \rightarrow Y'$, for $m_i : (X \rightarrow Y, A \rightarrow x)$ a matrix of type 2 in M .
3. $cA \rightarrow c h(x)D$, for $m_i : (X \rightarrow Y, A \rightarrow x)$ a matrix of type 2 in M .
4. $cD \rightarrow c$.
5. $cZ \rightarrow c\dagger$.
6. $\dagger \rightarrow \dagger$.
7. $Y' \rightarrow Y$, for all $Y \in N_1$.
8. $cX_i \rightarrow cY$, for $m_i : (X \rightarrow Y, A \rightarrow \dagger)$ a matrix of type 3 in M .
9. $A \rightarrow \dagger$, for all $A \in N_2$.
10. $X_i \rightarrow X'_i$, for $m_i : (X \rightarrow \lambda, A \rightarrow x)$ a matrix of type 4 in M .
11. $cA \rightarrow c h(x)D$, for $m_i : (X \rightarrow \lambda, A \rightarrow x)$ a matrix of type 4 in M .
12. $X'_i \rightarrow X''_i$, for $m_i : (X \rightarrow \lambda, A \rightarrow x)$ a matrix of type 4 in M .
13. $Z \rightarrow \lambda$.
14. $X''_i \rightarrow \lambda$, for $m_i : (X \rightarrow \lambda, A \rightarrow x)$ a matrix of type 4 in M .

The priorities are the following (at the same time, we give explanations about the work of Π):

- each rule of type 1 has priority over all rules of other types;
(In the presence of a symbol from N_1 no rule can be used, excepting a rule of type 1, which specifies a matrix to be simulated by the subscript of the symbol X .)
- each rule $X_i \rightarrow Y'$ of type 2 has priority over all rules of type 3 associated with matrices m_j with $j \neq i$, as well as over all rules of types 5, 9, 11, 13;
(If a symbol X_i is present, identifying a matrix $m_i : (X \rightarrow Y, A \rightarrow x)$ of type 2 from M , then the only rules which can be applied are $X_i \rightarrow Y'$, because only X_i is present, and $cA \rightarrow c h(x)D$, because all other rules are either of a lower priority than $X_i \rightarrow Y'$, or do not have symbols

to which they can be applied; note that always we have exactly one occurrence of the catalyst, hence the rule $cA \rightarrow c h(x)D$ can be used at most once; by using this rule, one occurrence of the symbol D is introduced.)

- the rule of type 4 has priority over the rule of type 5;
(This is a very important point of the construction, making a full use of the catalyst: if there is no occurrence of D in the multiset, then the rule $cZ \rightarrow c\dagger$ must be applied, introducing the trap-object \dagger which will evolve forever by the rule $\dagger \rightarrow \dagger$. Thus, at the same time with $X_i \rightarrow Y'$ we have to use the corresponding rule $cA \rightarrow c h(x)D$, which means that the use of the matrix m_i is correctly simulated. Note that the rule $cZ \rightarrow c\dagger$ cannot be used at the previous steps, because of the priority of $X_i \rightarrow Y'$ over it.)
- each rule $Y' \rightarrow Y$ of type 7, for $Y \in N_1$, has priority over all rules of types 3, 9, 11, 13;
(At the same time with the rule $cD \rightarrow c$, providing that D is present, we can use the rule $Y' \rightarrow Y$; no rule associated with a rule appearing in the second position in a matrix can be applied, the simulation of the matrix m_i is completed.)
- each rule $cX_i \rightarrow cY$ of type 8 has priority over all rules $cA \rightarrow c h(x)D$ associated with matrices of types 2 and 4, over all rules $B \rightarrow \dagger$ with $B \in N_2$ such that $B \neq A$, as well as over all rules of types 5, 11, 13;
(When the symbol X_i points to a matrix m_i of type 3, then the catalyst is “kept busy” by the rule $cX_i \rightarrow cY$, in order not to use the rule $cZ \rightarrow c\dagger$; no rule for evolving a symbol from N_2 can be used, because of the priority; if, however, the symbol A from $m_i : (X \rightarrow Y, A \rightarrow \dagger)$ appears in the current multiset, then the corresponding rule of type 9 should be used and the trap-object is introduced. In this way, we simulate the use of this rule in the appearance checking mode.)
- each rule $X_i \rightarrow X'_i$ of type 10 has priority over all rules of type 3, of type 11 associated with matrices m_j with $j \neq i$, as well as over all rules of types 5, 9, 13;
(When simulating the use of a matrix of type 4, at the last step of a derivation in G , we proceed as for matrices of type 2, with the difference that at the end we have also to remove the primed successors of X .)
- each rule $X'_i \rightarrow X''_i$ of type 12 has priority over all rules of types 3, 11, 13;
(After introducing X_i we replace it with X'_i and, at the same time, we use the corresponding rule $cA \rightarrow c h(x)D$. At the next step, we check whether or not D is introduced, that is, whether or not the simulation is correct. The symbol Z is still present, but it is not used, because of

the priorities mentioned above. At the same time, we check whether or not any nonterminal symbol from N_2 is still present: the rules $A \rightarrow \dagger$ are available and no other rule using symbols from N_2 can be used; if any rule $A \rightarrow \dagger$ can be applied, then it has to be applied.)

- each rule of type 14 has priority over $cZ \rightarrow c\dagger$;
(If a symbol X''_i is present, then this means that the computation is finished; we remove X''_i and we also remove the “semi-trap” object Z ; the rule $cZ \rightarrow c\dagger$ cannot be used.)

From the previous explanations, it is easy to see that each derivation in G can be simulated by a computation in Π and, conversely, each computation in Π corresponds to a derivation in G . It is worth mentioning that this is possible because we are interested in the Parikh set of a language, hence the order of symbols appearing in a sentential form of G is not important, only their presence matters (exactly as in a multiset). Moreover, at each moment when an occurrence of a terminal symbol is introduced, it is introduced directly into the output membrane. Nothing else can reach the output membrane. If the derivation is not correctly simulated or it is not terminal, then at least a rule can be further applied, in particular, the rule $\dagger \rightarrow \dagger$ if this symbol was produced. Thus, we can conclude that, because $L(G) = L(Q)$, we have $N(\Pi) = Q$. \square

We also have the following results, where P families are compared with families in the Chomsky or in the Lindenmayer area:

- Theorem 3.3** (i) $REG \subset LP_*(nPri, nCat, n\delta)$.
(ii) $PsP_1(nPri, nCat, n\delta) = PsP_*(nPri, nCat, n\delta) = PsCF$.
(iii) All languages in $LP(nPri, nCat, n\delta)$ are semilinear.

- Theorem 3.4** (i) $PsE0L \subseteq PsP_2(nPri, nCat, \delta)$.
(ii) $PsET0L \subseteq PsP_1(Pri, nCat, n\delta)$.
(iii) $PsP_*(nPri, nCat, \delta) \subseteq PsET0L$.

The last two inclusions, proved in [76], imply similar inclusions for $PsLP$ instead of PsP .

A natural question in this framework is whether or not the number of membranes (the degree of P systems) induces an infinite hierarchy. We have seen above that in several cases, this hierarchy collapses. In general, when no dissolving action is used, this hierarchy collapses (both in the internal and the external mode).

Theorem 3.5 $PsP_*(\alpha, \beta, n\delta) = PsP_m(\alpha, \beta, n\delta)$, for all $\alpha \in \{Pri, nPri\}$, $\beta \in \{Cat, nCat\}$, and $m \geq 2$. For families LP instead of PsP , the result above holds for $m = 1$.

With the exception of relation (i) in Theorem 3.3, up to now all results refer to Parikh images of languages. Here are two results dealing directly with languages:

Let us say that a language L is *strictly bounded* if $L \subseteq a_1^* a_2^* \dots a_k^*$, for some symbols a_1, a_2, \dots, a_k different from each other.

Theorem 3.6 *If L is a strictly bounded semilinear language, then $L \in LP_1(Pri, nCat, n\delta)$.*

If we also use catalysts, then more complex languages than the strictly bounded ones can be produced.

Theorem 3.7 *If h is a morphism and L is a strictly bounded semilinear language, then $h(L) \in LP_1(Pri, Cat, n\delta)$.*

3.4 Decidability Results

If, for a given P system, we could know in advance that a given elementary membrane will never be dissolved, then in Theorem 3.1(iii) we can use it as an output membrane, without being necessary to consider one further membrane (see the proof in [226]). Unfortunately, this is an undecidable question for systems of a general type.

Theorem 3.8 *It is undecidable whether or not a specified membrane of a P system of type (Pri, Cat, δ) will be ever dissolved.*

Proof. According to Theorem 3.2 and using the fact that for languages L over the one-letter alphabet we have $L = p(L)$, for each computably enumerable set of numbers $M \subseteq \mathbb{N}$, the language $l(M) = \{a^n \mid n \in M\}$ belongs to the family $LP_*(Pri, Cat, \delta)$. Let $\Pi = (V, \{a\}, C, \mu, w_1, \dots, w_m, (R_1, \rho_1), \dots, (R_m, \rho_m))$ be a P system such that $L(\Pi) = l(M)$, for an arbitrary set $M \subseteq \mathbb{N}$. We construct a new system, Π_t , where t is any integer $t \geq 0$, in the following way.

The total alphabet of Π_t is

$$V' = V \cup \{b, b', c, c', d, d', e, e', f\},$$

where c, c' are new catalysts, that is, the set of catalysts of Π_t is $C \cup \{c, c'\}$, and its output alphabet is $T = \{f\}$. The symbols $b, b', c, c', d, d', e, e', f$ are not in V .

Assume that the membranes in Π are labelled with $1', \dots, m'$ (the shell membrane is $1'$). Then, the system Π_t is as indicated in Figure 3.4, that is, it has:

$$\begin{aligned} \mu' &= [1[2 \mu]_2[3]_3[4]_4]_1, \\ w'_1 &= \lambda, \end{aligned}$$

$$\begin{aligned}
w'_2 &= b^{2t}f, \\
w'_3 &= cc'd', \\
w'_4 &= f, \\
R'_1 &= \{a \rightarrow a, e \rightarrow e_{in_3}, b \rightarrow b_{in_3}, f \rightarrow f_{out}, d \rightarrow d_{in_4}\}, \\
\rho'_1 &= \emptyset, \\
R'_2 &= \{r_1 : f \rightarrow f, r_2 : f \rightarrow f', r_3 : a \rightarrow e^2, f' \rightarrow \delta\}, \\
\rho'_2 &= \{r_1 > r_3, r_2 > r_3\}, \\
R'_3 &= \{r_4 : ce \rightarrow ce', r_5 : c'b \rightarrow c'b', r_6 : cb \rightarrow c\delta, \\
&\quad r_7 : c'e \rightarrow c'\delta, r_8 : d' \rightarrow d_{out}\}, \\
\rho'_3 &= \{r_4 > r_6, r_5 > r_7, r_6 > r_8, r_7 > r_8\}, \\
R'_4 &= \{d \rightarrow \delta\}, \\
\rho'_4 &= \emptyset.
\end{aligned}$$

All the components of Π remain unmodified. The shadowed membrane $1'$ in Figure 3.4 indicates that the contents of this membrane is not of interest for what follows, important is that as a result of the complete computations in this membrane we get strings a^n , for $n \in M$.

Claim: *Membrane 4 in system Π_t is dissolved if and only if $t \in M$.*

In order to prove this claim, and hence the theorem, let us examine the work of Π_t . The rule $f \rightarrow f$ in membrane 2 can be used an arbitrary number of times. In the meantime, from membrane $1'$ we can get the output of the system Π , in the form of a string a^r . In any moment, we can use the rule $f \rightarrow f'$. In parallel, further occurrences of a can exit from membrane $1'$. Assume that after the use of the rule $f \rightarrow f'$ we have in total n occurrences of a . At the next step we have to use the rule $f' \rightarrow \delta$, in parallel with the rule $a \rightarrow e^2$ (this last rule cannot be used before, because of the priority relations). In this way, membrane 2 is dissolved, its rules are removed (in particular, the rule $a \rightarrow e^2$ is no longer available). In membrane 1 we will have $2n$ occurrences of e , as well as the $2t$ occurrences of b introduced in the initial configuration of Π_t .

It is important to note that if any new occurrence of a was produced in the same step when membrane 2 was dissolved or any further occurrence of a will ever be expelled from membrane $1'$ at the next steps, then the computation in Π_t will never correctly end, because of the rule $a \rightarrow a$ present in membrane 1. Therefore, the work of Π must be finished when a^n is made available, that is, n must be an element of M .

The rules $e \rightarrow e_{in_3}, b \rightarrow b_{in_3}$ just pass to membrane 3 the $2n$ copies of e and the $2t$ copies of b . Note that if $n \neq t$, then $2n$ differs from $2t$ by at least 2. This is important below.

In membrane 3 we have two catalysts, c and c' , in one copy each. In every time unit, each catalyst decreases by one the number of copies of e and of b

(by priming them). This is done by the rules $ce \rightarrow ce'$, $c'b \rightarrow c'b'$ which have priority over all other rules in membrane 3.

If $n = t$, then the objects e, b are finished at the same time. This makes possible the use of the rule $d' \rightarrow d_{out}$ (no rule with a higher priority is applicable). From membrane 1, the symbol d enters membrane 4 and dissolves it. Just to have some output, the symbol f released in this way is sent out of the system.

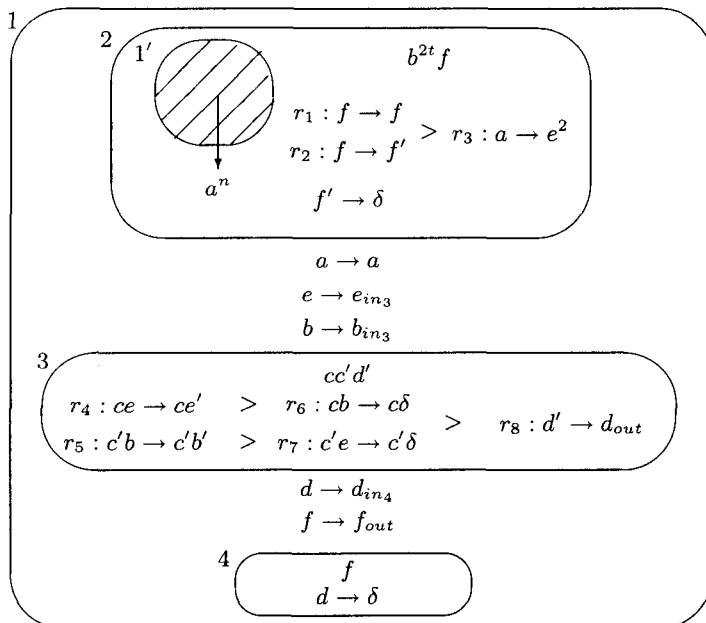


Figure 3.4: The system in the proof of the theorem.

Assume now that $n \neq t$. The two possibilities, $n > t$ and $t > n$ are handled in the same way, so we discuss only the case when $n > t$. This implies that $2n - 2t \geq 2$. After using the rules r_4, r_5 for $2t$ times, we have no further copy of b in membrane 3, but we still have at least two copies of e . One of them will be used by the rule $ce \rightarrow ce'$; because neither the rule $ce \rightarrow ce'$ can be used once more (the catalyst c is present in one copy only), nor the rule $c'b \rightarrow c'b'$ (no b is present), the rule $c'e \rightarrow c'\delta$ can be used (and should be used).

This means that membrane 3 is dissolved. Its content is released in membrane 1, its rules are lost. Thus, the symbol d' will remain unmodified, that is the rule $d \rightarrow d_{in_4}$ from membrane 1 cannot be applied. The computation stops here without dissolving membrane 4 (and without sending out any symbol, because f remains “locked” in membrane 4).

In conclusion, membrane 4 is dissolved if and only $t = n$, that is, if and only if $t \in M$. Because M is an arbitrary computably enumerable set, this is undecidable (if M is not computable, then its membership problem is not decidable). \square

Actually, we can also say something about the number of membranes in the system used in the previous proof: from Theorem 3.2 we know that Π is of degree 2, hence Π_t is of degree 6. We do not know which is the optimal result from this point of view.

From the previous proof, we can infer many other undecidability results. Note that the appearance of the symbol d , the use of the rule $d \rightarrow \delta$, and the fact whether or not f is sent out of membrane 1 depend on the fact whether or not $t \in M$, which is undecidable. Thus all these events are undecidable.

At the bottom of the hierarchy over the parameters α, β, γ in $PsLP(\alpha, \beta, \gamma)$ we can, however, decide whether or not a membrane is ever dissolved.

Theorem 3.9 *It is decidable whether or not a specified membrane in P systems of type $(nPri, nCat, \delta)$ is ever dissolved during a successful computation.*

Proof. Consider a system Π of type $(nPri, nCat, \delta)$ and a specified membrane i in it. Assume that the output alphabet of Π is T . We modify the system as follows. Take a new symbol, d , and add it to the output alphabet; denote by T' this new output alphabet. If there is a rule of the form $a \rightarrow v\delta$ in membrane i , then we replace it with $a \rightarrow vd\delta$. Now, we add the rule $d \rightarrow d_{out}$ to all membranes, including the skin one. Let us denote by Π' the system obtained in this way.

It is clear that $L(\Pi')$ contains all the strings in $L(\Pi)$ which were obtained at the end of computations where membrane i was not dissolved, as well as strings of the form $w_1dw_2\dots dw_k$ for $k \geq 1$ and $w_1w_2\dots w_k \in L(\Pi)$ which were obtained by computations where membrane i was dissolved. (The computation in Π' cannot stop before sending out the symbol d , because each region contains the rule $d \rightarrow d_{out}$. It is possible to use several rules of the form $a \rightarrow vd\delta$ at the step when membrane i is dissolved.)

From Theorem 3.4(iii) we know that $PsLP_*(nPri, nCat, \delta) \subseteq PsET0L$. This inclusion follows from the constructive proof of Theorem 3.1 and the relation $PsP_*(nPri, nCat, \delta) \subseteq PsET0L$ proved in [76]. This last relation is also proved in a constructive way. Therefore, starting from our system Π' , we can effectively construct an ET0L system G such that $\Psi_{T'}(L(\Pi')) = \Psi_{T'}(L(G))$. Consider the language $L = L(G) \cap T'^*\{d\}T'^*$. This is also an ET0L language (the family $ET0L$ is closed under intersection with regular languages). Clearly, L is non-empty if and only if the symbol d appears in at least one string of $L(G)$, that is, if and only if membrane i was ever dissolved during the computation of a string in $L(\Pi)$. The emptiness is decidable for ET0L languages, hence our problem is decidable. \square

There are two types of P systems, placed in between those considered in Theorems 3.8 and 3.9, for which the decidability of the membrane dissolving remains to be investigated: $(Pri, nCat, \delta)$ and $(nPri, Cat, \delta)$.

3.5 Rewriting P Systems

The P systems of the form considered up to now can be interpreted as using no data structure for codifying the information: the numbers are encoded as the cardinality of multisets, hence they are represented in the base one. This can be adequate to a biochemical implementation, but it looks inefficient from a classic point of view. Moreover, in this way we can deal only with problems on numbers, not (directly, without a number codification) with symbolic computations. That is why we look now for representing information by using a data structure of a standard type, *strings*.

Thus, in this section, instead of objects of an atomic type (i.e., without “parts”), we consider objects which can be described by finite strings over a given finite alphabet. The evolution of an object will then correspond to a transformation of the string. In this section we consider transformations in the form of rewriting steps, as usual in formal language theory.

The rules are also provided with indications on the target membrane of the produced string (we do no longer consider the membrane dissolving action, because, similarly to the case of Theorem 3.2, it will not be necessary in order to obtain computational completeness; of course, if for other purposes it will be useful/necessary to use this action, then it can be introduced in the same way as in the previous sections). Always we use only context-free rules, that is, the rules are of the form

$$(X \rightarrow v; tar),$$

where $X \rightarrow v$ is a context-free rule and $tar \in \{here, out, in_j\}$ (“tar” comes from “target”, j is the label of a membrane), with the obvious meaning: the string produced by using this rule will go to the membrane indicated by *tar*.

Note the important fact that a string is now a unique object, hence it passes through membranes as a unique entity, its symbols do not follow different itineraries, as it was possible for the objects in a multiset; of course, in the same region we can have several strings at the same time, but is irrelevant whether or not we consider multiplicities of strings: each string follows its own “fate”. That is why we do not speak here about multiplicities. In this framework, also the catalysts are meaningless.

Consequently, we obtain a language generating mechanism of the form

$$\Pi = (V, T, \mu, L_1, \dots, L_n, (R_1, \rho_1), \dots, (R_n, \rho_n), i_0),$$

where V is an alphabet, $T \subseteq V$, μ is a membrane structure, L_1, \dots, L_n are finite languages over V , R_1, \dots, R_n are finite sets of context-free evolution

rules, ρ_1, \dots, ρ_n are partial order relations over R_1, \dots, R_n , and i_0 is the output membrane.

We call such a system a *rewriting P system*.

The language generated by a system Π is denoted by $L(\Pi)$ and it is defined as explained in Section 3.1, with the differences specific to an evolution based on rewriting: we start from an initial configuration of the system and proceed iteratively, by transition steps performed by using the rules in parallel, to all strings which can be rewritten, obeying the priority relations, and collecting the terminal strings generated in a designated membrane, the output one.

Note that each string is processed by one rule only, the parallelism refers here to processing simultaneously all available strings by all applicable rules. If several rules can be applied to a string, maybe in several places each, then we take only one rule and only one possibility to apply it and consider the obtained string as the next state of the object described by the string. It is important to have in mind the fact that the evolution of strings is not independent of each other, but interrelated in two ways: (1) if we have priorities, a rule r_1 applicable to a string x can forbid the use of another rule, r_2 , for rewriting another string, y , which is present at that time in the same membrane; after applying the rule r_1 , if r_1 is not applicable to y or to the string x' obtained from x by using r_1 , then it is possible that the rule r_2 can now be applied to y ; (2) even without priorities, if a string x can be rewritten for ever, in the same membrane or on an itinerary through several membranes, and this cannot be avoided, then all strings are lost, because the computation never stops, irrespective of the strings collected in the output membrane and which cannot evolve further.

We denote by $RP_m(Pri)$ the family of languages generated by rewriting P systems of degree at most m , $m \geq 1$, using priorities; when priorities are not used, we replace Pri with $nPri$; the union of all families $RP_m(\alpha)$ is denoted by $RP_*(\alpha)$, $\alpha \in \{Pri, nPri\}$.

We do not recall here all known results about these families, but only the main one, the characterization of computably enumerable languages; some further details can be found in [218].

In order to illustrate the way of working in a rewriting P system, we consider an example (which also proves that the family $RP_2(nPri)$ contains non-context-free languages):

$$\begin{aligned} \Pi &= (\{A, B, a, b, c\}, \{a, b, c\}, [{}_1[{}_2]{}_2]{}_1, \emptyset, \{AB\}, (R_1, \emptyset), (R_2, \emptyset), 2), \\ R_1 &= \{(B \rightarrow cB; in_2)\}, \\ R_2 &= \{(A \rightarrow aAb; out), (A \rightarrow ab; here), (B \rightarrow c; here)\}. \end{aligned}$$

It is easy to see that $L(\Pi) = \{a^n b^n c^n \mid n \geq 1\}$ (if a string $a^i Ab^i c^i B$ is rewritten in membrane 2 to $a^i Ab^i c^{i+1}$ and then to $a^{i+1} Ab^{i+1} c^{i+1}$ and sent out, then it will never come back again in membrane 2, the computation stops, but the output membrane will remain empty). This is not a context-free language. \square

Theorem 3.10 $RP_m(Pri) = RP_*(Pri) = CE$, for all $m \geq 3$.

Of course, the only relation which needs a proof is that in the next lemma:

Lemma 3.4 (The Computational Completeness Lemma for Rewriting P Systems) $CE \subseteq RP_3(Pri)$.

Proof. Let $G = (N, T, S, M, F)$ be a matrix grammar with appearance checking in the binary normal form. We replace each matrix $(X \rightarrow \lambda, A \rightarrow x)$ of type 4, with $x \in T^*$, by the matrix $(X \rightarrow X', A \rightarrow x)$, which is considered of type 4'; we also add the matrices $(X' \rightarrow \lambda)$; X' is a new symbol associated with X . Clearly, the generated language is not changed. We assume the matrices of the types 2, 3, 4' labelled in a one-to-one manner with m_1, \dots, m_k .

We construct the following rewriting P system:

$$\begin{aligned} \Pi &= (V, T, \mu, L_1, L_2, L_3, (R_1, \rho_1), (R_2, \rho_2), (R_3, \rho_3), 2), \\ V &= N_1 \cup N_2 \cup \{E, Z, \dagger\} \cup T \cup \{X_i, X'_i \mid X \in N_1, 1 \leq i \leq k\}, \\ \mu &= [1[2]_2[3]_3]_1, \\ L_1 &= \{XAE\}, \text{ for } (S \rightarrow XA) \text{ the initial matrix of } G, \\ L_2 &= L_3 = \emptyset, \\ R_1 &= \{r_\alpha : (\alpha \rightarrow \alpha; here) \mid \alpha \in V - T, \alpha \neq E\} \\ &\cup \{r_0 : (E \rightarrow \lambda; in_2)\} \\ &\cup \{(X \rightarrow Y_i; in_2) \mid m_i : (X \rightarrow Y, A \rightarrow x) \text{ is a matrix of type 2}\} \\ &\cup \{(X \rightarrow Y_i; in_3) \mid m_i : (X \rightarrow Y, A \rightarrow \dagger) \text{ is a matrix of type 3}\} \\ &\cup \{(X \rightarrow X'_i; in_2), (X'_i \rightarrow \lambda; here) \mid m_i : (X \rightarrow X', A \rightarrow x) \\ &\quad \text{is a matrix of type 4'}\} \\ &\cup \{(Y_i \rightarrow Y; here), (Y'_i \rightarrow Y; here) \mid Y \in N_1, 1 \leq i \leq k\}, \\ \rho_1 &= \{r_\alpha > r_0 \mid \alpha \in V - T, \alpha \neq E\}, \\ R_2 &= \{r_i : (Y_i \rightarrow Y_i; here), r'_i : (A \rightarrow x; out) \mid m_i : (X \rightarrow Y, A \rightarrow x) \\ &\quad \text{is a matrix of type 2}\} \\ &\cup \{r_i : (X'_i \rightarrow X'_i; here), r'_i : (A \rightarrow x; out) \mid m_i : (X \rightarrow X', A \rightarrow x) \\ &\quad \text{is a matrix of type 4'}\} \\ \rho_2 &= \{r_i > r'_j \mid i \neq j, \text{ for all possible } i, j\}, \\ R_3 &= \{p_i : (Y_i \rightarrow Y'_i; here), p'_i : (Y'_i \rightarrow Y_i; here), \\ &\quad p''_i : (A \rightarrow \dagger; out) \mid m_i : (X \rightarrow Y, A \rightarrow \dagger) \\ &\quad \text{is a matrix of type 3}\} \\ &\cup \{p_0 : (E \rightarrow E; out)\}, \\ \rho_3 &= \{p''_i > p_i, p_i > p_0 \mid \text{for all possible } i\}. \end{aligned}$$

The system works as follows. Observe first that the rules $(\alpha \rightarrow \alpha; in_2)$ from membrane 1 change nothing, can be used for ever, and prevent the use

of the rule $(E \rightarrow \lambda; here)$, which sends the string to membrane 2, the output one.

Assume that in membrane 1 we have a string of the form XwE (initially, we have here the string XAE , for $(S \rightarrow XA) \in M$). In membrane 1 one chooses the matrix to be simulated, m_i , and one simulates its first rule, $X \rightarrow Y$, by introducing Y_i in the case of matrices of types 2, 3 and X'_i in the case of matrices of type 4'; the string is sent to membrane 2 if we deal with a matrix of types 2 or 4' (without a rule which has to be applied in the appearance checking mode), and to membrane 3 if we have to simulate a matrix of type 3.

In membrane 2 we can use the rule $(Y_i \rightarrow Y_i; here)$ forever. The only way to quit this membrane is by using the rule $A \rightarrow x$ appearing in the second position of a matrix of type 2. Due to the priority relation, this matrix should be exactly m_i as specified by the subscript of Y_i . Therefore, we can continue the computation only when the matrix is correctly simulated.

The process is similar in membrane 3: The rules $(Y_i \rightarrow Y'_i; here)$, $(Y'_i \rightarrow Y_i; here)$ can be used for ever. We can quit the membrane either by using a rule $(A \rightarrow \dagger; out)$ or by using the rule $(E \rightarrow E; out)$. In the first case the computation will never end. Because of the priority relation, such a rule must be used if the corresponding symbol A appears in the string. If this is not the case, then the rule $(Y_i \rightarrow Y'_i; here)$ can be used. If we now use the rule $(Y'_i \rightarrow Y_i; here)$, then we get nothing. If we use the rule $(E \rightarrow E; out)$, and this is possible because Y_i is no longer present, then we send out a string of the form Y'_iwE .

In membrane 1 we replace Y_i or Y'_i by Y , and thus the process of simulating the use of matrices of types 2, 3 can be iterated.

A slightly different procedure is followed for the matrices of type 4'; they are of the form $m_i : (X \rightarrow X', A \rightarrow x)$. In membrane 1 we use $(X \rightarrow X'_i; in_2)$ and the string arrives in membrane 2. Again the only way to leave this membrane is by using the associated rule $(A \rightarrow x; out)$. In membrane 1 we have to apply $(X'_i \rightarrow \lambda; here)$. If no symbol different from E and terminals is present, then we can apply the rule $(E \rightarrow \lambda; in_2)$. Thus, a terminal string is sent to membrane 2, where no rewriting can be done, the computation stops. If any nonterminal symbol is still present, then the computation will never halt, because of the rules $(\alpha \rightarrow \alpha; here)$ from membrane 1.

Therefore, we collect in the output membrane exactly the terminal strings generated by the grammar G , that is $L(G) = L(\Pi)$. \square

Without a proof, we mention the following result, related to the previous one:

Lemma 3.5 $MAT \subseteq RP_*(nPri)$.

Several problems are *open* in this area: Is the hierarchy $RP_m(nPri)$, $m \geq 1$, an infinite one? Is the result $CE \subseteq RP_3(Pri)$ optimal? Is the inclusion

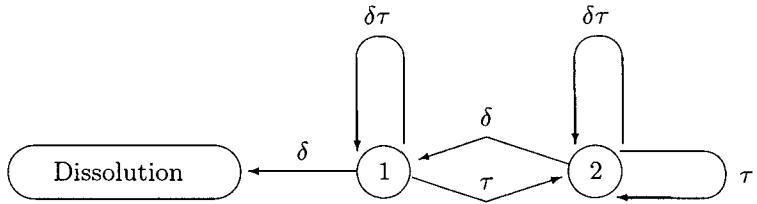
$MAT \subseteq RP_*(nPri)$ proper? (The difficulty in proving that $RP_*(nPri) \subseteq MAT$ lies in the dependence between the evolution of the words initially placed in a rewriting P system: even if a string has reached the output membrane and it cannot further evolve, in order to accept it we have to make sure that no other string present in the system can further evolve. This can easily be controlled in a matrix grammar with appearance checking, but we see no way to do it without using the appearance checking.)

3.6 P Systems with Polarized Membranes

The way of controlling the communication of objects by means of labels, in the form a_{in_j} , looks rather unrealistic. In real cells, the molecules can pass through membranes mainly because of concentration difference in neighbouring regions, or by means of electrical charges (ions can be transported in spaces of opposite polarization). This last variant is a much more restricted possibility as compared with the specification of the target membrane by its label: we only have two labels, + and −, associated in a non-injective way with the membranes. However, from the two Computational Completeness Lemmas (Lemmas 3.3 and 3.4) we know that systems with three membranes suffice. Three membranes means a skin membrane and two inner membranes which can be labelled with + and −. Consequently, using only polarized membranes we can still obtain computational completeness. To this aim, we have to use priority relations. However, the priority also looks a little bit “non-biochemical”, so it would be good to also avoid the use of this feature. Pleasantly enough, a variant of P systems with communication controlled by an “electrical charge” associated with each object and without a priority relation among evolution rules, still characterizes the computably enumerable sets of numbers, providing that a further feature is considered: controlling the thickness (permeability) of membranes by means of evolution rules.

The charge of objects can be “positive”, “negative” (identified with +, −, respectively), or “neutral”. An object marked with + will enter any of the membranes marked with − which are adjacent to the region where this object is produced; symmetrically, an object marked with − will enter a membrane marked with +, non-deterministically chosen from the set of reachable membranes. The neutral objects are not introduced in an inner membrane (they can only be sent out of the current membrane – the indications *here* and *out* are used as in the previous sections).

The control of membrane permeability can be achieved as follows: besides the action of dissolving a membrane (indicated by introducing the symbol δ), we also use the action of making a membrane thicker (this is indicated by the symbol τ). Initially, all membranes have the thickness 1. If a rule in a membrane of thickness 1 introduces the symbol τ , then the membrane becomes of thickness 2. A membrane of thickness 2 does not become thicker by using further rules which introduce the symbol τ , but no object can enter

Figure 3.5: The effect of actions δ, τ .

or exit it. If a rule which introduces the symbol δ is used in a membrane of thickness 1, then the membrane is dissolved; if the membrane had thickness 2, then it returns to thickness 1. If at the same step one uses rules which introduce both δ and τ in the same membrane, then the membrane does not change its thickness. These actions of the symbols δ, τ are illustrated in Figure 3.5.

We consider here only P systems with polarized membranes of a variable thickness with external output (so, the output region is no longer specified, it is always ∞). Such a system of degree $m, m \geq 1$, is a construct

$$\Pi = (V, T, C, \mu, w_1, \dots, w_m, R_1, \dots, R_m),$$

where all components are defined as in the previous sections, with the following differences:

- μ is a membrane structure consisting of m membranes (labelled, for reference only, with $1, 2, \dots, m$), where each membrane is marked with one of the symbols $+, -, 0$ (these markers are written as superscripts of the right square bracket representing the membrane, e.g. $[_1[_2]_2^+]_1^0$); all membranes in μ have thickness 1;
- the evolution rules in sets $R_i, 1 \leq i \leq m$, are of the forms $a \rightarrow v$ or $ca \rightarrow cv$, where a is an object from $V - C$ and $v = v'$ or $v = v'\delta$ or $v = v'\tau$, where v' is a string over

$$\{a_{\text{here}}, a_{\text{out}}, a^+, a^- \mid a \in V - C\},$$

and δ, τ are special symbols not in V . (As usual, the indication *here* is in general omitted.)

When applying a rule, if an object appears in v marked with *here*, then it remains in the same region; if we have a_{out} and the membrane has thickness 1, then a copy of the object a will be introduced in the region placed immediately outside; if we have a^+ (or a^-), then a copy of a is introduced in one of the membranes marked with $-$ (respectively $+$), of thickness one, and adjacent

to the region of the rule (if no such membrane exists, then the rule cannot be applied); if the special symbol δ appears in v and the membrane where we work has thickness 1, then this membrane is dissolved; in this way, all the objects in this region become elements of the region placed immediately outside, while the rules of the dissolved membrane are removed.

The communication of objects has priority on the actions δ, τ , in the sense that if at the same step one both sends a symbol through a membrane and one changes the thickness of that membrane, then one first transmits the symbol and after that one changes the thickness.

We denote by $LP_*^\pm(Cat, \delta, \tau)$ the family of languages generated by P systems as above, using catalysts and both actions indicated by δ, τ ; when one of the features $\alpha \in \{Cat, \delta, \tau\}$ is not used, we write $n\alpha$ instead of α . If only systems with at most m membranes are used, then we add the subscript m to LP.

The following example illustrates this definition. Consider the P system of degree 4

$$\begin{aligned}\Pi &= (V, T, C, \mu, w_1, w_2, w_3, w_4, R_1, R_2, R_3, R_4), \\ V &= \{a, a', b, b', c, d, \dagger\}, \\ T &= \{a\}, \\ C &= \{c\}, \\ \mu &= [{}_1[{}_2[{}_3[{}_4]_4^0]_3^0]_2^0]_1^0, \\ w_1 &= \lambda, R_1 = \{a \rightarrow a_{out}\}, \\ w_2 &= \lambda, R_2 = \{a \rightarrow a', a' \rightarrow a_{out}, b \rightarrow \dagger, d \rightarrow \tau, \dagger \rightarrow \dagger\}, \\ w_3 &= \lambda, R_3 = \{a \rightarrow a_{out}a, cb \rightarrow cb', cd \rightarrow cd\delta\}, \\ w_4 &= a^n cd, R_4 = \{a \rightarrow ab\delta\}.\end{aligned}$$

The initial configuration of the system is presented in Figure 3.6. With each label of a membrane, we present its polarity; here, all membranes are neutral. (Actually, the polarity is not used, all objects are neutral.)

The system works as follows. Initially, we have objects only in membrane 4, where n copies of a and one copy of each of c and d are present. The rule $a \rightarrow ab\delta$ should be applied to each of these n copies of a ; as a result we get n copies of b and the membrane is dissolved. The used rule is “lost” and the objects (n copies of a , n copies of b , one copy of c , and one of d) are left free in the next region, that associated with membrane 3. If the rule $cd \rightarrow cd\delta$ is used in a moment when copies of b are still present, then such a symbol will reach region 2, where the rule $b \rightarrow \dagger$ will be used. The symbol \dagger can evolve indefinitely by using the rule $\dagger \rightarrow \dagger$, that is, the computation never halts (hence we have no output). In order to avoid the use of this trap-rule, we have to make sure that we dissolve membrane 3 after transforming all symbols b in b' . This means that we have to use n times (the number of occurrences of b) the rule $cb \rightarrow cb'$; in parallel, we have to use n times the

rule $a \rightarrow a_{out}a$ for all copies of a which are present. Therefore, n copies of a are sent out of membrane 3 at each step.

In membrane 2, each a becomes a' and, at the next step, is sent out, to membrane 1. From membrane 1, each object a leaves the system, hence participates to the generated string.

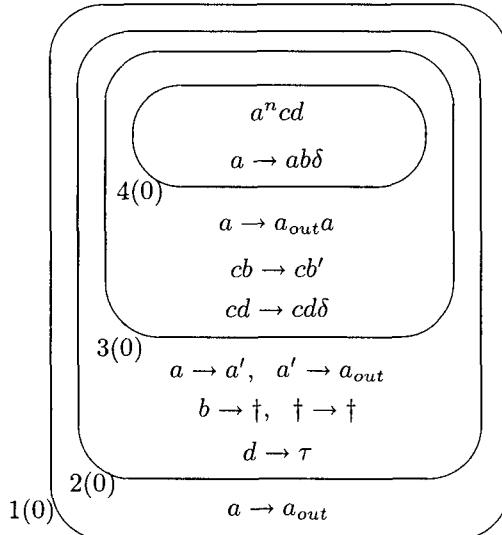


Figure 3.6: An example of a P system using action τ .

When d arrives in membrane 2, it must evolve by the rule $d \rightarrow \tau$. In this way, membrane 2 becomes thicker and from now on no symbol can leave it. This means that, after using the rule $d \rightarrow \tau$, the rule $a' \rightarrow a_{out}$ is no longer applicable. Thus, although from membrane 3 we send $n^2 + n$ copies of a to membrane 2 (n^2 during transforming b into b' and n at the step when we use the rule $cd \rightarrow cd\delta$), from membrane 2 to membrane 1 we send only n^2 copies of a (in the step when using $d \rightarrow \tau$ we also use $a \rightarrow a'$ and $a' \rightarrow a_{out}$, but from now on this latter rule will not be applied).

Consequently, the output consists of n^2 copies of the symbol a . We can say that the previous P system computes the mapping $f(n) = n^2$.

It is easy to modify the previous system in such a way to generate the (non-context-free) language $\{a^{n^2} \mid n \geq 1\}$: consider one further membrane inside membrane 4, with the rules $f \rightarrow af, f \rightarrow afcd\delta$, where f is a new object; when this membrane is dissolved, n copies of a , for some $n \geq 1$, are left free in membrane 4 (together with the catalyst c and the auxiliary symbol d ; of course, no object there initially is in membrane 4).

We do not start here a systematic study of the generative power of the

different variants of P systems with polarized membranes (with or without catalysts, with or without using the actions δ and τ , etc.), but we only provide a basic result, stating that they are computationally complete.

Theorem 3.11 $PsCE = PsLP_*^\pm(Cat, \delta, \tau)$.

As usual, we only prove the inclusion \subseteq :

Lemma 3.6 (The Computational Completeness Lemma for P Systems with Polarized Membranes) $PsCE \subseteq PsLP_*^\pm(Cat, \delta, \tau)$.

Proof. Consider a language $L \in CE$, $L \subseteq T^*$, and take a matrix grammar with appearance checking, $G = (N, T, S, M, F)$, in the binary normal form, generating this language.

We modify the grammar G by replacing each rule $X \rightarrow \lambda$ in matrices of type 4 by $X \rightarrow b$, where b is a new symbol. We denote by $G' = (N, T \cup$

- $\{b\}, S, M', F)$ the obtained grammar. Clearly, $L(G') = \{b\}L(G)$.

We label the matrices in M' in a one-to-one manner; assume that matrices m_1, \dots, m_k are of type 3 (with rules $A \rightarrow \dagger$ appearing in F) and matrices m_{k+1}, \dots, m_{k+n} are of types 2 and 4, for some $k \geq 0$ and $n \geq 1$.

We construct the system (of degree $2k + 3n - 1$)

$$\begin{aligned} \Pi = (V, T, C, \mu, & w_1, w_{1'}, w_2, w_{2'}, \dots, w_k, w_{k'}, w_{(k+1)}, w_{(k+1)'}, w_{(k+1)''}, \\ & w_{(k+2)}, w_{(k+2)'}, w_{(k+2)''}, \dots, w_{(k+n-1)}, w_{(k+n-1)'}, \\ & w_{(k+n-1)''}, w_{(k+n)'}, w_{(k+n)''}, \\ & R_1, R_{1'}, R_2, R_{2'}, \dots, R_k, R_{k'}, R_{(k+1)}, R_{(k+1)'}, R_{(k+1)''}, \\ & R_{(k+2)}, R_{(k+2)'}, R_{(k+2)''}, \dots, R_{(k+n-1)}, R_{(k+n-1)'}, \\ & R_{(k+n-1)''}, R_{(k+n)'}, R_{(k+n)''}), \end{aligned}$$

with the following components:

$$\begin{aligned} V &= \{X, X', X'', X''', X^{iv}, \bar{X} \mid X \in N_1\} \\ &\cup \{A, A', A'', A''', A^{iv}, \bar{A}, A^{(1)}, A^{(2)} \mid A \in N_2\} \\ &\cup \{D, D', D'', \bar{D}, b, c, \dagger\} \cup T, \end{aligned}$$

$$C = \{c\},$$

$$\begin{aligned} \mu = [&[1[1',]_1^-][2[2',]_2^-] \cdots [k[k',]_{k'}^-][(k+1)[(k+1)',]_{(k+1)'}^-][(k+1)'']_{(k+1)''}^-]_{(k+1)'}^- \\ &\cdots [(k+n-1)[(k+n-1)',]_{(k+n-1)'}^-][(k+n-1)'']_{(k+n-1)''}^-]_{(k+n-1)'}^- \\ &[(k+n)[(k+n)',]_{(k+n)'}^-][(k+n)'']_{(k+n)''}^+]_{(k+n-1)}^+ \cdots]_{(k+1)}^+]_k^+ \cdots]_2^+]_1^0, \end{aligned}$$

$$w_1 = XA, \text{ for } (S \rightarrow XA) \text{ the initial matrix of } G,$$

$$w_{i'} = \lambda, 1 \leq i \leq k,$$

$$w_i = \lambda, 2 \leq i \leq k+n-1,$$

$$w_{(k+i)'} = c, \quad 1 \leq i \leq n,$$

$$w_{(k+i)''} = \lambda, \quad 1 \leq i \leq n,$$

$$\begin{aligned} R_1 &= \{X \rightarrow X^+, X \rightarrow X^-, \bar{X} \rightarrow X \mid X \in N_1\} \\ &\cup \{A \rightarrow A^+, A \rightarrow A^-, \bar{A} \rightarrow A, A'' \rightarrow \dagger, A^{(2)} \rightarrow \dagger \mid A \in N_2\} \end{aligned}$$

$$\cup \{\bar{a} \rightarrow a_{out} \mid a \in T\} \cup \{\dagger \rightarrow \dagger\},$$

$$\begin{aligned} R_i &= \{X \rightarrow X^+, X \rightarrow X^-, \bar{X} \rightarrow \bar{X}_{out} \mid X \in N_1\} \\ &\cup \{A \rightarrow A^+, A \rightarrow A^-, \bar{A} \rightarrow \bar{A}_{out}, A'' \rightarrow \dagger, A^{(2)} \rightarrow \dagger \mid A \in N_2\} \\ &\cup \{\bar{a} \rightarrow \bar{a}_{out} \mid a \in T\} \cup \{\dagger \rightarrow \dagger\}, \end{aligned}$$

for $i = 2, \dots, k + n - 1$,

$$\begin{aligned} R_{j'} &= \{X \rightarrow Y'\tau, Y' \rightarrow Y'', Y'' \rightarrow \bar{Y}_{out}, A \rightarrow \dagger, \dagger \rightarrow \dagger\} \\ &\cup \{Z \rightarrow \dagger \mid Z \in N_1, Z \neq X\} \\ &\cup \{B \rightarrow B^{(1)}, B^{(1)} \rightarrow B^{(2)}\delta, B^{(2)} \rightarrow \bar{B}_{out} \mid B \in N_2, B \neq A\}, \end{aligned}$$

for $i = 1, 2, \dots, k$, with $m_i = (X \rightarrow Y, A \rightarrow \dagger)$,

$$\begin{aligned} R_{j'} &= \{X \rightarrow Y'D^+\tau, Y' \rightarrow Y'', Y'' \rightarrow Y''', Y''' \rightarrow Y^{iv}, \\ &\quad Y^{iv} \rightarrow \bar{Y}^{(out)}, Y^{iv} \rightarrow \dagger, \dagger \rightarrow \dagger, cA \rightarrow cA', A' \rightarrow A''\delta, \\ &\quad A'' \rightarrow A''', A''' \rightarrow A^{iv}, A^{iv} \rightarrow h_{out}(x)\} \\ &\cup \{B \rightarrow B^+, \bar{B} \rightarrow \bar{B}_{out}, B^{(2)} \rightarrow \dagger \mid B \in N_2\} \\ &\cup \{Z \rightarrow \dagger \mid Z \in N_1, Z \neq X\}, \end{aligned}$$

where h_{out} is the morphism defined by $h_{out}(\alpha) = \bar{a}_{out}$,

for each $\alpha \in N_2 \cup T$,

$$\begin{aligned} R_{i''} &= \{D \rightarrow D'\tau\} \\ &\cup \{B \rightarrow B^{(1)}, B^{(1)} \rightarrow B^{(2)}\delta, B^{(2)} \rightarrow \bar{B}_{out} \mid B \in N_2\}, \\ &\quad \text{for } i = k + 1, \dots, k + n, \text{ with } m_i = (X \rightarrow Y, A \rightarrow x) \\ &\quad (\text{note that } Y \in N_1 \cup \{b\}). \end{aligned}$$

As in the previous example, once the symbol \dagger is introduced, the computation is lost, because it will never end.

The shape of the initial configuration – for the case $k = 2, n = 4$ – is indicated in Figure 3.7.

The system Π works as follows.

In the initial configuration we can apply a rule only in membrane 1. Consider an arbitrary step, when in membrane 1 we have one occurrence of a symbol $X \in N_1$ and several occurrences of symbols from N_2 (initially, we have here only one object from N_2). In other regions we have at most symbols \bar{b}, D' , and c (initially, only occurrences of the catalyst c).

All symbols in membrane 1 get an “electrical charge”, in a non-deterministic manner, by means of the rules of the form $X \rightarrow X^+, X \rightarrow X^-$ and $A \rightarrow A^+, A \rightarrow A^-$. This happens also in each of the membranes $2, 3, \dots, k + n - 1$. The symbols marked in this way have to be commu-

nicated to a membrane of an opposite polarity. Always we have only two membranes where the symbols can be moved, of opposite polarities (note that membrane $(k + n - 1)'$ is marked with $-$ and membrane $(k + n)'$ with $+$).

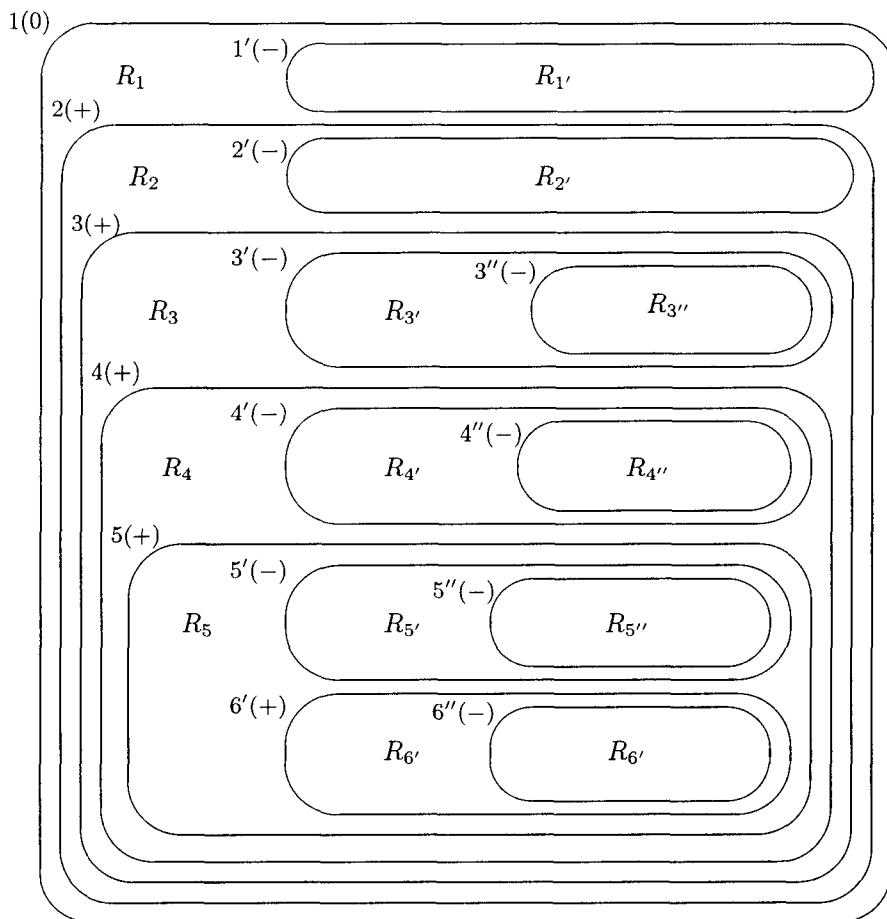


Figure 3.7: The shape of the system in the proof of lemma.

Assume that some symbols are moved in this way until reaching a membrane i' , for some $1 \leq i \leq k$, associated with a matrix $m_i = (X \rightarrow Y, A \rightarrow \dagger)$ from the grammar G' . We distinguish several cases:

1. If an occurrence of A is present, then the rule $A \rightarrow \dagger$ from $R_{i'}$ must be used, hence the computation never halts.

2. If a symbol $Z \in N_1$ different from X is present, then the rule $Z \rightarrow \dagger$ must be used and the computation never halts.
3. If X is not present, but some symbols $B \in N_2 - \{A\}$ are present, then we have to use the rules $B \rightarrow B^{(1)}, B^{(1)} \rightarrow B^{(2)}\delta$. The membrane is dissolved and the symbol $B^{(2)}$ is left free in region i . The rule $B^{(2)} \rightarrow \dagger$ is present here and it has to be used, hence the computation never halts.
4. Assume now that both X and A are present in region i' . At the first step, X is replaced with Y' , one also introduces the symbol τ , and all symbols $B \in N_2 - \{A\}$ are replaced by $B^{(1)}$. The membrane gets thickness 2. At the next step, we use the rules $Y' \rightarrow Y''$ and $B^{(1)} \rightarrow B^{(2)}\delta$, for all $B \in N_2 - \{A\}$ which are present in the membrane. (When simulating a derivation with respect to the grammar G' , in the system Π there is at least one symbol B as above, because the matrices of this form are not used at the last step of a derivation. We shall see immediately that the nonterminals must travel through membranes together, otherwise the computation will never end.) In this way, the membrane returns to thickness one. At the next step, all symbols Y and $B \in N_2 - \{A\}$ are sent out, in a barred form (and this is possible, because the membrane has the thickness 1). In this way, the matrix m_i has been correctly simulated: X is replaced with Y , providing that A is not present. We will see below that A is not only absent from membrane i' , but is was absent also from membrane 1 at the beginning of the simulation of the matrix m_i .

Let us now consider the case when some symbols enter a membrane j' for some $j = k + 1, \dots, k + n$, associated with a matrix $m_j = (X \rightarrow Y, A \rightarrow x)$, with $Y \in N_1 \cup \{b\}$ and $x \in (N_2 \cup T)^*$. We again distinguish several cases:

1. If a symbol $Z \in N_1$ different from X is present in membrane j' , then the rule $Z \rightarrow \dagger$ must be used and the computation never halts.
2. If the right symbol X is present, but A is not present, then in the first step we use the rule $X \rightarrow Y'D^+\tau$ (hence membrane j' becomes thicker and the object D is sent to membrane j'') and $B \rightarrow B^+$ for all symbols $B \in N_2$ which are present. At the next step, we use the rule $Y' \rightarrow Y''$ in membrane j' and the rules $D \rightarrow D'\tau, B \rightarrow B^{(1)}$ in membrane j'' (which becomes thicker). We continue by using the rule $Y'' \rightarrow Y'''$ in membrane j' and $B^{(1)} \rightarrow B^{(2)}\delta$ in membrane j'' (which returns to thickness 1). At the next step we use $Y''' \rightarrow Y^{iv}$ in membrane j' and $B^{(2)} \rightarrow \bar{B}_{out}$ in membrane j'' . We continue by using $Y^{iv} \rightarrow \dagger$ in membrane j' (the rule $Y^{iv} \rightarrow \bar{Y}_{out}$ cannot be used, because membrane j' is of thickness 2.) The computation will never stop.
3. If both X and A are present, then the rules we use at the next steps are the following:

- step 1: $X \rightarrow Y'D^+\tau$, $cA \rightarrow cA'$, $B \rightarrow B^+$,
for $B \in N_2$, in membrane j' (which becomes thicker);
- step 2: $Y' \rightarrow Y''$, $A' \rightarrow A''\delta$,
in membrane j' (which returns to thickness 1)
 $D \rightarrow D'\tau$, $B \rightarrow B^{(1)}$,
for $B \in N_2$, in membrane j'' (which becomes of thickness 2),
- step 3: $Y'' \rightarrow Y'''$, $A'' \rightarrow A'''$, in membrane j'' ,
 $B^{(1)} \rightarrow B^{(2)}\delta$, for $B \in N_2$, in membrane j'' (which returns to thickness 1),
- step 4: $Y' \rightarrow Y^{iv}$, $A''' \rightarrow A^{iv}$, in membrane j'' ,
 $B^{(2)} \rightarrow \bar{B}_{out}$, for $B \in N_2$, in membrane j'' ,
- step 5: $Y^{iv} \rightarrow \bar{Y}_{out}$, $A^{iv} \rightarrow h_{out}(x)$, $\bar{B} \rightarrow \bar{B}_{out}$,
for $B \in N_2$, in membrane j' .

In this way, the matrix m_j is correctly simulated.

In conclusion, we either correctly simulate the matrix m_j , or we introduce the trap-symbol \dagger . It is important to note that the symbol \dagger is introduced also when a symbol $B \in N_2$ enters this membrane without also having here the symbol X ; this symbol B can be any one from N_2 (equal or not to A). When simulating a matrix $(X \rightarrow \lambda, A \rightarrow x)$ at the last step of a derivation in G , in G' we use the matrix $(X \rightarrow b, A \rightarrow x)$, hence $Y = b$.

If the rule $cA \rightarrow cA'$ is not used at the first step, then the computation never stops: at step 5 we cannot use the rule $Y^{iv} \rightarrow \bar{Y}_{out}$, because membrane j' remains of thickness 2; this implies that the rule $Y^{iv} \rightarrow \dagger$ must be used and the derivation will never stop.

4. If no symbol from N_1 is present, but some symbols from N_2 are present in membrane j' , then we have two subcases.

If also A is present and we use the rule $cA \rightarrow cA'$ in membrane j' , then, at the next step we have to use the rule $A' \rightarrow A''\delta$, the membrane is dissolved, and A'' is left free in membrane j . Here, the rule $A'' \rightarrow \dagger$ must be used.

If we do not use the rule $cA \rightarrow cA'$, then all symbols are moved to membrane j'' by means of rules $B \rightarrow B^+$. In membrane j'' we use the rules $B \rightarrow B^{(1)}$, $B^{(1)} \rightarrow B^{(2)}\delta$ and this membrane is dissolved. The symbols $B^{(2)}$ are left free in membrane j' , where the rules $B^{(2)} \rightarrow \dagger$ must be used.

Consequently, we either correctly simulate the matrix m_j , or the trap symbol \dagger is introduced and the computation never ends.

Note that the barred symbols different from \bar{b} are always sent out of membranes $2, 3, \dots, k+n-1$, until reaching membrane 1. In this membrane,

the nonterminals lose the bars (while \bar{a} , for $a \in T$, sends out of the system a copy of a). In this way, the process can be iterated.

We emphasize again that if some symbols from N_2 arrive in a membrane i' , for any i , and the corresponding symbol X is not present, then the computation cannot stop. This means that always the symbols must travel together, namely together with the symbol from N_1 present in the current configuration. This ensures that the simulation of matrices ($X \rightarrow Y, A \rightarrow \dagger$) is correctly done: we are sure that A is not present, because all symbols from N_2 are present in the same membrane.

Thus, the equality $\Psi_T(L(\Pi)) = \Psi_T(L(G))$ follows (the symbol b does not exit the system). \square

Note that the system above has a number of membranes which depend on the number of matrices in the starting matrix grammar. It is an *open problem* to find a bound to the degree of P systems of this type which are able to characterize the Parikh sets of computably enumerable languages.

3.7 Normal Forms

The tree describing the membrane structure of the system involved in the proof of Lemma 3.6 is of the form indicated in Figure 3.8 (actually, we have considered the precise case of the membrane structure from Figure 3.7). The number of nodes depends on the number of matrices in the starting grammar; branches of type i, i' are associated with matrices used in the non-appearance checking manner and branches of type i, i', i'' are associated with matrices containing rules used in the appearance checking manner.

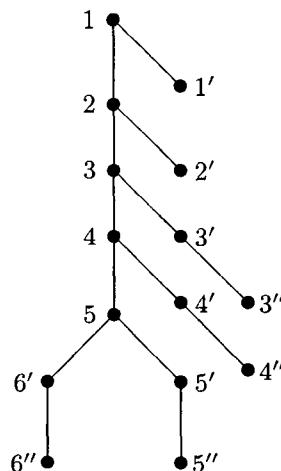


Figure 3.8: The shape of the tree in the previous proof.

It is a natural question (also of a possible practical importance) whether or not we can obtain the same result, but using a tree of a simpler form, for instance, a linear one, without any branch. Note that the membrane structure described by such a tree is of the form in Figure 3.9.

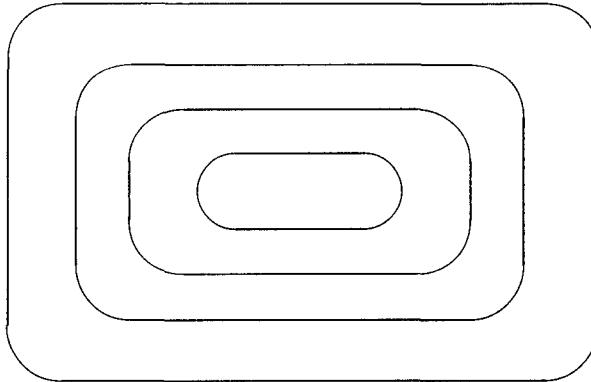


Figure 3.9: A membrane structure described by a linear tree.

We do not have an answer to this question for P systems as considered in the previous section, but we will obtain such a normal form providing that a more powerful type of catalysts is used, namely allowing them to have a “short memory”: each catalyst c is allowed to have two states, c and \bar{c} ; the rules involving catalysts always switch among these states. Thus, the allowed rules are of the forms: $ca \rightarrow \bar{c}v$, and $\bar{c}a \rightarrow cv$. We say that we use *flip-flop* catalysts. We indicate the use of such bi-stable catalysts by writing $2Cat$ instead of Cat . If the number of catalysts is bounded by a constant r , then the corresponding families of languages are denoted by $LP_m^\pm(2Cat_r, \delta, \tau)$, possibly with the subscript m replaced by $*$.

We also look for an “orthogonal normal form”, where the tree is of a minimal depth, a star. In the same framework as before – with bi-stable catalysts and actions δ, τ – we will obtain a normal form theorem similar to that announced above.

There is here an important point. In systems of the type described in Figure 3.9 there is no need to give any target indication, always we have exactly one lower membrane (excepting the central membrane) and one upper membrane (excepting the skin membrane). Thus, when we want to communicate, we can only indicate *in* and *out*, without any other information. (Specifically, we write a_{in}, a_{out} if we want to send a into any lower level membrane or out of the current membrane, respectively.) What about using such loose indications in systems with a non-linear graph and allowing the symbols to go to any membrane accessible in the indicated direction? (Of course, always an object can pass through one membrane only.) Surprisingly enough, in our setup this does not decrease the power; furthermore, a representation of *pCE*

is obtained by systems using a membrane structure of depth two, hence with a star graph. Note that in the case of a star, the ambiguity of an indication of the form a_{in} introduced by a rule in the skin membrane is maximal, all second level membranes can be the target of the symbol a .

When dealing with only *in/out* indications, no sign $+$, $-$ will be associated with objects and membranes; the corresponding families are denoted by $LP_*^{i/o}(2Cat, \alpha, \beta), \alpha \in \{\delta, n\delta\}, \beta \in \{\tau, n\tau\}$.

Let us first prove the fact that the bi-stable catalysts are very powerful (if their number is not restricted).

Theorem 3.12 $pCE \subseteq LP_1^{i/o}(2Cat, n\delta, n\tau)$.

Proof. We use again the equality $MAT_{ac} = CE$. Let us consider a matrix grammar with appearance checking, $G = (N, T, S, M, F)$, in the binary normal form, that is, with $N = N_1 \cup N_2 \cup \{S, \dagger\}$ and matrices of the four types specified in Section 3.3. Assume that the two-rules matrices of M are labelled in a one-to-one manner by m_1, \dots, m_n ; that is, we assume that we have n matrices of this type.

We construct the P system (of degree one)

$$\Pi = (V, T, C, \mu, w_1, R_1),$$

with

$$\begin{aligned} V &= N_1 \cup N_2 \cup T \cup C \cup \{b, d, e, e', e'', f, \dagger\} \\ &\cup \{E_i \mid 1 \leq i \leq n\} \cup \{X' \mid X \in N_1\}, \\ C &= \{c_i, \bar{c}_i \mid 0 \leq i \leq n\}, \\ \mu &= []_1, \\ w_1 &= XAbec_0c_1c_2\dots c_n, \text{ for } (S \rightarrow XA) \text{ the initial matrix of } G, \end{aligned}$$

and R_1 containing the following rules:

- For each matrix m_i of the form $(X_i \rightarrow z_i, A_i \rightarrow x_i)$, with $X_i \in N_1, z_i \in N_1 \cup \{\lambda\}, A_i \in N_2, x_i \in (N_2 \cup T)^*$, we consider the rules:

$$\begin{aligned} X_i &\rightarrow X'_i, \\ c_i X'_i &\rightarrow \bar{c}_i, \\ \bar{c}_i A_i &\rightarrow c_i e_i, \\ e_i &\rightarrow z_i x_i, \\ \bar{c}_i b &\rightarrow c_i \dagger. \end{aligned}$$

- For each matrix m_i of the form $(X_i \rightarrow Y_i, A_i \rightarrow \#)$, with $X_i, Y_i \in N_1, A_i \in N_2$, we consider the rules:

$$\begin{aligned} X_i &\rightarrow X'_i, \\ c_i X'_i &\rightarrow \bar{c}_i f, \\ \bar{c}_i A_i &\rightarrow c_i \dagger, \\ \bar{c}_i d &\rightarrow c_i Y_i. \end{aligned}$$

3. Moreover, we consider the following rules:

$$\begin{aligned} f &\rightarrow d, \\ a &\rightarrow a_{out}, \text{ for all } a \in T, \\ c_0e &\rightarrow \bar{c}_0e', \\ \bar{c}_0B &\rightarrow c_0B, \\ c_0e' &\rightarrow \bar{c}_0e'', \\ \bar{c}_0e'' &\rightarrow c_0e, \\ e'' &\rightarrow \dagger, \text{ for all } B \in N_2. \end{aligned}$$

The system works as follows.

At each moment, at most one symbol from N_1 is present, therefore at most one catalyst $c_i, 1 \leq i \leq n$, can be used. At the first step, the symbol from N_1 is primed.

If we apply a rule $c_1X'_i \rightarrow \bar{c}_i$, associated with a matrix $m_i : (X_i \rightarrow z_i, A_i \rightarrow x_i)$, hence not used in the appearance checking mode, then at the next step we have to use the rule $\bar{c}_iA_i \rightarrow c_iE_i$, otherwise the trap-object \dagger is introduced and the computation will never stop. At the fourth step, the symbols of $Y_i x_i$ are introduced. Thus, the matrix m_i is correctly simulated (both its rules are simulated).

If we apply a rule $c_iX'_i \rightarrow \bar{c}_i f$, associated with a matrix $m_i : (X_i \rightarrow Y_i, A_i \rightarrow \#)$, hence used in the appearance checking mode, then at the next step we either apply the rule $\bar{c}_iA_i \rightarrow c_i\dagger$, if A_i is present, or the catalyst \bar{c}_i is not modified, if A_i is not present. In the first case, the computation never halts. The symbol f is immediately transformed in d ; thus, at the third step, the rule $\bar{c}_id \rightarrow c_iY_i$ can be used. In the former case the computation will never stop, in the latter case the matrix is correctly simulated.

The simulation of any matrix takes always four steps.

The process can be iterated. Note that in both cases the symbol Y_i is available only after completing the simulation of a matrix and that no symbol from N_2 can be processed if we do not first process a symbol from N_1 .

The terminal symbols can be sent out of the system at any moment, so any permutation of a string can be obtained. As long as any symbol $B \in N_2$ is present, the computation can continue. This is ensured by the rules which involve the catalyst c_0 : in cycles of four steps, at step two, a symbol $B \in N_2$ is used by the rule $\bar{c}_0B \rightarrow c_0B$. Note that the simulation of matrices use symbols from N_2 at the third step, hence the simulation is not confused by the use of rules in group 3. Moreover, at step four we have to use the rule $\bar{c}_0e'' \rightarrow c_0e$ (instead of $\bar{c}_0B \rightarrow c_0B$ for some $B \in N_2$), otherwise the trap-object is introduced. Therefore, on the one hand, we have to send out all terminal symbols, on the other hand, a computation stops only if it corresponds to a terminal derivation in G .

Consequently, $p(L(G)) = L(\Pi)$. □

Corollary 3.1 $PsCE = PsLP_*^{i/o}(2Cat, n\delta, nr)$.

In the previous construction, the number of catalysts can be arbitrarily large, it depends on the number of matrices in the starting grammar. If we bound the number of catalysts we use, then we obtain a similar result, but without having a bound on the number of membranes. Thus, a trade-off between the number of membranes and the number of catalysts seems to hold.

Theorem 3.13 $pCE \subseteq LP_*^{i/o}(2Cat_2, \delta, \tau)$; moreover, P systems with a linear underlying tree suffice.

Proof. As in the previous proof, consider a matrix grammar $G = (N, T, S, M, F)$ in the binary normal form, with n matrices m_1, \dots, m_n .

We construct the P system (of degree $n + 1$)

$$\Pi = (V, T, C, \mu, w_0, w_1, \dots, w_n, R_0, R_1, \dots, R_n)$$

(the skin membrane is labelled with 0) with the following components:

$$V = N_1 \cup N_2 \cup T \cup C \cup \{d_1, d_2, d'_1, d'_2, d''_1, d''_2, e, \dagger\}$$

$$\cup \{\alpha', \alpha'', \alpha''', \bar{\alpha} \mid \alpha \in N_1 \cup N_2\}$$

$$\cup \{\alpha^{iv} \mid \alpha \in N_1 \cup N_2 \cup T\}$$

$$\cup \{(\alpha, i) \mid \alpha \in N_1 \cup N_2, 1 \leq i \leq n\}$$

$$\cup \{f_i \mid 1 \leq i \leq n\},$$

$$C = \{c_1, c_2, \bar{c}_1, \bar{c}_2\},$$

$$\mu = [{}_0[{}_1[{}_2 \cdots [{}_{n-1}[{}_n]_n]_{n-1} \cdots]_2]_1]_0,$$

$$w_0 = XA, \text{ for } (S \rightarrow XA) \text{ being the initial matrix of } G,$$

$$w_i = c_1 c_2 f_i, 1 \leq i \leq n,$$

and with the sets R_0, R_1, \dots, R_n constructed as follows.

1. The set R_0 contains the following rules:

$$1. \alpha \rightarrow (\alpha, i)_{in}, \text{ for } \alpha \in N_1 \cup N_2, 1 \leq i \leq n$$

(send down nonterminals, at random addresses indicated by the second component of (α, i)),

$$2. a \rightarrow a,$$

$$a \rightarrow a_{out}, \text{ for } a \in T$$

(send out terminals, at any time after having them in the skin membrane),

$$3. f_i \rightarrow \dagger, \text{ for all } 1 \leq i \leq n,$$

$$\dagger \rightarrow \dagger$$

(trap-rules; once introduced, the symbol \dagger can evolve for ever).

2. Each set $R_i, 1 \leq i \leq n$, contains the following rules (slight differences for the cases $i = 1$ and $i = n$ will be mentioned below):

1. $(\alpha, j) \rightarrow (\alpha, j)_{in}$, for $\alpha \in N_1 \cup N_2, i < j \leq n$
(send down nonterminals, to the addresses specified in the skin membrane),
 2. $(\alpha, i) \rightarrow \bar{\alpha}$
(when a symbol sent to membrane i reaches this membrane, its barred version is introduced),
 3. $\bar{\alpha} \rightarrow \alpha\delta$, for $\alpha \in N_2$,
 $\bar{\beta} \rightarrow \beta\tau$, for $\beta \in N_1$
(these rules check whether or not all nonterminals are present in the same place; this is a very important point of the construction – see below complete explanations),
 4. $f_j \rightarrow \dagger$, for all $i < j \leq n$,
 $\dagger \rightarrow \dagger$
(if any symbol f_j with $j > i$ reaches membrane i , then the computation never stops),
 5. $\alpha^{iv} \rightarrow \alpha_{out}^{iv}$, for $\alpha \in N_1 \cup N_2 \cup T$
(the symbols marked with iv are sent to the upper membrane).
3. In the set R_1 , instead of rules of type 5 above we introduce the rules
- 4'. $\alpha^{iv} \rightarrow \alpha_{out}$, for $\alpha \in N_1 \cup N_2 \cup T$
(the symbols reach the skin membrane without any marking).

In the set R_n no rule of type 1 above is introduced.

4. If the matrix m_i is of the form $(X \rightarrow z, A \rightarrow x)$, for some $X \in N_1, A \in N_2, x \in (N_2 \cup T)^*$, and $z \in N_1 \cup \{\lambda\}$ (we cover at the same time both the case of nonterminal and of terminal matrices which are not used in the appearance checking mode), then in R_i we also introduce the following rules:

1. $\alpha \rightarrow \alpha'$, for $\alpha \in N_1 \cup N_2$,
 $c_1 X \rightarrow \bar{c}_1 h(z) d'_1 e$,
 $c_2 A \rightarrow \bar{c}_2 h(x) d'_2 e$,
where h is the morphism defined by

$$h(\alpha) = \begin{cases} \alpha', & \text{if } \alpha \in N_1 \cup N_2, \\ \alpha_{out}^{iv}, & \text{if } \alpha \in T \end{cases}$$

(we simulate the two rules of matrix m_i),

2. $\alpha' \rightarrow \alpha''$, for $\alpha \in N_1 \cup N_2$,
 $\bar{c}_1 d'_1 \rightarrow c_1 d''_1$,
 $\bar{c}_2 d'_1 \rightarrow c_2 d''_2$,
 $c_1 e \rightarrow \bar{c}_1 \dagger$,
 $c_2 e \rightarrow \bar{c}_2 \dagger$,

$$c_2 d'_1 \rightarrow \bar{c}_2 \dagger,$$

$$c_1 d'_2 \rightarrow \bar{c}_1 \dagger$$

(we check whether or not both rules were simulated; if only one of them was simulated, then the trap-symbol \dagger is introduced and the computation never ends; see more complete explanations below),

3. $\alpha'' \rightarrow \alpha'''$, for $\alpha \in N_1 \cup N_2$,

$$c_1 d''_1 \rightarrow \bar{c}_1,$$

$$c_2 d''_2 \rightarrow \bar{c}_2$$

(the auxiliary symbols d_1, d_2 , in their primed versions, are removed),

4. $\alpha''' \rightarrow \alpha_{out}^{iv}$, for $\alpha \in N_1 \cup N_2$,

$$\bar{c}_1 e \rightarrow c_1,$$

$$\bar{c}_2 e \rightarrow c_2$$

(also the two copies of the auxiliary symbol e are removed; the catalysts return to their non-barred state).

5. If the matrix m_i is of the form $(X \rightarrow Y, A \rightarrow \dagger)$, for $X, Y \in N_1, A \in N_2$ (hence with the second rule used in the appearance checking manner), then we introduce in R_i the following rules:

1. $X \rightarrow Y_{out}^{iv}$,

$$Z \rightarrow \dagger, \text{ for all } Z \in N_1 - \{X\},$$

2. $A \rightarrow \dagger,$

3. $\alpha \rightarrow \alpha_{out}^{iv}$, for all $\alpha \in N_2 - \{A\}$

(all nonterminals go out, except A ; if A is present, then the computation will never finish).

We claim that $p(L(G)) = L(\Pi)$, which would conclude the proof.

Let us examine the work of the system Π . As already sketched above, the skin membrane sends randomly the current nonterminal symbols to the inner membranes, by attaching to them target integers: each α is replaced with (α, i) , for some $1 \leq i \leq n$. Membranes $1, 2, \dots, n$ simulate the corresponding matrices m_1, m_2, \dots, m_n .

The symbols (α, i) are sent down until reaching the membrane i . This means exactly i steps (counting also the step when the rules $\alpha \rightarrow (\alpha, i)$ were used). When (α, i) reaches membrane i , we introduce the barred variant of α . It is important to note that all symbols sent to membrane i arrive in this membrane at the same time.

Our aim is to simulate in Π derivations of G . Initially, we have in the skin membrane the symbols XA . If the simulation is correct, then always we will have in our system only one occurrence of a symbol from N_1 . Assume that we start from such a situation, that is, from a multiset in membrane 0 which contains exactly one occurrence of a symbol from N_1 .

Suppose that a nonterminal $A \in N_2$ has reached a membrane i , but the currently available symbol $X \in N_1$ is not present in the same membrane.

Then, the rule $\bar{A} \rightarrow A\delta$ is used in membrane i , without also using a rule of the type $\bar{\beta} \rightarrow \beta\tau$, for some $\beta \in N_1$. In this way, the membrane is dissolved, the symbol f_i is left free in the upper membrane (or a membrane placed at a higher level, in the case when several membranes are dissolved at the same time). In the upper membrane (or any superior one) we can use the rule $f_i \rightarrow \dagger$ and the computation is never finished.

Consequently, all the nonterminals from N_2 present in the system must be together with the unique symbol from N_1 . This means that *all the nonterminals are together*. Although the skin membrane uses the rules $\alpha \rightarrow (\alpha, i)$ randomly, the computation will continue in a correct way only when all symbols get the same “address” i . This is a crucial observation for the good functioning of our system, for instance, in the case of simulating the matrices with appearance checking rules.

Let us now look how the matrices of G are simulated.

Consider first the case of a matrix m_i of the form $(X \rightarrow Y, A \rightarrow \dagger)$, hence with the second rule used in the appearance checking manner. Assume that all nonterminals have reached membrane i . We can change X with Y_{out}^{iv} and continue correctly only if no occurrence of A is present, otherwise the trap-symbol \dagger is introduced. If a symbol $Z \in N_1$ different from X is present (this means that membrane 0 has incorrectly guessed the membrane where the nonterminals are sent), then again the trap-symbol \dagger is introduced. Thus, we either correctly simulate the matrix, or the computation will never end. Note here the important role of the maximal parallelism: if a rule *can* be applied to an available symbol, then it *must* be applied.

Assume now that we are in a membrane i associated with a matrix $m_i = (X \rightarrow Y, A \rightarrow x)$. If we use only rules of the forms $\alpha \rightarrow \alpha', \alpha' \rightarrow \alpha'', \alpha'' \rightarrow \alpha''', \alpha''' \rightarrow \alpha_{out}^{iv}$, then we can return all the symbols unmodified to the skin membrane. In particular, we can simulate only one rule of the matrix, using only one rule from the pair $c_1 X \rightarrow \bar{c}_1 Y' d'_1 e, c_2 A \rightarrow \bar{c}_2 h(x) d'_2 e$. Assume that the first rule is used, the second not; the other case is symmetric. This means that in membrane i we have the symbols \bar{c}_1, c_2, d'_1, e . The symbol e can now be paired either with \bar{c}_1 or with c_2 , that is one of the rules $\bar{c}_1 e \rightarrow c_1, c_2 e \rightarrow \bar{c}_2 \dagger$ must be used. The second rule introduces the trap-symbol \dagger , hence the computation will never stop. If we use the first rule, then also the rule $c_2 d'_1 \rightarrow \bar{c}_2 \dagger$ must be used, hence again the trap-symbol \dagger is introduced. Therefore, both rules $c_1 X \rightarrow \bar{c}_1 Y' d'_1 e, c_2 A \rightarrow \bar{c}_2 h(x) d'_2 e$ must be used. The catalysts (barred or not) will remove the auxiliary symbols d_1, d_2, e and will return to their non-barred forms. This is done in four steps and exactly at the same time all the nonterminals present in the membrane will be sent to the upper membrane, in the form α^{iv} . Such symbols will be pushed up until they again reach the skin membrane.

The simulation of a terminal matrix $(X \rightarrow \lambda, A \rightarrow x)$ is done exactly in the same way.

Note that any terminal symbol is immediately sent up. In the skin mem-

brane, each terminal can wait as long as we want (by using rules $a \rightarrow a$) or it can be sent out of the system (by rules $a \rightarrow a_{out}$). In this way, all permutations of a string in $L(G)$ can be obtained.

After simulating a terminal matrix, we remove the symbol from N_1 , therefore if any symbol from N_2 is still present in the system, then the computation will never halt (such nonterminals will dissolve membranes, hence symbols f_i will be released in upper membranes). Thus, a computation in Π halts only if the corresponding derivation in G is a terminal one.

It is now clear that each derivation in G can be simulated in Π in such a way that any permutation of the terminal string generated by this derivation can be produced at the end of the computation and, conversely, all computations in Π which end in a correct way correspond to terminal derivations in G . \square

Note that during a correctly ended computation the actions δ, τ are never used; these operations are only useful for preventing “wrong” steps, which do not correspond to correct derivation steps in the grammar G .

The depth of the system in the previous proof depends on the number of matrices in the grammar we start with. Let us now look for a graph of a minimal depth. A counterpart of the previous theorem can be obtained (even using no target indication when sending symbols from the skin membrane to the lower level membranes; in this moment, this is somewhat expected, taking into account the way of keeping together all the nonterminal symbols, by using the actions δ, τ).

Theorem 3.14 *Each language $p(L), L \in CE$, can be generated by a P system of type $(2Cat_2, \delta, \tau)$, communicating by using in/out indications only, and with a membrane structure of depth two.*

Proof. The proof is similar to that of Theorem 3.13, but because of the importance of this result, for the sake of completeness, we give the core construction with almost full details.

Start again from a matrix grammar with appearance checking, $G = (N, T, S, M, F)$, in the binary normal form, with n matrices, m_1, \dots, m_n .

We construct the P system (of degree $n + 1$)

$$\Pi = (V, T, C, \mu, w_0, w_1, \dots, w_n, R_0, R_1, \dots, R_n)$$

(the skin membrane is labelled with 0) with the following components:

$$\begin{aligned} V &= N_1 \cup N_2 \cup T \cup C \cup \{d_1, d_2, d'_1, d'_2, d''_1, d''_2, e, f, \dagger\} \\ &\cup \{\alpha', \alpha'', \alpha''', \bar{\alpha} \mid \alpha \in N_1 \cup N_2\} \end{aligned}$$

$$C = \{c_1, c_2, \bar{c}_1, \bar{c}_2\},$$

$$\mu = [{}_0[{}_1]_1 \cdots [{}_n]_n]_0,$$

$$w_0 = XA, \text{ for } (S \rightarrow XA) \text{ being the initial matrix of } G,$$

$$w_i = c_1 c_2 f, 1 \leq i \leq n,$$

and with the sets R_0, R_1, \dots, R_n constructed as follows.

1. R_0 contains the following rules:

1. $\alpha \rightarrow \bar{\alpha}_{in}$, for $\alpha \in N_1 \cup N_2$
(send down nonterminals, to any membrane $1, 2, \dots, n$),
2. $a \rightarrow a$,
 $a \rightarrow a_{out}$, for $a \in T$
(send out terminals, at any time after having them in the skin membrane),
3. $f \rightarrow \dagger$,
 $\dagger \rightarrow \dagger$
(trap-rules).

2. Each set $R_i, 1 \leq i \leq n$, contains the following rules:

1. $\bar{\alpha} \rightarrow \alpha\delta$, for $\alpha \in N_2$,
 $\bar{\beta} \rightarrow \beta\tau$, for $\beta \in N_1$
(check whether or not all nonterminals are present in the same place),
2. $\dagger \rightarrow \dagger$.

3. If the matrix m_i is of the form $(X \rightarrow z, A \rightarrow x)$, for some $X \in N_1, A \in N_2, x \in (N_2 \cup T)^*$, and $z \in N_1 \cup \{\lambda\}$, then in R_i we introduce the rules of types 4.1 – 4.3 from the previous proof, as well as the following rules:

$$\begin{aligned} \alpha''' &\rightarrow \alpha_{out}, \text{ for } \alpha \in N_1 \cup N_2, \\ \bar{c}_1 e &\rightarrow c_1, \\ \bar{c}_2 e &\rightarrow c_2 \end{aligned}$$

(also the auxiliary symbol e is removed; the catalysts return to their non-barred state).

4. If the matrix m_i is of the form $(X \rightarrow Y, A \rightarrow \dagger)$, for $X, Y \in N_1, A \in N_2$ (hence with the second rule used in the appearance checking manner), then we introduce to R_i the following rules:

1. $X \rightarrow Y_{out}$,
 $Z \rightarrow \dagger$, for all $Z \in N_1 - \{X\}$,
2. $A \rightarrow \dagger$,
3. $\alpha \rightarrow \alpha_{out}$, for all $\alpha \in N_2 - \{A\}$
(all nonterminals go out, except A ; if A is present, then the computation will never finish).

In the same way as in the proof of Theorem 3.13, we have the equality $p(L(G)) = L(\Pi)$. It is clear that the membrane structure of Π is of the desired type. \square

A similar result is obtained in [237] for a class of P systems which is not known to characterize the computably enumerable sets of vectors; we give this theorem without a proof (note that in [237] one deals with target indications of the form in_j and priorities, but not with catalysts and action τ):

Theorem 3.15 *For each language $L \in LP_m(Pri, nCat, \delta)$, $m \geq 1$, we can construct a P system Π of degree m and of depth 2 such that $L(\Pi) = L$.*

3.8 P Systems on Asymmetric Graphs

The observation that a membrane structure corresponds to a tree and that we can compute directly on a tree suggests the following general idea: consider computing devices similar to P systems (we call them P' systems), using an arbitrary graph as an underlying structure. Some of the operations specific to P systems (using evolution rules in parallel, maybe subject to some priority relations, moving symbols from a node to another one, reading the result in a node or outside the graph) can be easily extended to such a general case, but others should be carefully defined. This is the case with the dissolving operation, δ , and with its dual, τ .

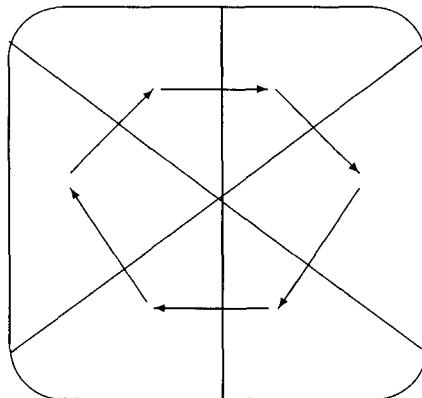


Figure 3.10: A possible support for a computation.

The idea of using graphs different from trees is not completely unrealistic from a (bio)chemical point of view. For instance, we can imagine a “membrane structure” as that in Figure 3.10, where six regions are delimited by six “walls” built in a large “pot”. A computation based on multiset processing can be done also in such a framework (described by a graph in the form of a ring).

In general, any type of a planar map (hence planar graph) can be considered as an underlying structure for a P' system, but again we look for structures which are as simple as possible. Here is an immediate question: what about of the support of the type in Figure 3.10, with a ring as a graph?

There appears here an interesting point. In trees, each edge is supposed to be used for passing objects in both directions; in most cases, a symbol which reaches a node must also leave the node. In the case of a ring graph (in general, for arbitrary graphs) we may ask whether or not we can pass symbols through walls only in one direction. This corresponds to using asymmetric graphs: for each vertices i, j , at most one of $(i, j), (j, i)$ is an edge.

In the case of a cycle-ring, the communication must go in one direction only, that is, each wall is supposed to be one-way. Thus, the only possibility to send a symbol a from a region to another one can be simply indicated by a_{go} ; as usual, a_{out} means sending a outside the system (this is necessary in order to collect the result of a computation) and a_{here} means to keep a in the same region (the subscript *here* will be systematically omitted).

We can define the actions δ, τ also for asymmetric graphs, in the following way. In general, dissolving a membrane means to move all its objects to the upper membrane and remove its rules; the membrane itself disappears. In the case of planar maps it is not clear how a room can disappear (how the neighbouring rooms must connect each other after removing the room). Thus, we keep here only the first action mentioned above: when δ is introduced by a rule in a given room and we do not introduce at the same time in that room also the symbol τ , then all the objects in that room will be communicated, randomly, to any of the neighbouring rooms to which a communication can take place (the action is equivalent to the fact that all symbols a in that room are transformed into a_{go} and they move through the walls which allow communication). If in some room we introduce both δ and τ at the same step, then no symbol leaves the room because of δ , but, if there are rules which introduce symbols a_{out} or a_{go} , then these symbols leave the room. If τ is introduced in a room without introducing at the same time δ also, then nothing happens. Thus, actions δ, τ do not act on the walls of the rooms, but on the objects in that room.

Of course, the case when arbitrarily many flip-flop catalysts are used is trivial: in view of Theorem 3.12, one room is enough. When we bound the number of catalysts, we can obtain one more representation of pCE similar to those in Theorems 3.13 and 3.14:

Theorem 3.16 *Each language in pCE can be generated by a P' system using two bi-stable catalysts, actions δ, τ , and an underlying graph in the form of a cycle-ring.*

Proof. (Sketch) We proceed as in the proof of Theorem 3.13. Starting from a matrix grammar with appearance checking in the binary normal form, with n matrices, m_1, \dots, m_n , we construct a map like that in Figure 3.10, with

a room associated with each matrix (labelled with the numbers $1, 2, \dots, n$) and a further room, labelled by 0, placed between rooms n and 1.

Each room $i = 1, 2, \dots, n$ has associated a symbol f_i , placed initially there, together with the two bi-stable catalysts c_1, c_2 . In room 0 we initially place the objects XA for ($S \rightarrow XA$) the S -matrix of G .

From room 0 we send all nonterminal symbols α of G to room 1, in the form (α, i) , where $1 \leq i \leq n$ (to this aim, we use rules $\alpha \rightarrow (\alpha, i)_{go}$). In each room i we consider rules which send the symbols (α, j) with $j > i$ to the next room $((\alpha, j) \rightarrow (\alpha, j)_{go})$. For (α, i) , we introduce the rules $(\alpha, i) \rightarrow \bar{\alpha}$. As in the proof of Theorem 3.13 we now check whether or not all nonterminals are in the same room (that is, whether or not in room 0 we have associated the same address k to all symbols (α, k)). In the opposite case, the symbol δ is introduced and the symbol f_i is sent to the next room. In each room, we provide rules $f_j \rightarrow \dagger$ for all j smaller than the label of the room.

The simulation of matrices of G are performed in the same way as in the proof of Theorem 3.13 (using the two bi-stable catalysts and the auxiliary symbols d_1, d_2, d'_1, d'_2, e).

For all symbols α^{iv} produced in each room when simulating a matrix, we introduce the symbols α_{go}^{iv} ; moreover, each room – excepting room number n – contains rules $\alpha^{iv} \rightarrow \alpha_{go}^{iv}$. In room n , these rules are replaced by $\alpha^{iv} \rightarrow \alpha_{go}$. In this way, all symbols return (at the same time) to room 0. The process can be iterated.

As usual, the terminal symbols are collected in room 0, where they can wait for any number of steps (by using rules $a \rightarrow a$) or can be sent out of the system. After using a terminal matrix, hence after removing the unique occurrence of a symbol from N_1 , if any nonterminal is still present, then the computation never stops: no further matrix can be simulated, the nonterminals have to travel through the rooms of the map until reaching a room where they entail the use of action δ and the trap-symbol \dagger will be produced.

Thus, our system can generate all permutations of all strings in the language generated by the grammar we start with. The formal details are left to the reader. \square

It is of interest to note that without using the actions δ, τ and using only two bi-stable catalysts we still can generate a large family of languages; we omit the proof, which is similar to that of the previous theorem:

Theorem 3.17 *Each language in the family pMAT can be generated by a P' system using two bi-stable catalysts, no action δ, τ , and having a cycle-ring as the underlying graph.*

3.9 P Systems with Active Membranes

In all variants of P systems considered in the previous sections, the number of membranes can only decrease during a computation, by dissolving membranes

as a result of applying evolution rules to the objects present in the system.

A natural possibility is to let the number of membranes also to increase during a computation, for instance, by division, as it is well-known in biology. Actually, the membranes from biochemistry are not at all passive, like those in the models discussed so far. For example, the passing of a chemical compound through a membrane is often done by a direct interaction with the membrane itself (with the so-called *protein channels* or *protein gates* present in the membrane); during this interaction, the chemical compound which passes through membrane can be modified, while the membrane itself can in this way be modified (at least locally).

We will here make use of these observations and we will consider P systems where the central role in the computation is played by the membranes: evolution rules are associated both with objects and membranes, while the communication through membranes is performed with the direct participation of the membranes; moreover, the membranes cannot only be dissolved, but they also can multiply by *division*. An elementary membrane can be divided by means of an interaction with an object from that membrane. As in Section 3.6, each membrane is supposed to have an “electrical polarization”, one of the three possible: *positive*, *negative*, or *neutral*. If in a membrane we have two immediately lower membranes of opposite polarizations, one *positive* and one *negative*, then that membrane can also divide in such a way that the two membranes of opposite charge are separated; all membranes of neutral charge and all objects are duplicated and a copy of each of them is introduced in each of the two new membranes. The skin is never divided.

In this way, the number of membranes can grow, even exponentially. As expected, by making use of this increased parallelism we can compute faster. We will see that this is the case, indeed: SAT can be solved in this framework in linear time (the time units are steps of a computation in a P system as sketched above, where we perform in parallel, in all membranes of the system, applications of evolution rules or division of membranes). Moreover, the model is shown to be computationally universal: any computably enumerable set of (vectors of) natural numbers can be generated by our systems.

A P system with active membranes is a construct

$$\Pi = (V, T, H, \mu, w_1, \dots, w_m, R),$$

where:

- (i) $m \geq 1$ (the initial degree of the system);
- (ii) V is an alphabet (the *total alphabet* of the system);
- (iii) $T \subseteq V$ (the *terminal alphabet*);
- (iv) H is a finite set of *labels* for membranes;

- (v) μ is a *membrane structure*, consisting of m membranes, labelled (not necessarily in a one-to-one manner) with elements of H ; all membranes in μ are supposed to be neutral;
- (vi) w_1, \dots, w_m are strings over V , describing the *multisets of objects* placed in the m regions of μ ;
- (vii) R is a finite set of *developmental rules*, of the following forms:

- (a) $[_h a \rightarrow v]_h^\alpha$,
for $h \in H, \alpha \in \{+, -, 0\}, a \in V, v \in V^*$
(object evolution rules, associated with membranes and depending on the label and the charge of the membranes, but not directly implying the membranes, in the sense that the membranes are neither taking part to the application of these rules nor are they modified by them);
- (b) $a[_h]_h^{\alpha_1} \rightarrow [_h b]_h^{\alpha_2}$,
for $h \in H, \alpha_1, \alpha_2 \in \{+, -, 0\}, a, b \in V$
(communication rules; an object is introduced in the membrane, maybe modified during this process; also the polarization of the membrane can be modified, but not its label);
- (c) $[_h a]_h^{\alpha_1} \rightarrow [_h]_h^{\alpha_2} b$,
for $h \in H, \alpha_1, \alpha_2 \in \{+, -, 0\}, a, b \in V$
(communication rules; an object is sent out of the membrane, maybe modified during this process; also the polarization of the membrane can be modified, but not its label);
- (d) $[_h a]_h^\alpha \rightarrow b$,
for $h \in H, \alpha \in \{+, -, 0\}, a, b \in V$
(dissolving rules; in reaction with an object, a membrane can be dissolved, while the object specified in the rule can be modified);
- (e) $[_h a]_h^{\alpha_1} \rightarrow [_h b]_h^{\alpha_2} [_h c]_h^{\alpha_3}$,
for $h \in H, \alpha_1, \alpha_2, \alpha_3 \in \{+, -, 0\}, a, b, c \in V$
(division rules for elementary membranes; in reaction with an object, the membrane is divided into two membranes with the same label, maybe of different polarizations; the object specified in the rule is replaced in the two new membranes by possibly new objects; all other objects are reproduced in both the two new membranes);
- (f)
$$\begin{aligned} &[_{h_0}[_{h_1}]_{h_1}^{\alpha_1} \cdots [_{h_k}]_{h_k}^{\alpha_1} [_{h_{k+1}}]_{h_{k+1}}^{\alpha_2} \cdots [_{h_n}]_{h_n}^{\alpha_2}]_{h_0}^{\alpha_0} \\ &\quad \rightarrow [_{h_0}[_{h_1}]_{h_1}^{\alpha_3} \cdots [_{h_k}]_{h_k}^{\alpha_3} [_{h_0}[_{h_{k+1}}]_{h_{k+1}}^{\alpha_4} \cdots [_{h_n}]_{h_n}^{\alpha_4}]_{h_0}^{\alpha_6}], \end{aligned}$$

for $k \geq 1, n > k, h_i \in H, 0 \leq i \leq n$, and $\alpha_0, \dots, \alpha_6 \in \{+, -, 0\}$ with $\{\alpha_1, \alpha_2\} = \{+, -\}$; if this membrane with the label h_0 contains other membranes than those with the labels h_1, \dots, h_n specified above, then they should have neutral charge in order to apply this rule

(division of non-elementary membranes; this is possible only if a membrane contains two immediately lower membranes of opposite polarization, + and −; the membranes of opposite polarizations are separated in the two new membranes, but their polarization can change; always, all membranes of opposite polarizations are separated by applying this rule; all objects and all other membranes from membrane h_0 are reproduced in both the two new membranes with label h_0).

Note that in all rules of types (a)–(e) only one object is specified (that is, the objects do not directly interact) and that, with the exception of rules of type (a), always single objects are transformed into single objects (the two objects produced by a division rule of type (e) are placed in two different regions).

These rules are applied according to the following *principles*:

1. All the rules are applied in parallel: in a step, the rules of type (a) are applied to all objects to which they can be applied, all other rules are applied to all membranes to which they can be applied; an object can be used by only one rule, non-deterministically chosen (there is no priority relation among rules), but any object which can evolve by a rule of any form, should evolve.
2. If a membrane is dissolved, then all the objects in its region are left free in the region immediately above it. Because all rules are associated with membranes, the rules of a dissolved membrane are no longer available at the next steps. The skin membrane is never dissolved.
3. All objects and membranes not specified in a rule and which do not evolve are passed unchanged to the next step. For instance, if a membrane with the label h is divided by a rule of type (e) which involves an object a , then all other objects in membrane h which do not evolve are introduced in each of the two resulting membranes h . Similarly, when dividing a membrane h by means of a rule of type (f), the neutral membranes are reproduced in each of the two new membranes with the label h , unchanged if no rule is applied to them (in particular, the contents of these neutral membranes are reproduced unchanged in these copies, providing that no rule is applied to their objects).
4. If at the same time a membrane h is divided by a rule of type (e) and there are objects in this membrane which evolve by means of rules of type (a), then in the new copies of the membrane we introduce the result of the evolution; that is, we may suppose that first the evolution rules of type (a) are used, changing the objects, and then the division is produced, so that in the two new membranes with label h we introduce copies of the changed objects. Of course, this process takes only one step. The same assertions apply to the division by means of a rule of

type (f): always we assume that the rules are applied “from bottom-up”, in one step, but first the rules of the innermost region and then level by level until the region of the skin membrane.

5. The rules associated with a membrane h are used for all copies of this membrane, irrespective whether or not the membrane is an initial one or it is obtained by division. At one step, a membrane h can be the subject of only one rule of types (b)–(f).
6. The skin membrane can never divide. As any other membrane, the skin membrane can be “electrically charged”.

We can pass from a configuration to another one by using the rules from R according to the principles given above. We say that we have a (direct) *transition* among configurations. We do not define formally a transition. We will immediately consider an example which can enlighten the idea.

As usual, a computation is *complete* if it cannot be continued: there is no rule which can be applied to objects and membranes in the last configuration.

Note that during a computation the number of membranes (hence the degree of the system) can increase and decrease but the labels of these membranes are always among the labels of membranes present in the initial configuration (by division we only produce membranes with the same label as the label of the divided membrane).

During a computation, objects can leave the skin membrane (by means of rules of type (c)). By arranging these symbols in a string, a language is associated with Π , denoted by $L(\Pi)$.

In order to prove the usefulness of using active membranes (in particular, membrane division) and in order to illuminate the informal definition of a transition in a P system as given above, we will consider an example which is also very significant by itself: solving the SAT problem by a P system with active membranes.

Theorem 3.18 *The SAT problem can be solved by a P system with active membranes in a time which is linear in the number of variables and the number of clauses.*

Proof. Let us consider a propositional formula

$$\gamma = C_1 \wedge C_2 \wedge \dots \wedge C_m,$$

with

$$C_i = y_{i,1} \vee \dots \vee y_{i,p_i},$$

for some $m \geq 1$, $p_i \geq 1$, and $y_{i,j} \in \{x_k, \sim x_k \mid 1 \leq k \leq n\}$, for each $1 \leq i \leq m$, $1 \leq j \leq p_i$.

We construct the P system

$$\Pi = (V, T, H, \mu, w_0, w_1, \dots, w_m, w_{m+1}, R)$$

with the components

$$\begin{aligned}
 V &= \{a_i, t_i, f_i \mid 1 \leq i \leq n\} \\
 &\cup \{c_i \mid 0 \leq i \leq 2n + m - 1\} \\
 &\cup \{t\}, \\
 T &= \{t\}, \\
 H &= \{0, 1, \dots, m + 1\}, \\
 \mu &= [_{m+1}[_m[_{m-1} \cdots [_1[_0]_0^0]_1^0 \cdots]_{m-1}^0]_m^0]_{m+1}^0, \\
 w_0 &= c_0 a_1 a_2 \dots a_n, \\
 w_i &= \lambda, \text{ for all } i = 1, 2, \dots, m + 1,
 \end{aligned}$$

while the set R contains the following rules:

$$1. [{}_0 c_i \rightarrow c_{i+1}]_0^\alpha, \text{ for all } 0 \leq i \leq 2n + m - 2 \text{ and } \alpha \in \{+, -, 0\}$$

(we count to $2n + m - 1$, which is the time needed for producing all 2^n truth assignments for the n variables, as well as 2^n membrane sub-structures which will examine the truth value of formula γ for each of these truth assignments; this counting is done in the central membrane, irrespective which is its polarity);

$$2. [{}_0 a_i]_0^0 \rightarrow [{}_0 t_i]_0^+ [{}_0 f_i]_0^-, \text{ for all } 1 \leq i \leq n$$

(in membrane 0, when it is “electrically neutral”, we non-deterministically choose one variable x_i and both values *true* and *false* are associated with it, in the form of objects t_i, f_i , which are separated in two membranes with the label 0 which differ only by these objects t_i, f_i and by their charge);

$$3. [{}_0 c_{2n+m-1}]_0^0 \rightarrow t$$

(after $2n + m - 1$ steps, each copy of membrane 0 is dissolved and its contents are released in the upper membranes, those labelled with 1);

$$4. [{}_j t_i]_j^0 \rightarrow t_i, \text{ if } x_i \text{ appears in clause } C_j, 1 \leq i \leq n, 1 \leq j \leq m, \text{ and}$$

$$[{}_j f_i]_j^0 \rightarrow f_i, \text{ if } \sim x_i \text{ appears in clause } C_j, 1 \leq i \leq n, 1 \leq j \leq m$$

(a membrane with label j , $1 \leq j \leq m$, is dissolved if and only if clause C_j is satisfied by the current truth assignment; if this is the case, then the truth values associated with the variables are released in the upper membrane, that associated with the next clause, C_{j+1} , otherwise these truth values remain blocked in membrane j and never used at the next steps by the membranes placed above; note that, as

we will see immediately, after $2n + m - 1$ steps we have 2^n membrane sub-structures of the form $[_m[_{m-1} \dots [1]_1^0 \dots]_{m-1}^0]_m^0$ working in parallel in the skin membrane);

$$5. [_{m+1}t]_{m+1}^0 \rightarrow [_{m+1}]_{m+1}^+ t$$

(together with the truth assignments, we also have the object t , which can be passed from a level to the upper one only by dissolving membranes; this object reaches the skin membrane if and only if all membranes in a sub-structure of the form $[_m[_{m-1} \dots [1]_1^0 \dots]_{m-1}^0]_m^0$ are dissolved, which means that the associated truth assignment has satisfied all the clauses, that is, the formula is satisfiable; therefore, t leaves the system if and only if the formula is satisfiable; when this rule is applied, the skin membrane gets a “positive charge”, so the rule can be applied only once);

$$6. [_{i+1}[_i]_i^+ [_i]_i^-]_{i+1}^0 \rightarrow [_{i+1}[_i]_i^0]_{i+1}^+ [_{i+1}[_i]_i^0]_{i+1}^-, \text{ for all } 0 \leq i \leq m-2, \text{ and}$$

$$[_m[_{m-1}]_{m-1}^+ [_{m-1}]_{m-1}^-]_m^0 \rightarrow [{}_m[_{m-1}]_{m-1}^0]_m^0 [{}_m[_{m-1}]_{m-1}^0]_m^0$$

(division rules for membranes labelled with $0, 1, \dots, m$; the opposite polarization introduced when dividing a membrane 0 is propagated from lower levels to upper levels of the membrane structure and the membranes are continuously divided until dividing also membrane m – which will get a neutral charge).

From the previous explanations one can easily see that

$$L(\Pi) = \begin{cases} \{t\}, & \text{if formula } \gamma \text{ is satisfiable,} \\ \emptyset, & \text{otherwise.} \end{cases}$$

Therefore, we get the answer to the question whether or not γ is satisfiable by examining the emptiness of the language $L(\Pi)$ (by checking whether or not any object leaves the system Π during the computation). This is achieved in $2n + 2m + 1$ steps: in $2n + m - 1$ steps we create the 2^n membrane sub-structures of the form $[_m[_{m-1} \dots [1]_1^0 \dots]_{m-1}^0]_m^0$ (as well as the 2^n different truth assignments), then we dissolve all membranes 0 (one further step) and we check the satisfiability of each clause for each truth assignment, in parallel in the 2^n sub-structures (this takes further m steps); one more step is necessary in order to send out of the system one copy of the object t , if any copy of t has reached the skin membrane. If no copy of t leaves the system at this step, then the formula is not satisfiable. \square

Note that we have used rules of all types (a)–(f), excepting the type (b), and that also in the case of rules of type (a) we have object-to-object rules (and not object-to-multiset).

We illustrate the previous construction and the work of the system II obtained in this way for the propositional formula

$$\beta = (x_1 \vee x_2) \wedge (\sim x_1 \vee \sim x_2),$$

also considered in Section 2.5.

Thus, $n = 2, m = 2$. The initial configuration of the system is

$$[3[2[1[_0c_0a_1a_2]_0^0]_1^0]_2^0]_3^0.$$

The computation proceeds as follows (we specify the current configuration at each step):

$$\text{Step 0: } [3[2[1[_0c_0a_1a_2]_0^0]_1^0]_2^0]_3^0;$$

$$\text{Step 1: } [3[2[1[_0c_1t_1a_2]_0^+[_0c_1f_1a_2]_0^-]_1^0]_2^0]_3^0$$

(in parallel, the rule $[_0c_0 \rightarrow c_1]_0^0$ and the division rule $[_0a_1]_0^0 \rightarrow [_0t_1]_0^+[_0f_1]_0^-$ were used; membrane 1 must immediately divide, because of the two copies of membrane 0 with opposite polarizations);

$$\text{Step 2: } [3[2[1[_0c_2t_1a_2]_0^0]_1^+[_1[_0c_2f_1a_2]_0^0]_1^-]_2^0]_3^0$$

(the counter c_1 is replaced with c_2 and membrane 1 is divided; because the two membranes with label 0 are not of neutral polarity, no new truth value is introduced at this step; at the next step, membrane 2 must divide);

$$\text{Step 3: } [3[2[1[_0c_3t_1t_2]_0^+[_0c_3t_1f_2]_0^-]_1^0]_2^0[2[1[_0c_3f_1t_2]_0^+[_0c_3f_1f_2]_0^-]_1^0]_2^0]_3^0$$

(simultaneously, the counter c_2 is replaced by c_3 , each membrane 0 is divided again, producing membranes of opposite polarity, and membrane 2 is also divided, because of the existence of the two membranes 1 with opposite polarity; the generation of the truth assignments is completed, but we still have to divide membranes, because of the existence of membranes of opposite polarity);

Step 4:

$$[3[2[1[_0c_4t_1t_2]_0^0]_1^+[_1[_0c_4t_1f_2]_0^0]_1^-]_2^0[2[1[_0c_4f_1t_2]_0^0]_1^+[_1[_0c_4f_1f_2]_0^0]_1^-]_2^0]_3^0$$

(we divide the two membranes 1, in parallel, and we increase the counter; no other rule can be applied);

$$\text{Step 5: } [3[2[1[_0c_5t_1t_2]_0^0]_1^0]_2^0[2[1[_0c_5t_1f_2]_0^0]_1^0]_2^0[2[1[_0c_5f_1t_2]_0^0]_1^0]_2^0 \\ [2[1[_0c_5f_1f_2]_0^0]_1^0]_2^0]_3^0$$

(we increase again the counter and we divide the two membranes 2).

The membrane structure (and the contents of membranes with label 0) is represented in Figure 3.11. One sees that for each truth assignment we have a sub-structure of the form $[_2[_1[0]_0]_1]_2$. All membranes have neutral polarity.

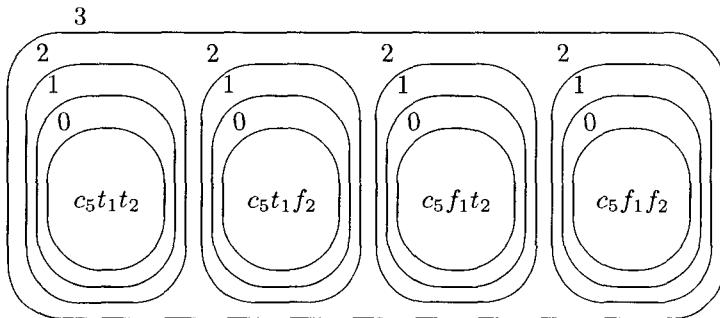


Figure 3.11: The membrane structure in the example, after Step 5.

$$\text{Step 6: } [_3[_2[_1tt1t2]_1]_2^0[_2[_1tt1f2]_1]_2^0[_2[_1tf1t2]_1]_2^0[_2[_1tf1f2]_1]_2^0]_3^0$$

(the counter has reached its maximal value; it can be transformed in t while dissolving all membranes 0);

$$\text{Step 7: } [_3[_2tt1t2]_2^0[_2tt1f2]_2^0[_2tf1t2]_2^0[_2tf1f2]_1]_2^0]_3^0$$

(clause 1 is satisfied by the first three truth assignments; the corresponding membranes with label 1 are dissolved; the last truth assignment does not satisfy the first clause, its associated membrane 1 remains unchanged and the truth values in it will be of no use from now on);

$$\text{Step 8: } [_3[_2tt1t2]_2^0tt1f2tf1t2[_2[_1tf1f2]_1]_2^0]_3^0$$

(clause 2 is satisfied by the second and the third truth assignments, so the corresponding membranes 2 are dissolved; two copies of t are left free in the skin membrane, corresponding to the two truth assignments which satisfy the formula);

Step 9: One of the two copies of t will be sent out of the system, so we know that the formula was satisfied. This is the last step of the computation, because no further rule can be applied (the skin membrane is now “positively charged”, so the second copy of t cannot leave it).

It is of a clear interest to find also other NP-complete problems which can be solved (directly, not via SAT) by means of P systems with active membranes. We leave this *research topic* to the reader and we pass to investigate the computability power of P systems with active membranes.

We denote by LPA the family of languages $L(\Pi)$, generated by P systems with active membranes as (informally) defined above.

Theorem 3.19 $PsCE = PsLPA$.

Proof. The inclusion $LPA \subseteq CE$ follows from Church–Turing thesis or can be proved directly, in a straightforward (but involving a long construction) way. This implies the inclusion $PsLPA \subseteq PsCE$. So, we only have to prove the inclusion $PsCE \subseteq PsLPA$, which is done in the following lemma. \square

Lemma 3.7 (The Computational Completeness Lemma for P Systems with Active Membranes) $PsCE \subseteq PsLPA$.

Proof. We make use of the equality $PsCE = PsMAT_{ac}$. Let $G = (N, T, S, M, F)$ be a matrix grammar with appearance checking in the binary normal form, with $N = N_1 \cup N_2 \cup \{S, \dagger\}$. Each matrix of the form $(X \rightarrow \lambda, A \rightarrow x)$, $X \in N_1, A \in N_2, x \in T^*$, is replaced by $(X \rightarrow Z, A \rightarrow x)$, where Z is a new symbol. We denote by G' the obtained grammar. Assume that we have n_1 matrices of the form $(X \rightarrow Y, A \rightarrow x)$, with $X \in N_1, Y \in N_1 \cup \{Z\}, x \in (N_2 \cup T)^*$, and n_2 matrices of the form $(X \rightarrow Y, A \rightarrow \dagger)$, $X, Y \in N_1, A \in N_2$. (That is, we consider separately the matrices having rules used in the appearance checking mode and the matrices not having such rules.)

We construct the P system (with $p = n_1 + n_2 + 2$ initial membranes)

$$\Pi = (V, T, H, \mu, w_1, \dots, w_p, R),$$

with

$$\begin{aligned} V &= N_1 \cup N_2 \cup T \cup \{Z, \dagger\} \\ &\cup \{\bar{X} \mid X \in N_1\} \\ &\cup \{\langle x \rangle \mid x \in (N_2 \cup T)^*, (X \rightarrow Y, A \rightarrow x) \text{ is a matrix in } G'\}, \\ H &= \{1, 2, \dots, p\}, \\ \mu &= [p [p-1 [1]_1^0 [2]_2^0 \cdots [n_1]_{n_1}^0 [n_1+1]_{n_1+1}^0 \cdots [n_1+n_2]_{n_1+n_2}^0]_{p-1}]_p^0, \\ w_i &= \lambda, \text{ for all } i \in H - \{p-1\}, \\ w_{p-1} &= XA, \text{ for } (S \rightarrow XA) \text{ the initial matrix of } G, \end{aligned}$$

and the following set R of rules:

1. For each matrix $m_i = (X \rightarrow Y, A \rightarrow x)$, $1 \leq i \leq n_1$, we introduce the rules:

$$\begin{aligned} X[i]_i^0 &\rightarrow [iY]_i^+, \\ A[i]_i^+ &\rightarrow [iA]_i^-, \\ [iA]_i^- &\rightarrow [i]_i^0 \langle x \rangle, \\ [iY]_i^0 &\rightarrow [i]_i^0 Y, \end{aligned}$$

as well as the rules

$$[p-1]\langle x \rangle \rightarrow x[p-1]^0.$$

2. For each matrix $m_i = (X \rightarrow Y, A \rightarrow \dagger)$, $n_1 + 1 \leq i \leq n_1 + n_2$, we introduce the rules:

$$\begin{aligned} X[i]_i^0 &\rightarrow [iX]_i^0, \\ [iX]_i^0 &\rightarrow [i\bar{X}]_i^+ [iY]_i^-, \\ [p-1][i]_i^+ [i]_i^- [p-1]^0 &\rightarrow [p-1][i]_i^+ [p-1][i]_i^- [p-1]^0, \\ A[i]_i^+ &\rightarrow [i\dagger]_i^0, \\ [iY]_i^- &\rightarrow [i]_i^0 Y, \\ [i\dagger \rightarrow \dagger]_i^0. \end{aligned}$$

3. We also consider the following rules, for all $a \in T$,

$$\begin{aligned} [p-1]_p^0 a &\rightarrow [p-1]_{p-1}^0 a, \\ [p]_p^0 a &\rightarrow [p]_p^0 a, \end{aligned}$$

as well as the following rules for all $\alpha \in N_1 \cup N_2$

$$[p-1]\alpha \rightarrow \alpha[p-1]^0.$$

The system works as follows.

Membranes labelled with $1, 2, \dots, n_1$ are associated with matrices not used in the appearance checking mode; each matrix is simulated with the help of the associated membrane. In any moment, in membrane $p-1$ there is only one symbol from the set N_1 . If this symbol enters a membrane with the label i , $1 \leq i \leq n_1$, then the membrane gets the “electrical charge” + (initially, all membranes are neutral). We can continue only by introducing in this membrane i also the corresponding symbol A (at that time, the membrane gets a negative “electrical charge”). The continuation is deterministic: we send out of membrane i the symbol $\langle x \rangle$ (and the membrane is again neutral), then we send out also the symbol Y ; at the second step, the symbol $\langle x \rangle$ is replaced in membrane $p-1$ by the string x . In this way, in membrane $p-1$ we have simulated the use of the matrix $m_i = (X \rightarrow Y, A \rightarrow x)$. Note how the “electrical charge” of the membrane controls the process and that the symbol Y is available in membrane $p-1$ only after completing the simulation of the matrix.

Membranes with labels $n_1 + 1, \dots, n_1 + n_2$ are associated with matrices used in the appearance checking mode. Let $i, n_1 + 1 \leq i \leq n_1 + n_2$, be

such a membrane, associated with $m_i = (X \rightarrow Y, A \rightarrow \dagger)$. As above, the symbol X can enter membrane i (unchanged); having this symbol inside, this membrane is divided in two membranes, of opposite polarity. In the first membrane, that with positive “charge”, we check whether or not any occurrence of A is present in the membrane $p - 1$. This is done as follows. Because of this opposite polarity of membranes with label i , membrane $p - 1$ is also divided, in a copy of positive “charge” and a neutral copy. In this way, all objects from the former membrane with the label $p - 1$ are duplicated and introduced in each of these membranes. If the symbol A is present, then at the same step when membrane $p - 1$ is divided, we introduce the trap-object \dagger in the positively “charged” copy of membrane $p - 1$. This object will evolve forever, so the computation will never finish. Also in parallel with the division of membrane $p - 1$, we release the symbol Y in the copy of membrane $p - 1$ which is neutral. Thus, the simulation of the matrix m_i is successful if and only if the computation will ever stop, that is, if and only if the symbol A was not present.

The process can continue. Each terminal symbol present in the membrane with the label $p - 1$ and of neutral polarity is sent to the skin membrane and from here it is sent out of the system. As long as any nonterminal symbol from $N_1 \cup N_2$ is present in the membrane with the label $p - 1$ and of neutral polarity, the computation is not halting. (Note that the copies of membrane $p - 1$ produced for simulating matrices in the appearance checking mode and used only for checking the non-appearance of symbols $A \in N_2$ have a positive charge, so the nonterminal symbols present in them do not evolve.) Consequently, we simulate in Π exactly the terminal derivations in G' . Because the symbol Z cannot evolve, we have the equality $\Psi_T(L(G)) = \Psi_T(L(\Pi))$. \square

In the construction above we have used rules of all types (a)–(f), with the exception of dissolving rules of type (d).

Because the division of membranes is used only for simulating matrices which contain rules used in the appearance checking manner, from the previous construction we obtain the fact that the Parikh sets of languages in MAT can be obtained as the Parikh sets of languages generated by P systems which do not use membrane division. We denote by $LPA(ndiv)$ this family of languages, hence we can write:

Corollary 3.2 $PsMAT \subseteq PsLPA(ndiv)$.

Actually, this is a proper inclusion, because of the following result:

Theorem 3.20 $LPA(ndiv) - MAT \neq \emptyset$.

Proof. We consider the P system

$$\Pi = (\{a, b\}, \{a\}, \{1\}, [1]_1^0, ab, R),$$

$$R = \{[1]_1 a \rightarrow aa [1]_1^0, [1]_1 b \rightarrow b [1]_1^0, [1]_1 b [1]_1^0 \rightarrow [1]_1 b [1]_1^+, [1]_1 a [1]_1^+ \rightarrow [1]_1 [1]_1^+ a\}.$$

As long as membrane 1 is neutral, the number of occurrences of object a is doubled. At the same time, the object b is “doing nothing”. At any moment, object b can determine the change of the polarity of membrane 1 (it becomes positive). From now on, no other rule can be used than those which send out of the system all available copies of object a . Consequently, we have $L(\Pi) = \{a^{2^n} \mid n \geq 1\}$. This is not a language in the family MAT (it is a non-regular one-letter language). \square

3.10 Splicing P Systems

We now relate the idea of computing with membranes to the splicing operation. Specifically, we consider P systems with objects in the form of strings (as also considered in Section 3.5) and with the evolution rules based on the splicing operation (as investigated in Chapter 2).

A *splicing P system* (of degree $m, m \geq 1$) is a construct

$$\Pi = (V, T, \mu, L_1, \dots, L_m, R_1, \dots, R_m),$$

where:

- (i) V is an alphabet;
- (ii) $T \subseteq V$ (the *output alphabet*);
- (iv) μ is a membrane structure consisting of m membranes (labeled with $1, 2, \dots, m$);
- (v) $L_i, 1 \leq i \leq m$, are languages over V associated with the regions $1, 2, \dots, m$ of μ ;
- (vi) $R_i, 1 \leq i \leq m$, are finite sets of *evolution rules* associated with the regions $1, 2, \dots, m$ of μ , given in the following form: $(r; tar_1, tar_2)$, where $r = u_1 \# u_2 \$ u_3 \# u_4$ is a usual splicing rule over V and $tar_1, tar_2 \in \{\text{here}, \text{out}\} \cup \{in_j \mid 1 \leq j \leq m\}$.

Note that, as usual in H systems, when a string is present in a region of our system, it is assumed to appear in arbitrarily many copies (any number of copies of a DNA molecule can be obtained by amplification). Thus, we do not use here multisets, as in basic P systems.

Any m -tuple (M_1, \dots, M_m) of languages over V is called a *configuration* of Π . For two configurations $(M_1, \dots, M_m), (M'_1, \dots, M'_m)$ of Π we write $(M_1, \dots, M_m) \Rightarrow (M'_1, \dots, M'_m)$ if we can pass from (M_1, \dots, M_m) to (M'_1, \dots, M'_m) by applying the splicing rules from each region of μ , in parallel, to all possible strings from the corresponding regions, and following the target indications associated with the rules. More specifically (but not completely formal), if $x, y \in M_i$ and $(r = u_1 \# u_2 \$ u_3 \# u_4; tar_1, tar_2) \in R_i$

such that we can have $(x, y) \vdash_r (w, z)$, then w and z will go to the regions indicated by $\text{tar}_1, \text{tar}_2$, respectively. If $\text{tar}_j = \text{here}$, then the string remains in M_i , if $\text{tar}_j = \text{out}$, then the string is moved to the region immediately outside membrane i (maybe, in this way the string leaves the system), if $\text{tar}_j = \text{in}_k$, then the string is moved to the region k , providing that this is immediately below; if not, then the rule cannot be applied. Note that the strings x, y are still available in region M_i , because we have supposed that they appear in arbitrarily many copies (an arbitrarily large number of them were spliced, arbitrarily many remain), but if a string w, z is sent out of region i , then no copy of it remains here.

A sequence of transitions between configurations of a given P system Π , starting from the initial configuration (L_1, \dots, L_m) , is called a *computation* with respect to Π . The result of a computation consists of all strings over T which are sent out of the system at any time during the computation. We denote by $L(\Pi)$ the language of all strings of this type. We say that $L(\Pi)$ is *generated* by Π .

Note that in this section we do not consider halting computations. We leave the process to continue forever and we just observe it from outside and collect the terminal strings leaving it.

We denote by $SPL(\text{tar}, m, p)$ the family of languages $L(\Pi)$ generated by splicing P systems as above, of degree at most $m, m \geq 1$, and of depth at most $p, p \geq 1$.

If all target indications $\text{tar}_1, \text{tar}_2$ in the evolution rules of a P system are of the form *here*, *out*, *in*, then we say that Π is of the *i/o type*; the strings produced by splicing and having associated the indication *in* are moved into any lower region immediately below the region where the rule is used. The family of languages generated by P systems with this weaker target indication and of degree at most m and depth at most p is denoted by $SPL(i/o, m, p)$.

We start by four results showing the computational universality of splicing P systems of rather simple forms.

Theorem 3.21 $SPL(i/o, 3, 3) = CE$.

Proof. Let $G = (N, T, S, P)$ be a type-0 Chomsky grammar and let B be a new symbol. Assume that $N \cup T \cup \{B\} = \{\alpha_1, \dots, \alpha_n\}$ and that P contains m rules, $u_i \rightarrow v_i, 1 \leq i \leq m$. Consider also the rules $u_{m+j} \rightarrow v_{m+j}, 1 \leq j \leq n$, for $u_{m+j} = v_{m+j} = \alpha_j$.

We construct the splicing P system (of degree 3)

$$\Pi = (V, T, \mu, L_1, L_2, L_3, R_1, R_2, R_3),$$

$$V = N \cup T \cup \{B, X, X', Y, Z, Z'\}$$

$$\cup \{X_i \mid 1 \leq i \leq n+m\}$$

$$\cup \{Y_i \mid 0 \leq i \leq n+m\},$$

$$\mu = [1[2[3]_3]_2]_1,$$

$$\begin{aligned}
L_1 &= \{XSBY, Z'\} \cup \{ZY_i \mid 0 \leq i \leq n+m\}, \\
L_2 &= \{X'Z\} \cup \{X_iv_iZ, X_iZ \mid 1 \leq i \leq n+m\}, \\
L_3 &= \{ZY\}, \\
R_1 &= \{(\#u_iY\$Z\#Y_i; in, here), (\#Y_i\$Z\#Y_{i-1}; in, here) \mid 1 \leq i \leq n+m\} \\
&\quad \cup \{(\#BY\$Z'\#; here, here), (X\#\$\#Z'; here, out)\}, \\
R_2 &= \{(X\#\$X_iv_i\#Z; here, out) \mid 1 \leq i \leq n+m\} \\
&\quad \cup \{(X_i\#\$X_{i-1}\#Z; here, out) \mid 2 \leq i \leq n+m\} \\
&\quad \cup \{(X_1\#\$X'\#Z; here, in), (X'\#\$X\#Z; here, out)\}, \\
R_3 &= \{(\#Y_0\$Z\#Y; out, here)\}.
\end{aligned}$$

The idea of the proof is again the “rotate-and-simulate” technique, as used in the proof of Lemma 2.3.

The sentential forms generated by G are simulated in Π in a circular permutation: Xw_1Bw_2Y , maybe with variants of X, Y , will be present in a region of Π if and only if w_2w_1 is a sentential form of G . Note that we can remove the nonterminal symbol Y only together with B from strings of the form $XwBY$. In this way, we ensure that the string is in the correct permutation.

The simulation of rules in P and the rotation are done in the same way. Assume that some string Xwu_iY is present in region 1, $1 \leq i \leq n+m$; initially we have here the string $XSBY$. We can perform $(Xw|u_iY, Z|Y_i) \vdash (XwY_i, Zu_iY)$; the string XwY_i is sent to region 2, Zu_iY remains in region 1 (and all possible splicings which involve it produce two strings starting with Z). In region 2 we can only perform a splicing of the form $(X|wY_i, X_jv_j|Z) \vdash (XZ, X_jv_jwY_i)$, for some $1 \leq j \leq n+m$. The string $X_jv_jwY_i$ is sent back to region 1, XZ remain here and will be used later. Now, in region 1 the only splicing which can involve the string $X_jv_jwY_i$ is $(X_jv_jw|Y_i, Z|Y_{i-1}) \vdash (X_jv_jwY_{i-1}, ZY_i)$. The string $X_jv_jwY_{i-1}$ is sent to region 2, ZY_{i-1} remains in region 1 (it is an axiom). In region 2 we now decrease by one the subscript of X_j , by $(X_j|v_jwY_{i-1}, X_{j-1}|Z) \vdash (X_jZ, X_{j-1}v_jwY_{i-1})$. The process of decreasing the subscripts continues.

If at some moment we reach X_1 , hence in region 2 we have a string $X_1v_jwY_k$, then we perform $(X_1|v_jwY_k, X'|Z) \vdash (X_1Z, X'v_jwY_k)$ and $X'v_jwY_k$ is sent to membrane 3. If $k \neq 0$, then nothing can be done, the string is “lost”. Otherwise, Y_0 is replaced with Y and the string $X'v_jwY$ is sent to region 2; X' is replaced here by X and the string Xv_jwY is sent to the skin membrane.

If at some moment in region 2 we get a string $X_kv_jwY_0$, for $k \geq 2$, then this string cannot be processed in the skin membrane, hence it is “lost”.

Thus, we can correctly continue only when $i = j$, hence we have passed from Xwu_iY to Xv_iwY ; in this way we have either correctly simulated a rule from P or we have circularly permuted the string with one symbol.

The process of simulating a rule or of rotating the string with one symbol

can be iterated. Therefore, all derivations in G can be simulated in Π and, conversely, all correct computations in Π correspond to correct derivations in G . Because we collect only terminal strings which leave the system, we have the equality $L(G) = L(\Pi)$. \square

In order to minimize the depth of the used system we have to pay some price. In the next theorem, this price is the use of target information; the proof is similar to the previous one, but slightly simpler, because we can make use of the powerful indications of the form in_j :

Theorem 3.22 $SPL(tar, 3, 2) = CE$.

However, at the price of using two more membranes, we can get rid of the target addresses, still using a system of depth 2:

Theorem 3.23 $SPL(i/o, 5, 2) = CE$.

Proof. Starting from a type-0 grammar $G = (N, T, S, P)$ as above and with B a new symbol, we construct the system (of degree 5)

$$\begin{aligned} \Pi &= (V, T, \mu, L_1, L_2, L_3, L_4, L_5, R_1, R_2, R_3, R_4, R_5), \\ V &= N \cup T \cup \{B, X, X', Y, Y', Y'', Z, Z'\} \\ &\quad \cup \{X_i \mid 1 \leq i \leq n+m\} \\ &\quad \cup \{Y_i \mid 0 \leq i \leq n+m\}, \\ \mu &= [1[2]2[3]3[4]4[5]5]_1, \\ L_1 &= \{XSBY, ZY'', Z'\} \cup \{ZY_i \mid 0 \leq i \leq n+m\}, \\ L_2 &= \{X_i v_i Z \mid 1 \leq i \leq n+m\}, \\ L_3 &= \{X_i Z \mid 1 \leq i \leq n+m\}, \\ L_4 &= \{XZ, ZY\}, \\ L_5 &= \{X'Z, ZY'\}, \\ R_1 &= \{(\#u_i Y \$ Z \# Y_i; in, here), (\#Y_i \$ Z \# Y_{i-1}; in, here) \mid 1 \leq i \leq n+m\} \\ &\quad \cup \{(\#Y' \$ Z \# Y''; in, here), (\#BY \$ Z' \#; here, here), \\ &\quad (X \$ \# Z'; here, out)\}, \\ R_2 &= \{(X \$ X_i v_i \$ Z; here, out) \mid 1 \leq i \leq n+m\}, \\ R_3 &= \{(X_i \$ X_{i-1} \$ Z; here, out) \mid 2 \leq i \leq n+m\}, \\ R_4 &= \{(X' \$ X \$ Z; here, out), (\#Y'' \$ Z \# Y; here, here)\}, \\ R_5 &= \{(X_1 \$ X' \$ Z; here, here), (\#Y_0 \$ Z \# Y'; out, here)\}. \end{aligned}$$

This system works in a way similar to the system in the proof of Theorem 3.21, but the fact that we can continue the simulation of rules in P or the circular permutation of strings only in the correct manner is much more subtle.

In region 1 we start to simulate the application of rules in P and the circular permutation of a string, by cutting from the right hand end of the string a substring $u_i, 1 \leq i \leq n + m$. In membrane 2 we add to the left end of the string a prefix v_j . These operations are “memorized” also in the subscripts of the end markers X and Y . The subscript of X is decreased in membrane 3 while the subscript of Y is decreased in membrane 1. Only when the two subscripts reach the value 0 at the same time the process has to correctly continue. This happens, indeed.

In order to see this, let us consider the two possible cases when in region 1 we get a string with one of the subscripts of X and Y of a minimal value (this string should be sent to an inner membrane). When the subscript of X is minimal, our string is of the form X_1wY_k , with $k \geq 1$. When the subscript of Y is minimal, the string is of the form X_kwY_0 , for some $k \geq 1$. Only the case X_1wY_0 should continue correctly. We consider in detail each case:

1. The string $X_1wY_k, k \geq 1$, can enter no splicing in membranes 2, 3, 4, while the only possible splicing in membrane 5 leads to the string $X'wY_k$ which stops here. No continuation is possible.
2. A string of the form X_1wY_0 can enter no splicing in membranes 2, 3, 4. In membrane 5 we have two possibilities:
 - 2.1. If we produce the string X_1wY' which is sent to the skin membrane, this string is transformed here into X_1wY'' and it must again enter an inner membrane. No splicing is possible on it in membranes 2, 3. In membrane 4 we can produce X_1wY , but the string does not leave the membrane. In membrane 5 we can produce $X'wY''$ and again the string is “lost”, because it cannot leave the membrane.
 - 2.2. If in membrane 5 we apply both productions, then we produce the string $X'wY'$ which is sent to the skin membrane. Here we produce $X'wY''$, which should again enter an inner membrane. In membranes 2, 3, 5 we can apply no splicing rule, but in membrane 4 we have two possibilities. If we produce the string XwY'' which is sent to the skin membrane, then the only possible splicing here is that by the rule $(X\#\$\#Z'; here, out)$. The obtained string leaves the system, but it is not terminal. If in membrane 4 we apply both rules, then we get XwY , which is the correct continuation. The process of simulating rules in P and of circularly permuting the string can be continued.
3. A string of the form $X_kwY_0, k \geq 2$, enters no splicing in membranes 2 and 4. In membrane 3 we can produce $X_{k-1}wY_0$, which is sent to the skin membrane, but no further splicing is possible here. In membrane 5 we can produce the string X_kwY' which is sent to the skin membrane. From membrane 1 we return to the inner membranes the string X_kwY'' .

This string can enter no splicing in membranes 2 and 5. In membrane 3 we can produce $X_{k-1}wY''$, which is sent to the skin membrane, where no further splicing can involve this string. In membrane 4 we can produce X_kwY , but this string does not leave the membrane.

In none of the cases different from X_1wY_0 we can produce a string which can be further processed.

We conclude that all computations in Π which lead to terminal strings which leave the system correspond to terminal derivations in G . Conversely, it is easy to see that each terminal derivation with respect to G can be simulated in Π . Consequently, $L(G) = L(\Pi)$. \square

We do not know whether or not these results can be improved. Here are some information about the power of systems of a lower degree (we present them without a proof; the reader is referred to [231] for details).

Theorem 3.24 (ii) $SPL(i/o, 1, 1) = REG$, (ii) $SPL(i/o, 2, 2) - MAT \neq \emptyset$.

P systems based on splicing can also work on asymmetric graphs, as already considered for P systems with symbol objects in Section 3.8. We denote by $SP'L(go, m)$ the family of languages generated by such systems of degree at most m , $m \geq 1$, and by $SP'L(go, *)$ the union of all these families.

As expected, also splicing P' systems are computationally universal. However, the proof of this assertion does not give a (small) bound on the degree of the necessary systems.

Theorem 3.25 $SP'L(go, *) = CE$.

Proof. Let $G = (N, T, S, P)$ be a type-0 Chomsky grammar, B a new symbol, $u_i \rightarrow v_i$, $1 \leq i \leq m$, the rules in P and $u_{m+j} \rightarrow v_{m+j}$, $1 \leq j \leq n$, new rules, with $u_{m+j} = v_{m+j} = \alpha_j$, for $N \cup T \cup \{B\} = \{\alpha_1, \dots, \alpha_n\}$. We construct the P' system (of degree $2(n + m + 1)$)

$$\begin{aligned} \Pi &= (V, T, g, L_0, L_{0'}, L_1, L_{1'}, \dots, L_{(n+m+1)}, L_{(n+m+1)'}, \\ &\quad R_0, R_{0'}, R_1, R_{1'}, \dots, R_{(n+m+1)}, R_{(n+m+1)'}), \\ V &= N \cup T \cup \{B, X, Y, Z, Z'\} \\ &\cup \{X_i, Y_i \mid 0 \leq i \leq n + m\}, \\ g &= (\{0, 0', 1, 1', \dots, (n + m + 1), (n + m + 1)'\}, \\ &\quad \{(i, i') \mid 0 \leq i \leq n + m + 1\} \\ &\cup \{(i', i + 1), (i', n + m + 1) \mid 0 \leq i \leq n + m\} \\ &\cup \{((n + m + 1)', 0)\}), \\ L_0 &= \{XSBY, Z'\} \cup \{ZY_i \mid 1 \leq i \leq n + m\}, \\ L_{0'} &= \{X_i v_i Z \mid 1 \leq i \leq n + m\}, \\ L_i &= \{ZY_j \mid 0 \leq j \leq n + m\}, \quad 1 \leq i \leq n + m, \end{aligned}$$

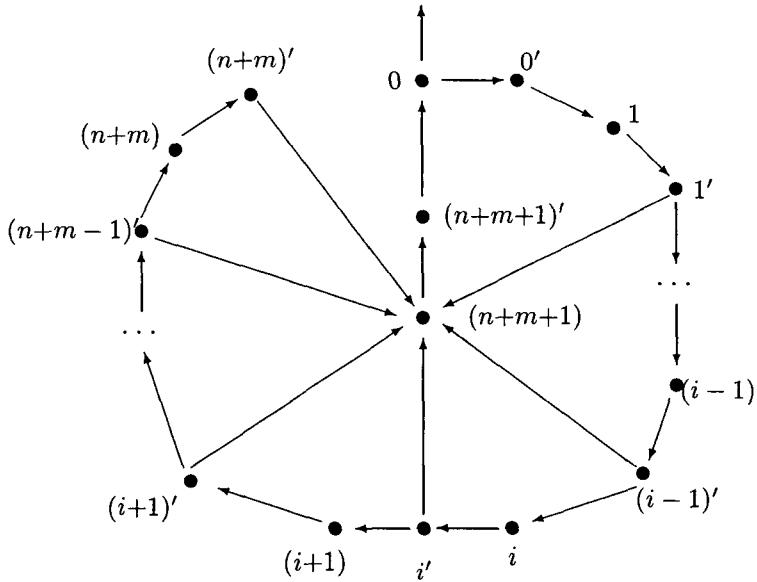


Figure 3.12: The graph of the system in the proof of the theorem.

$$\begin{aligned}
 L_{i'} &= \{X_j Z \mid 0 \leq j \leq n + m\}, \quad 1 \leq i \leq n + m, \\
 L_{(n+m+1)} &= \{XZ\}, \\
 L_{(n+m+1)'} &= \{ZY\}, \\
 R_0 &= \{(\#u_i Y \$ Z \# Y_i; go, here) \mid 1 \leq i \leq n + m\} \\
 &\cup \{(\#BY\$Z'\#; here, here), (X\#\$\#Z'; here, out)\}, \\
 R_{0'} &= \{(X\#\$X_i v_i \# Z; here, go) \mid 1 \leq i \leq n + m\}, \\
 R_i &= \{(\#Y_j \$ Z \# Y_{j-1}; go, here) \mid 1 \leq j \leq n + m\}, \quad 1 \leq i \leq n + m, \\
 R_{i'} &= \{(X_j \#\$\#X_{j-1} \# Z; here, go) \mid 1 \leq j \leq n + m\}, \quad 1 \leq i \leq n + m, \\
 R_{(n+m+1)} &= \{(X_0 \#\$X \# Z; here, go)\}, \\
 R_{(n+m+1)'} &= \{(\#Y_0 \$ Z \# Y; go, here)\}.
 \end{aligned}$$

The graph in this system and a planar map associated with it are given in Figures 3.12 and 3.13, respectively.

The system works as follows. Any string of the form Xwu_iY from region 0 (initially we have $XSBY$) enters here a splicing $(Xw|u_iY, Z|Y_i) \vdash (XwY_i, Zu_iY)$, for some $1 \leq i \leq n + m$; the string XwY_i passes to region $0'$, where we perform $(X|wY_i, X_jv_j|Z) \vdash (XZ, X_jv_jwY_i)$, for some $1 \leq j \leq n + m$. The string $X_jv_jwY_i$ passes now from region to region, clockwise, and in regions k the subscript of Y is decreased by one, while in regions k' the subscript of X is decreased by one. If in a region k we receive a string of the form X_rzY_0 , then this string cannot be processed. Similarly, if a string of the form X_0zY_s arrives in a region k' , then it cannot be processed.

In any moment, from a region k' we can pass to region $(n+m+1)$. Only strings of the form X_0zY_s can be spliced here: $(X_0|zY_s, X|Z) \vdash (X_0Z, XzY_s)$. The string XzY_s is passed to region $(n+m+1)'$, where we can process this string only if $s = 0$. This means that the subscripts of X and Y have reached the value 0 at the same time, that is we have $i = j$. The simulation of rules in P and the circular permutation of symbols is done in a correct manner.

Thus, any derivation in G can be simulated in Π , with the sentential form circularly permuted, and, conversely, each computation in Π corresponds to a correct derivation in G .

A string can leave the system only from region 0; if no nonterminal is present (hence also B and Y were removed), this means that the symbols B, Y were adjacent when removing them, that is, the obtained string is in the same permutation as in G . We conclude that $L(G) = L(\Pi)$. \square

We close this section by pointing out the naturalness of splicing P systems, which are intrinsically biochemical-like: both the membrane computing and the splicing operation are biochemically inspired. Of course, this means nothing in what concerns the possibility of implementing such computing devices.

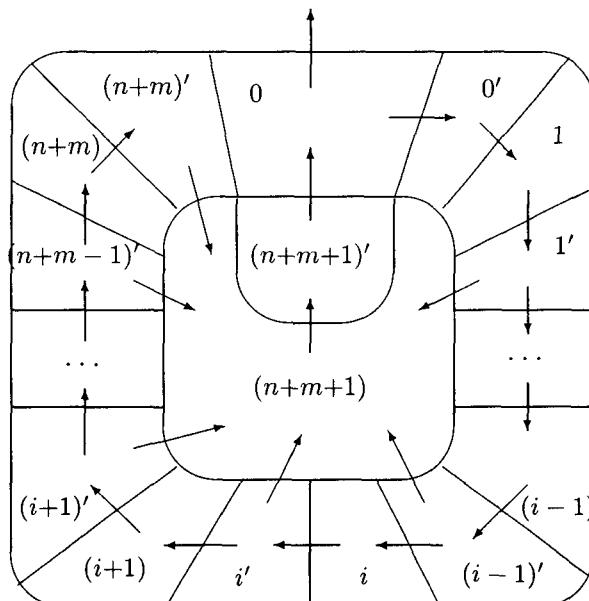


Figure 3.13: The planar map of the system in the proof of the theorem.

3.11 Variants, Problems, Conjectures

The possibility of inventing variants of the P systems seems to be limitless. One can go both towards considering simpler variants or more complex variants, looking either for adequacy to the biochemical origin of the idea or for mathematical appealing (whatever this means: elegance of definitions, strength of results, similarity with other areas of theoretical computer science). We do not intend to start here such an approach, but we just want to point out some questions which seem more natural and important for the immediate development of the theory.

First, the theory of computing with membranes misses basic tools for producing examples and counterexamples (necessary conditions). For instance, we believe that $\{a^n b^n \mid n \geq 1\} \notin LP(nPri, Cat, n\delta)$ and $\{a^n x_1 x_2 \dots x_n \mid n \geq 1, x_i \in \{de, ed\}, 1 \leq i \leq n\} \notin LP(Pri, nCat, \delta)$. (The idea supporting these conjectures is that we have to first send outside the system the prefix a^n , hence we have to store inside the system the information about n ; we can see no way of doing that which does not produce strings outside our languages.)

A basic question concerns the hierarchies on the number of membranes. We have seen above that in many cases these hierarchies collapse.

We *conjecture* that at least the hierarchy $LP_m(nPri, nCat, \delta), m \geq 1$, is infinite. A candidate sequence of languages for proving this assertion is that considered in Example 3.6. Similarly, we also believe that most of the hierarchies $LP_m^\pm(\alpha, \beta, \gamma)$, for $\alpha \in \{Cat, nCat\}$, $\beta \in \{\delta, n\delta\}$, and $\gamma \in \{\tau, n\tau\}$ are infinite.

The system in Example 3.6 has the *depth* equal to $m + 1$, but one can generate the same language by a system of depth two:

$$\begin{aligned}\Pi'_5 &= (V, \{a\}, \emptyset, \mu, a_1, \lambda, \dots, \lambda, (R_1, \emptyset), \dots, (R_{m+1}, \emptyset), \infty), \\ V &= \{a_1, \dots, a_m, a\}, \\ \mu &= [{}_1[{}_2] {}_2[{}_3] {}_3 \dots [{}_m] {}_m] {}_1, \\ R_1 &= \{a_i \rightarrow (a_{i+1})_{in_{i+2}} \mid 1 \leq i \leq m-1\} \\ &\cup \{a_m \rightarrow a_{out}\}, \\ R_{i+1} &= \{a_i \rightarrow a_i^{p_i}, a_i \rightarrow a_i^{p_i} \delta\}, \quad 1 \leq i \leq m.\end{aligned}$$

This suggests another *open problem* of interest: are there classes (α, β, γ) of P systems, $\alpha \in \{Pri, nPri\}$, $\beta \in \{Cat, nCat\}$, $\gamma \in \{\delta, n\delta\}$, for which the depth of the membrane structure induces infinite hierarchies of languages in families $LP(\alpha, \beta, \gamma)$? The same question holds for P systems with polarized membranes or using only *i/o* indications.

We have seen in Section 3.7 that the answer is negative for the case of families $LP_*^{i/o}(2Cat_2, \delta, \tau)$ and $LP_m(Pri, nCat, \delta), m \geq 1$

In [237] one conjectures that for families $LP_m(nPri, nCat, \delta), m \geq 1$, the hierarchy with respect to the depth is infinite.

In the systems above there is an asymmetry between the way of dealing with the objects and the rules in the case of membrane dissolving. Variants can be considered: with the rules preserved after membrane dissolving, or also with multisets of rules (thus, a limited parallelism is obtained). More generally, we can treat in the same way the rules and the objects, by means of certain meta-rules; thus, the rules can swim in the system, passing through membranes, and acting on objects when reaching each other in the same region. A natural variant is to leave the objects free in their regions and to consider certain “filters” associated with the membranes (in the sense used in networks of language processors, [71]); an object can pass through a membrane only when this is allowed by the associated filter.

Also natural is to look for a normal form where the same rules are used in all regions (then, by dissolving membranes we only handle objects, the rules are the same everywhere). Does such a restriction decrease the power of P systems of various types?

For all these variants, “classic” problems can be formulated: generative power, hierarchies (on the degree and on the depth), descriptional complexity, influence of determinism, of erasing (non-propagation). There also are several specific questions: normal forms, the power of parallelism and of synchronization (in the basic model, a universal clock is assumed, which is not very realistic from a biochemical point of view). Concerning normal forms, again we can consider classic problems (for instance, restricted forms of the rules), but also specific problems: Which is the influence of the topological properties of the membrane structure on the properties of the associated P systems? Can all P systems of degree m and of a given type (α, β, γ) be simulated by a P system with a fixed arrangement of membranes and of the same type?

Of a particular interest is the case of P systems whose membranes can be divided. For instance, a possible parameter for this case is the maximal number of membranes during a computation. Is this number computable? Does it give an infinite hierarchy of languages? Then, of a definite importance is the computational complexity direction of investigation. Which other NP-complete problems than SAT can be solved in a direct way by a P system, in polynomial time?

One more problem, of a clear natural computing significance (also related to quantum computing): what about *reversible* P systems? (We first need a good definition of reversibility – difficulties appear in the case of using the membrane dissolving action – and then we have to see how restrictive this condition is.)

All these variants and questions are valid both for the case when we work with symbol-objects and with string-objects. Of a special interest is the case when we deal with string transformations of a biochemical inspiration, thus having a more homogeneous *natural* computing model.

Of course, an important research topic is to implement a P system, either

in a biochemical media or in an electronic media. Attempts of the latter type were done in [179] and [278], but the problem still needs further efforts. What about designing a specific (silicon) hardware adequate for P systems?

3.12 Bibliographical Notes

The P systems were introduced in [218] and then investigated in a series of papers. An early survey of the domain can be found in [219]. Theorem 3.1 is from [226], Theorems 3.2 and 3.10 are from [218], Theorems 3.3–3.7 are from [76] and [226]. Theorems 3.8 and 3.9 are from [226]. Theorem 3.6 is from [220], where P systems with polarized membranes (and action τ) are introduced. Sections 3.7 and 3.8 are based on [227]; Theorem 3.15 is from [237]. Systems with active membranes were introduced in [221]; Section 3.9 is based on this paper.

Splicing P systems were already considered in [218], in a form different from that discussed in Section 3.10. In [218], the rules are given in the form $(r, y; tar)$, where r is a splicing rule, y is a string, and tar is a target indication. To apply such a rule to a string x means to perform a splicing $(x, y) \vdash_r (w, z)$ or $(y, x) \vdash_r (w, z)$; only the string w is considered as an output of this operation and it is sent to the region indicated by tar .) The variant of splicing P systems considered in Section 3.10 was introduced in [231], where Theorems 3.21–3.25 can be found.

Bi-stable catalysts were first considered in [233], where the role of synchronization is investigated. Unsynchronized P systems without priority, but using catalysts and action δ , generate only Parikh sets of matrix languages, which is less than $PsCE$; if both a priority and bi-stable catalysts are used, then unsynchronized P systems characterize again $PsCE$.

Further variants of P systems are introduced in [102] (systems working on graphs and on arrays).

In [77] one considers P systems where the objects are moved from a region to another one because of differences in their concentrations: after using evolution rules, one counts the number of occurrences of (certain) objects and one redistributes them among regions in such a way to have the same number of occurrences (plus/minus one) of these objects in all regions. Again a characterization of $PsCE$ is obtained when arbitrarily many bi-stable catalysts are used, while $PsMAT$ can be covered by concentration controlled P systems with only one catalyst.

A sort of one-membrane P systems is investigated in [230]: a multiset of symbol-objects is processed by a given set of gsm's. (Characterizations of the permutation closures of computably enumerable languages are obtained, both by systems with only two components, without a bound on the number of states in each component, or by systems with an unbounded number of components but with at most two states in each component.) One-membrane devices processing multisets of objects are considered in [279], [280], [281].

Chapter 4

Quantum Computing

The aim of this chapter is to offer a glimpse into Quantum Computing. We start by explaining that computation, be it mental, mechanic, molecular or silicon, is ultimately a physical process.

4.1 The Church–Turing Thesis

The Church–Turing Thesis, a prevailing paradigm in classical computation theory, states that no realizable computing device can be “globally” more powerful, that is, aside from relative speedups, than a universal Turing machine (see, for example, Odifreddi [198]). The modern form of the Church–Turing Thesis states that

any “reasonable” model of computation can be effectively simulated by a (probabilistic) Turing machine.

As one can immediately note, the above statement is a *thesis*, and not a theorem, as it relates an informal notion – a realizable computing device – to the mathematical notion of (probabilistic) Turing machine.

Here are some reasons supporting the Church–Turing Thesis:

- *Philosophical argument:* Due to Turing’s analysis it seems very difficult to imagine some other method which falls outside the scope of his description.
- *Mathematical evidence:* Every mathematical notion of computability which has been proposed was proven equivalent to Turing computability.
- *Sociological evidence:* No example of classical computing device which cannot be simulated by a Turing machine has been given, i.e. the thesis

has not been disproved despite having been proposed for more than 60 years.

The Church–Turing’s Thesis includes a syntactic as well as a physical claim. In particular, it specifies which types of computations are physically realizable. According to Deutsch ([84, p. 101]; see also [86]):

The reason why we find it possible to construct, say, electronic calculators, and indeed why we can perform mental arithmetic, cannot be found in mathematics or logic. The reason is that the laws of physics “happen” to permit the existence of physical models for the operations of arithmetic such as addition, subtraction and multiplication. If they did not, these familiar operations would be non-computable functions. We might still know of them and invoke them in mathematical proofs (which would be presumably called “non-constructive”) but we could not perform them.

It’s not surprising that the Church–Turing Thesis was challenged by logicians (Kalmar [153], Davis [78], Kreisel [154]; see also Odifreddi [198]), computer scientists (Rosen [254], Hogarth [138], Siegelmann [271, 272]) and physicists (Landuaer [158, 159, 160, 161, 162], Svozil [284, 285]). For example, Davis’ classical book [78, p. 11] includes the following question:

“... how can we ever exclude the possibility of our being presented, some day (perhaps by some extraterrestrial visitors), with a (perhaps extremely complex) device or “oracle” that “computes” an uncomputable function?”

Thinking is an essential, if not the most essential, component of human life – it is a mark of “intelligence”.¹ The Church–Turing Thesis has been used to approach formally the notion of “intelligent being”. In simple terms, the Church–Turing Thesis was stated as follows:

What is human computable is computable by a universal Turing machine.

This sentence equates information-processing capabilities of a human being with the “intellectual capacities” of a universal Turing machine. This discussion leads directly to the traditional problem of mind and matter which exceeds the aim of this chapter. Instead, we will concentrate on its physical aspects.

¹Descartes placed the essence of being in thinking.

4.2 Computation is Physical

An operation is “logically reversible” if it can be *undone*, if it can be run *backwards*, that is, if its inputs can always be deduced from the outputs. Most logical gates are irreversible; a typical example is the NAND gate

$$(a, b) \mapsto \neg(a \wedge b) \quad (4.1)$$

which has two input bits and only *one* output bit. We cannot recover a unique input from the output bit because the result 1 can be obtained from three distinct inputs: $(0, 0), (0, 1), (1, 0)$.

Assume we operate the gate NAND with two Boolean variables, a, b , and suppose that the four initial states, $(0, 0), (0, 1), (1, 0), (1, 1)$, have the same probability distribution, $\frac{1}{4}$. Then, the initial entropy, which is calculated with Shannon’s formula²

$$H = - \sum_i p_i \cdot \log p_i,$$

is then

$$H_{initial} = -4 \cdot \left(\frac{1}{4} \log \frac{1}{4}\right) = 2 \text{ bits.}$$

The result will be a system with only two possible states, 0 and 1, the outcome 0 appearing with probability $\frac{1}{4}$ and the outcome 1 appearing with probability $\frac{3}{4}$. Consequently, the final entropy is

$$H_{final} = -\left(\frac{3}{4} \log \frac{3}{4} + \frac{1}{4} \log \frac{1}{4}\right) = 2 - \frac{3}{4} \log 3 \text{ bits,}$$

which means a loss of

$$H_{initial} - H_{final} = \frac{3}{4} \log 3 \text{ bits.}$$

Assume now that we operate the gate

$$(a, b) \mapsto (a \vee b, a \wedge b),$$

and, again, suppose that the four initial states of the Boolean variables a, b have the same probability distribution, $\frac{1}{4}$. This gate has finally only *three* final states, namely $(0, 0), (1, 0), (1, 1)$, two of them with probability $\frac{1}{4}$ and one with probability $\frac{1}{2}$. Consequently, the final entropy is

$$H_{final} = -(2 \cdot \frac{1}{4} \log \frac{1}{4} + \frac{1}{2} \log \frac{1}{2}) = 1.5 \text{ bits.}$$

In this case, the gate decreases the entropy by 0.5 bits.

²Here log is the logarithm in base 2.

The first gate is “more irreversible” than the second one, since it decreases the entropy more.

In thermodynamics the entropy is defined by

$$S = -k \cdot \sum_i p_i \cdot \ln(p_i).^3$$

This notion is coupled to energy through the temperature T of the system: when the entropy of a system is decreased by some amount, then the system dissipates energy equal to the amount of entropy reduction times the temperature. Von Neumann noticed that the two entropies are related by some constant factor, so they are in fact the same notion. When the probability distribution of the system is changed so that the entropy H is decreased by 1 bit, then the entropy S is decreased by $k \cdot \ln 2$ joule/°kelvin, and the system dissipates $kT \cdot \ln 2$ joules of energy in the form of heat. So, a challenging question arises:

*what is the minimum energy required to carry out a computation?*⁴

Does the above analysis apply to computation? In 1961 Landauer [158] (see also [161]) has produced evidence for the affirmative answer. To operate a computer we have to make sure that distinct logical states are represented by distinct physical states. Each bit has two values, 0, 1, so it has one degree of freedom; it corresponds to one or more degrees of freedom of physical states. In general, a set of n bits has n degrees of freedom; they correspond to 2^n physical states. If we erase n bits⁵, say we reset all to 0, then we have compressed 2^n logical states into a single state, a loss of entropy. The irreversible loss information increases temperature of the system, which means heat dissipation. Consequently, operations which are not one-to-one, which map distinct logical states into a common one, *cost energy*. This cost is expressed by *Landauer’s principle*:

erasure of information is a dissipative process.

Here is a simple example. One can store one bit of information by placing a single molecule in a box, either on the left side or the right side of a partition that divides the box. In this context, erasure means that we choose to move the molecule to the left (or right) side irrespective of whether it started out on the left or right. However, one can suddenly remove the partition, and then slowly compress the one-molecule “gas” with a piston until the molecule reaches the left side. This procedure reduces the entropy of the gas by $\Delta S = k \cdot \ln 2$, and a flow of heat from the box to the environment is produced. Assume now that the process is isothermal at temperature T . Then work

³Here $k \approx 1.38 \times 10^{-23}$ joule/°kelvin is Boltzmann’s constant.

⁴See Feynman [98] for a detailed discussion.

⁵Which could be in any of the 2^n possible logical states.

$$W = kT \cdot \ln 2 \quad (4.2)$$

is performed on the box, and this work has to be provided. If one decides to erase information, then “a power bill will be generated” and should be paid. For example, according to formula (4.2), the execution of the gate $(a, b) \mapsto (a \vee b, a \wedge b)$ dissipates at least $\frac{1}{2}kT \cdot \ln 2$ joules of energy. This is a theoretical limit expressing how long the gate can be operated with finite resources of energy.

The energy dissipation has been reduced by approximately a factor of ten every five years, so a rough extrapolation suggests that a reduction of the energy dissipation per logic operation below kT (thermal noise, that is of the order of 10^{-18} picojoule at room temperature) may become relevant in about 10-15 years. This issue may cause a variety of problems for classical computers, e.g. cooling may be difficult (according to current day knowledge/technology).

4.3 Reversible Computation

Irreversible operations as the NAND gate (4.1), the binary addition $(a, b) \mapsto (a \oplus b, a \wedge b)$ (sum and carry) and the real addition $(x, y) \mapsto x + y$, dissipate energy. Is logical reversibility dissipation free?

First, let’s note that the above irreversible operations can be easily simulated by reversible ones. A reversible version of the NAND gate⁶ is, for example, Toffoli’s gate

$$(a, b, c) \mapsto (a, b, c \oplus (a \wedge b)).^7 \quad (4.3)$$

Indeed, (4.3) is a reversible 3-bit gate that flips the third bit if the first two both take the value 1 and does nothing otherwise. Hence, the third output bit becomes the NAND of a and b in case $c = 1$. The price paid to get reversibility consisted of adding a new variable c .

Similar tricks can be used to produce reversible versions of the binary addition, $(a, b) \mapsto (a, a \oplus b, a \wedge b)$ and real addition $(x, y) \mapsto (x + y, x - y)$. In the first case we replicated the first variable a ; in the second case we added a new component storing some additional value.

A computer may be fully reversible and yet dissipate energy! The important point is that the laws of physics allow for technologies to make reversible computers operate with negligible dissipation. To build a reversible computer one needs only two types of logical gates, say AND and NOT (because any other gate can be constructed from these two types – they are *universal*).

⁶A single NAND gate is as good as having both AND and NOT: $\neg a = \text{NAND}(a, 1)$, $\text{AND}(a, b) = \neg(\text{NAND}(a, b)) = \text{NAND}(\text{NAND}(a, b), 1)$.

⁷Recall that $a \oplus b$ is 1 only if a and b have different values, i.e. $a = 0, b = 1$ or $a = 1, b = 0$.

Clearly, the NOT gate is reversible as its composition with itself gives the initial input. However, the AND gate is irreversible. Reason: it has two inputs and only one output, so it has to lose information (it is impossible to tell exactly what inputs must have been if all one is told is the output 0: any of the three combinations $(0, 0), (0, 1), (1, 0)$, could have been the “real input”).

To make a reversible variant of the gate AND we need to make sure that we have the same number of output lines as input ones, so, in principle, we can just add some “garbage” output lines to solve the problem. However, this may not be enough, as we want to guarantee also universality! One possibility is Toffoli’s [101] reversible 3-bit gate which uses in addition to a, b a control bit c . Input bits a and b do not change their states; the control bit, however, will change its state, but only when $a = b = 1$. Toffoli’s truth table is the following:

input			output		
a	b	c	a	b	c
0	0	0	0	0	0
0	1	0	0	1	0
1	0	0	1	0	0
1	1	0	1	1	1

input			output		
a	b	c	a	b	c
0	0	1	0	0	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

Table 4.1: Toffoli’s gate.

Fredkin’s [101] reversible 3-bit gate also uses in addition to a, b a control bit c in the following way: (a) if $c = 0$, then the values of a, b are transmitted unaltered, i.e. the output is the pair (a, b) , (b) if $c = 1$, then the values of a, b are switched to the opposite output, i.e. the output is the pair (b, a) . Its truth table is the following:

input			output		
a	b	c	a	b	c
0	0	0	0	0	0
0	1	0	0	1	0
1	0	0	1	0	0
1	1	0	1	1	0

input			output		
a	b	c	a	b	c
0	0	1	0	0	1
0	1	1	1	0	1
1	0	1	0	1	1
1	1	1	1	1	1

Table 4.2: Fredkin’s gate.

Fredkin’s gate is universal in the sense that it can be used to construct

reversible variants of all Boolean gates, and satisfies one additional requirement: the number of 1s and 0s never changes.⁸ To prove universality it is enough to show that the gates NOT and AND can be represented using Fredkin's gate FREDKIN with particular inputs. It is not difficult to check that the following formulae work:

$$\begin{aligned}\text{FREDKIN } (a, b, c) &= (((\neg c) \wedge a) \vee (c \wedge b)), ((a \wedge c) \vee (b \wedge (\neg c)), c), \\ \text{FREDKIN } (1, 0, c) &= (\neg c, c, c),\end{aligned}\tag{4.4}$$

and

$$\text{FREDKIN } (0, b, c) = (b \wedge c, b \wedge (\neg c), c),\tag{4.5}$$

So, both NOT and AND can be simulated by FREDKIN.

Fredkin's gate has often been used for photon based gates where a 1 represents a photon and a 0 simply denotes the absence of a photon; nonlinear optics is used to control the output of an interferometer (see Milburn [192]). The number of 1s cannot change as the number of photons cannot change – absorption is not allowed for reversible gates.

In both formulae (4.4) and (4.5) there are more outputs than are required for the computed functions (one for NOT and two for AND): these outputs, called *garbage* bits, are a necessary consequence of reversible logic. Consequently, one may wonder whether we have only postponed the energy cost; garbage bits can be irreversibly erased, but that would require to pay Landauer's price ...

Bennett [23] found in 1973 that any computation can be performed using only reversible steps, and so “in principle” requires no dissipation and no power expenditure: *we do not need to erase the garbage bits!* By pointing out that a reversible computer can run forward to the end of a computation, print out a copy of the answer (a logically reversible operation) and then reverse all of its steps to return to its initial configuration, Bennett invented a procedure to remove the garbage without any energy cost. Here is a simple illustration of this technique (cf. Milburn [193]):

```
INPUT00000 ↔ COMPUTER00000 ↔ INPUTOUTPUT ↔
COPYOUTPUT ↔ RETUPMOC ↔ INPUT00000 ↔ OUTPUT
```

The inevitability of handling garbage bits implies the necessity to allow more input bits than are needed in an irreversible computation ($\text{INPUT} \leftrightarrow \text{INPUT00000}$); of course, some garbage bits may be useful only for internal computation, but still, a certain number of additional bits will be required.

Consequently, in principle, we need not pay any “power bill” to compute reversibly. In practice, the (irreversible) computers in use today dissipate

⁸Note that Toffoli's gate does not satisfy this condition; for example, $(1, 1, 0)$ is transformed into $(1, 1, 1)$.

energy of orders of magnitude more than (4.2) per gate, so *today* Landauer's limit seems not to be an important engineering principle. But as computing hardware continues to shrink in size, it may become important to beat Landauer's limit, for example, to prevent the components from melting. Then reversible computation may be one, if not the only one, option (for more information we refer to the results of MIT group on reversible computation reported in [47]).

4.4 The Copy Computer

In this section we are going to follow Bennett [24] to discuss a very simple, almost dumb, computation: the copy computation. Interestingly enough, nature uses extensively this "computation".

In the beginning it is not at all obvious that we can copy information from one place to another "for free", that is, without expending at least some energy. Indeed, if the speed of the copying process is reasonable low, then, copying can be done for free. Here is how one can do it, at least in principle.

Assume that we have a Turing machine and we consider a message and its copy as two (identical) messages on tape. If we know what the original message was, then we need no energy to waste in order to clear the tape and none need be expended for the copy tape: we just turn it over when necessary. However, if we don't know what the original message was, then clearing the tape will cost energy, but *not for the copy*. We can use the first tape content to clear the copy by turning bits over again. Reason: there is no more information in the original message plus copy than in the original message. Consequently, clearing the band should not require more energy in the first case than in the second.

By now the reader knows what proteins are and how crucial they are for the structure and functioning of living organisms and ... Molecular Computing. Protein synthesis is done in two stages, but in what follows we will be interested solely in the first one⁹, in which the formation of another linear strand of sugar phosphates with bases attached is produced with the help of the messenger RNA. "Computationally", the code on the DNA is *copied* onto the RNA strand base by base, according to a matching rule. The completed RNA leaves the nucleus and travels elsewhere. The "computer"¹⁰ which executes the copying operation is an enzyme called RNA polymerase.

What actually happens has been directly or implicitly already described. The DNA and enzyme are floating around in a soup which contains (among other phosphates) nucleotides with two extra phosphates attached. The en-

⁹Recall that the "rules" reside in the DNA. DNA comprises a double chain, each strand of which is made up of alternating phosphate and pentose sugar groups by means of the four bases A, T, C, G. A specific sequence of bases provides the code for protein synthesis.

¹⁰Christened "Brownian computer" by Bennett.

zyme attaches itself to the part of the DNA strand which will be duplicated: it moves along it and builds the RNA copy, base by base.¹¹ The nucleotides are provided in the triphosphate form; in the process of addition, two of the phosphates will be released into the soup, but in an “entangled” way as a pyrophosphate.¹²

Enzymes are merely catalysts: they help reactions, but they don’t influence the direction in which they proceed. Chemical reactions are *reversible*, so it is equally possible for the polymerase reaction to go the other way, that is to *undo* the RNA chain. The relative concentration of pyrophosphates and triphosphates may trigger the reaction one way or another. If there are a lot of triphosphates, then there are more forward-moving than backward-moving states available, so RNA polymerase will tend to group towards the former state, and conversely. If the concentration is about right, then the copier will oscillate forever and will fail to make the copy. In a real cell the concentration of pyrophosphates is low (by hydrolysis), so the copy “computer” will work.

The dissipated energy is about $20\text{-}200 \text{ } kT$ per logical step, which is not very efficient, but far less than the energy dissipation per logical step in any conventional computer.¹³ This copy “computation” can be made arbitrarily close to being reversible by “running” it sufficiently slow, but, for the sake of life, the reaction has to be done at a certain speed!

Other models of reversible computers include the ballistic (billiard ball) computer invented by Fredkin and Toffoli [101] and a modified variant of the billiard ball computer by Bennett and Landauer [29].

4.5 Maxwell’s Demon

The above analysis led Bennett [24, 25] in 1982 to the reconciliation of Maxwell’s demon with the second law of thermodynamics. Maxwell’s Demon, a hypothetical intelligent entity capable of performing measurements on a thermodynamic system, was imagined by James Clerk Maxwell to “contradict” the second law of thermodynamics; see Maxwell [185], Leff and Rex [163]. Suppose that you have a box filled with a gas at some temperature. The average speed of the molecules depends on the temperature. Some of the molecules will be going faster than average and some will be going slower than average. Suppose that the box is divided by a partition into two parts, with both sides of the box filled with the gas at the same temperature. Maxwell imagined a molecule sized trapdoor in the partition and a demon poised at the door to observe molecules. The demon observes the molecules in the box as they approach the trapdoor, allowing fast ones to pass from left to

¹¹An essential ingredient is the use of “complementarity” relationships; see more in Chapter 2.

¹²The chosen nucleotide should be chosen correctly, that is complementary to the base on the DNA strand which is to be copied.

¹³A transistor dissipates about $10^8 \text{ } kT$.

right, and slow ones from right to left. After performing these operations the demon ends up with a box in which all the faster than average molecules are in the left side and all the slower than average ones are in the right side. So the box is hot on the left and cold on the right, and the expenditure of work is negligible, in apparent violation of the second law of thermodynamics.¹⁴

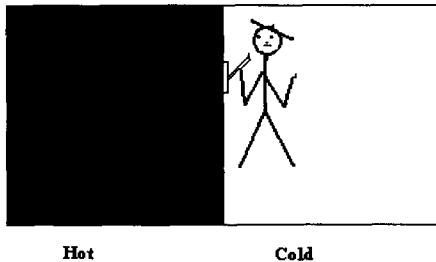


Figure 4.1: Maxwell’s Demon.

Bennett built his solution on Szilard’s [286] subtle analysis of the Maxwell demon.¹⁵ The resolution is that the demon *must* collect and store information about the molecules in the box (position and velocity of each molecule). A demon with a finite memory cannot continue to cool the gas indefinitely; *eventually, information must be erased*, and, as we now know, at exactly that point, we have to pay the “power bill”. This argument has been extended to quantum physics and may be even experimentally testable, cf. Lloyd [172]; see also the discussion in Zurek [308].

The moral we draw from the above discussion is, once again, that, to use Landauer’s [161] words,

information is inevitably physical.

So, let’s explore what physics, especially quantum physics, has to tell us about information.

¹⁴One can use this device to run a heat engine by allowing the heat to flow from the hot side to the cold side. Another possible action of the demon is to observe the molecules and only open the door if a molecule is approaching the trapdoor from the right. This would result in all the molecules ending up on the left side. Again this setup can be used to run an engine.

¹⁵Szilard invented the concept of a bit of information and associated the entropy $\Delta S = k \cdot \ln 2$ with the acquisition of one bit. However, the name “bit” was introduced only in 1946 by the statistician John Tukey.

4.6 Quantum World

On 19 October 1900, Max Planck, a 42 year old physicist, presented to a meeting of the German Physical Society a ground-breaking solution to a long-standing problem: Why does the colour of radiation from any glowing body change from red to orange and ultimately to blue as its temperature increases? Planck based his answer on the assumption that radiation (like matter) comes in *discrete quantities* of energy, the so-called “quanta”.¹⁶ The quantization of energy proved to be a revolutionary and fundamental rule of nature, with very strange conclusions contradicting what common sense or classical Newtonian mechanics imposed: things change because you “look” at them, an action can precede its cause (Compton effect), travelling faster than light might be possible (EPR), objects have “random” behaviour, etc.¹⁷

How come that an innocuous statement about energy can determine such an alarming behaviour as the uncertainty principle (one can never measure anything as accurately as one would like, to put it in a rather misleading way)?

Here is a simple example suggested by Feynman. Consider a mirror which reflects a light, say in a proportion of 95% : 5% of light passes through and it is absorbed or lost while 95% bounces off the mirror’s surface.

If light is seen as a continuous stream of energy then we don’t have any difficulty in explaining the above phenomenon. However, if light is a stream of indivisible quanta (photons), then we get a problem: each photon is either reflected or absorbed in its entirety, but one cannot accept that 5% of a photon goes into one direction and the remainder of 95% goes into a different direction! Consequently, one is led to the conclusion that out of 20 photons, 19 bounce off the mirror’s surface and just one photon goes in a different direction. OK, this is not difficult to accept, but *who* decides which photon goes astray? This is the crucial question!

Quantum theory claims that we cannot know the answer to the above question: what happens to any individual photon is completely *unpredictable*! We can only talk about chance, about probability: a photon will have a 95% chance of bouncing off and 5% chance of being absorbed/transmitted. One cannot say more. The unpredictability/randomness is part of the game, it’s innate! In opposition with classical physics, in quantum theory one can only describe a quantum “state” of a photon in terms of its probabilities and probabilities themselves *change* depending upon what one plans to do with the photon.

Confusing? Alarming? Shocking? Dangerous?

¹⁶The Latin for amount.

¹⁷Planck felt very uncomfortable himself with the idea that quantization is a law of nature. In this connection, he is notorious for saying that “new scientific theories supplant previous ones not because people change their minds, but simply because old people die.”

From the time of original discoverers to the present day people expressed wonder and disbelief:

Quantum mechanics is very impressive. But an inner voice tells me that it is not the real thing. The theory produces a great deal but hardly brings us closer to the secrets of the old one. A. Einstein.

... we always have had (secret, secret, close the doors!) ... a great deal of difficulty in understanding the world view that quantum mechanics represents. At least I do, because I'm an old enough man that I haven't got to the point that this stuff is obvious to me. Okay, I still get nervous with it. R. Feynman.

In spite of all of this, quantum theory is one of the most successful theories: it makes unbelievable good predictions which were tested to an unprecedented degree of accuracy, it underpins modern technology, from supermarket laser scanners to all goodies of microelectronics and ... quantum computers.

4.7 Bits and Qubits

A classical **bit** (e.g. the position of gear teeth in Babbage's differential engine, a memory element or wire carrying a binary signal, in contemporary machines) is a system comprising many atoms. Typically, the system is described by one or more continuous parameters, for example, voltage. Such a parameter is used to separate the space into two well-defined regions chosen to represent 0 and 1. Manufacturing imperfections and local perturbations may affect the signal, so signals are periodically restored near these regions to prevent them from drifting away. An n -bit register of memory can exist in any of 2^n logical states, from 00...0 (n zeros) to 11...1 (n ones).

A quantum event in which we have two possible mutually exclusive outcomes is the elementary act of observation: all knowledge of the physical world is based upon such acts. An elementary act of observation is simultaneously like a coin-toss and not like a coin-toss. The information derived from an elementary act of observation is no more than a single bit, but *there is more on it than that*. To mark this difference Schumaker [265] has coined the name qubit. A quantum bit, **qubit**, is typically a microscopic system, such as an atom or nuclear spin or polarized photon. For example, the state of a spin- $\frac{1}{2}$ particle, when measured, is always found to be in one of two possible states, represented as

$$\left| +\frac{1}{2} \right\rangle \text{ (spin-up)} \text{ or } \left| -\frac{1}{2} \right\rangle \text{ (spin-down).}$$

The intrinsic discrete character is a consequence of quantization discussed in Section 4.6. Consequently, one can use one spin state to represent 0, and

the other spin state to represent 1. There is nothing special about spin systems – any 2-state quantum system can be equally used to represent 0 and 1. What is really special here is the existence of a continuum of intermediate states which are superpositions of 0s and 1s.¹⁸ Unlike the intermediate states of a classical bit (for example, any voltages between the “standard” representations of 0 and 1) which can be distinguished from 0 and 1, but do not exist from an informational point of view, quantum intermediate states cannot be reliably distinguished, even in principle, from the basis states, but do have an informational “existence”.

An n -qubit system can exist in any superposition of the form

$$\Psi = \sum_{x=00\ldots0}^{11\ldots1} c_x |x\rangle, \quad (4.6)$$

where c_x are (complex) numbers such that $\sum_x |c_x|^2 = 1$.¹⁹ The exponential “explosion” represented by formula (4.6) distinguishes quantum systems from classical ones: in a classical system a state is described by a number of parameters growing only linearly with the size of the system,²⁰ but, as we shall see in the next section, quantum systems may not admit such a description (because quantum states may be “entangled”).

4.8 Quantum Calculus

For a better understanding we need some rudiments of (finite dimensional) Hilbert space theory. A Hilbert space is a mathematical model for representing vectors.²¹ The state of a quantum system can be described by a column vector in a Hilbert space of wave functions;²² as the system evolves, its state vector rotates with its base anchored to the origin of axes. Vectors can be added and multiplied by (complex) numbers. State vectors are typically written with a special angular bracket notation, the “ket vector” $|\Psi\rangle$.²³ Row vectors, such as $\langle\Psi|$, are known as “bra” vectors; when you put together a column and a bra vector, you get a bracket, that is the inner product of the two vectors, $\langle\Psi|\Psi\rangle$, also written as $\langle\Psi|\Psi\rangle$.

A simple 2-state quantum system (the basic block of a quantum memory register) can, by definition, be in one of two possible states. To model it

¹⁸Mathematically, as we will see, they are just linear combinations of the basis states.

¹⁹Re-phrased, a quantum state of n qubits is just a direction in a Hilbert space of dimension equal to the number of classical states, i.e. 2^n .

²⁰Reason: classical systems are completely described locally, that is, via each state in part.

²¹Formally, a Hilbert space is a complex linear vector space, with an inner product which is complete with respect to the induced norm.

²²To be precise, a state is a ray in a Hilbert space, i.e. an equivalence class of vectors that differ by multiplication by a non-zero complex number.

²³The word “ket” was invented by Paul Dirac [90].

we need the smallest non-trivial Hilbert space \mathbf{C}^2 , a two dimensional space. Assume that a particular complete orthonormal basis, denoted by $\{|0\rangle, |1\rangle\}$, has been fixed.²⁴ These vectors, $|0\rangle$ and $|1\rangle$, correspond to the classical bit values 0 and 1, respectively.

A qubit is a unit vector in the space \mathbf{C}^2 , so for each qubit $|x\rangle$, there are two (complex) numbers $a, b \in \mathbf{C}$ such that

$$|x\rangle = a|0\rangle + b|1\rangle = \begin{pmatrix} a \\ b \end{pmatrix}, \quad (4.7)$$

where

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

and $|a|^2 + |b|^2 = 1$.

The angle which a qubit makes with the vertical axis describes the relative contributions of $|0\rangle$ and $|1\rangle$. The angle through which the vector is rotated about the vertical axis induces the so-called “phase”. So, different qubits may have the same proportion of $|0\rangle$ and $|1\rangle$, but with different phase factors. Phase is irrelevant for the whole states but it’s crucial for “quantum interference effects”.

We can perform a measurement that projects the qubit onto the basis $\{|0\rangle, |1\rangle\}$. Then we will obtain the outcome $|1\rangle$ with probability $|b|^2$, and the outcome $|0\rangle$ with probability $|a|^2$. With the exception of limit cases $a = 0$ and $b = 0$, the *measurement irrevocably disturbs the state*: If the value of the qubit is initially unknown, then there is no way to determine a and b with any conceivable measurement. However, *after* performing the measurement, the qubit has been prepared in a known state (either $|0\rangle$ or $|1\rangle$); this state is typically different from the previous state.

The above facts point out an important difference between qubits and classical bits. There is no problem in measuring a classical bit without disturbing it, so we can decode all of the information that it encodes. If we have a classical bit with a fixed, but unknown value (0 or 1), then we can only say that there is a probability that the bit has the value 0, and a probability that the bit has the value 1, and these two probabilities add up to 1. When we measure the bit, we acquire additional information; after measurement, we will know *completely* the value of the bit.

The ability of quantum systems to exist in a “blend” of all their allowed states simultaneously is known as the *Principle of Superposition*. Even though a qubit can be put in a superposition (4.7), it contains no more information than a classical bit, in spite of its having infinitely many states. The reason is that information can be extracted only by measurement. But,

²⁴Here “complete” refers to the fact that every state vector in the Hilbert space can be represented in the form (4.7), and “orthonormal” means that vectors are perpendicular to one another, and normalized. For example, the vectors $|0\rangle$ and $|1\rangle$ may correspond to the horizontal polarization $|\rightarrow\rangle$ and the vertical polarization $|\uparrow\rangle$ of a photon, respectively.

as we have argued, for any measurement of a qubit with respect to a given orthonormal basis, there are only two possible results, corresponding to the two vectors of the basis. On the other hand, it is not possible to capture more information measuring in two different bases because the measurement changes the state. Even worse, quantum states cannot be cloned, hence it's impossible to measure a qubit in two different ways (even, indirectly, by using a copy trick, that is copying and measuring the copy).

Is a qubit identical to a probabilistic classical bit? The answer is negative and an argument is that the numbers a and b in (4.7) encode *more* than just the probabilities of the outcomes of a measurement in the $\{|0\rangle, |1\rangle\}$ basis. For example, the relative phase of a and b is crucial.

Systems of more than one qubit need a Hilbert space which captures the interaction of the qubits. A two qubit system can be represented by a unit vector in the tensor product of two copies of \mathbf{C}^2 , i.e. the space $\mathbf{C}^2 \otimes \mathbf{C}^2$. Using Dirac notation, if $|0\rangle$ and $|1\rangle$ are the vectors of a basis in \mathbf{C}^2 then, the set

$$\{|0\rangle \otimes |0\rangle, |0\rangle \otimes |1\rangle, |1\rangle \otimes |0\rangle, |1\rangle \otimes |1\rangle\} = \{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$$

is a basis in $\mathbf{C}^2 \otimes \mathbf{C}^2$; more precisely,

$$\begin{aligned} |00\rangle &= \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, & |01\rangle &= \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \\ |10\rangle &= \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, & |11\rangle &= \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}. \end{aligned}$$

In general, a system containing exactly $n \geq 2$ qubits is represented by n copies of \mathbf{C}^2 tensored together. Therefore, the state space is 2^n dimensional. A natural basis for this space consists of 2^n tensor products:

$$|0\rangle \otimes |0\rangle \otimes \dots \otimes |0\rangle,$$

$$|0\rangle \otimes |0\rangle \otimes \dots \otimes |1\rangle,$$

⋮

$$|1\rangle \otimes |1\rangle \otimes \dots \otimes |1\rangle.$$

A classical string of bits $i_1 i_2 \dots i_n$ with $i_k \in \{0, 1\}$, $1 \leq k \leq n$, corresponds to the quantum state $|i_1\rangle \otimes |i_2\rangle \otimes \dots \otimes |i_n\rangle$ which is simply denoted by $|i_1 i_2 \dots i_n\rangle$. If $|0\rangle$ and $|1\rangle$ are orthogonal unit vectors in \mathbf{C}^2 , then the set

$$\{|i_1 i_2 \dots i_n\rangle | i_k \in \{0, 1\}, 1 \leq k \leq n\}$$

is an orthonormal basis in $\mathbf{C}^2 \otimes \mathbf{C}^2 \otimes \dots \otimes \mathbf{C}^2$.

In contrast with the classical physics, where the state of a system is completely defined by describing the state of each of its component pieces separately, in a quantum system the state cannot always be described considering only the component pieces. For instance, the state

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

cannot be decomposed into separate states for each of the two bits. This means that we cannot express this state as a tensor product of two single qubits. Indeed, let's assume for the sake of a contradiction, that there exist two kets $|x\rangle$ and $|y\rangle$ in \mathbf{C}^2 such that

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = |x\rangle \otimes |y\rangle.$$

Since each single qubit is in a superposition of $|0\rangle$ and $|1\rangle$, there exist four complex numbers a_1, b_1, a_2, b_2 such that

$$|x\rangle = a_1|0\rangle + b_1|1\rangle$$

and

$$|y\rangle = a_2|0\rangle + b_2|1\rangle.$$

It follows that

$$\begin{aligned} |x\rangle \otimes |y\rangle &= (a_1|0\rangle + b_1|1\rangle) \otimes (a_2|0\rangle + b_2|1\rangle) \\ &= a_1a_2|00\rangle + a_1b_2|01\rangle + b_1a_2|10\rangle + b_1b_2|11\rangle, \end{aligned}$$

hence $a_1b_2 = 0$ and $a_1a_2 = \frac{1}{\sqrt{2}} = b_1b_2$, which is impossible.

A state that cannot be expressed as a tensor product is called an **entangled state**. Since the space $\mathbf{C}^2 \otimes \mathbf{C}^2$ is spanned by the set $\{|x\rangle \otimes |y\rangle \mid x, y \in \mathbf{C}^2\}$, the existence of entangled states proves that the previous set is not a linear space. One can easily find entangled states in an n qubit system, for any integer $n \geq 2$.

Note that it would require vast resources to simulate even a small quantum system on a conventional computer, as such a simulation would require keeping track of exponentially many states: the dimension of the cartesian product of multiple classical particles grows linearly with the number of particles, while the dimension of the tensor product of quantum systems grows exponentially. A reason for the (potential) power of quantum computers is the ability of exploiting the quantum state evolution as a computational mechanism.

4.9 Qubit Evolution

The quantum evolution (quantum transformation, operator) of (on) a qubit is described by a “unitary operator”, that is an operator induced by a unitary matrix.²⁵

We will present some simple examples of single qubit quantum state transformations. Any unitary operator $U : \mathbf{C}^2 \rightarrow \mathbf{C}^2$ can be viewed as a single qubit gate. Considering the basis $\{|0\rangle, |1\rangle\}$, the transformation is fully specified by its effect on the basis vectors. In order to obtain the associated matrix of an operator U , we put the coordinates of $U|0\rangle$ in the first column and the coordinates of $U|1\rangle$ in the second one. So, the general form of a transformation that acts on a single qubit is a 2×2 matrix

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix},$$

which transforms the qubit state $\alpha|0\rangle + \beta|1\rangle$ into the state $(\alpha a + \beta b)|0\rangle + (\alpha c + \beta d)|1\rangle$:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \alpha a + \beta b \\ \alpha c + \beta d \end{pmatrix}.$$

For an arbitrary real number $\theta \in [0, 2\pi)$, the rotation R_θ is given by

$$R_\theta = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}.$$

Hence, R_θ acts as follows:

$$|0\rangle \mapsto \cos \theta |0\rangle + \sin \theta |1\rangle, \quad |1\rangle \mapsto -\sin \theta |0\rangle + \cos \theta |1\rangle.$$

One can easily verify that $R_\theta R_\theta^\dagger = R_\theta R_\theta^T = I$, hence R_θ is unitary. Note that in the special case $\theta = 0$ we get the identity transformation of \mathbf{C}^2 :

$$R_0 = I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

We may think of logic gates as transformations. For example, the NOT transformation which interchanges the vectors $|0\rangle$ and $|1\rangle$, is given by R_π , that is the matrix

$$\text{NOT} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

It flips that state of its input,

$$\text{NOT } |0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle,$$

²⁵A quadratic matrix A of order n over \mathbf{C} is *unitary* if $AA^\dagger = I$ (the identity $n \times n$ matrix); A^\dagger is the transposed conjugate matrix of A .

and

$$\text{NOT } |1\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle.$$

The phase shift gate *Shift* is defined by the following operator: $\text{Shift}|0\rangle = |0\rangle$, $\text{Shift}|1\rangle = -|1\rangle$, so

$$\text{Shift} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Since $\text{NOT} \cdot \text{NOT}^\dagger = I$ and $\text{Shift} \cdot \text{Shift}^\dagger = I$, the operators NOT and Shift are also unitary. The operator $\text{Shift} \cdot \text{NOT}$ is also a unitary transformation and we have:

$$\text{Shift} \cdot \text{NOT}|0\rangle = \text{Shift}|1\rangle = -|1\rangle,$$

$$\text{Shift} \cdot \text{NOT}|1\rangle = \text{Shift}|0\rangle = |0\rangle.$$

Therefore, its associated matrix is

$$R_{3\pi/2} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}.$$

The square-root of NOT (introduced by Deutsch [85]) is the transformation

$$\begin{aligned} \sqrt{\text{NOT}} : \quad |0\rangle &\rightarrow \frac{1}{2}(1+i)|0\rangle + \frac{1}{2}(1-i)|1\rangle, \\ |1\rangle &\rightarrow \frac{1}{2}(1-i)|0\rangle + \frac{1}{2}(1+i)|1\rangle, \end{aligned}$$

$$\sqrt{\text{NOT}} = \frac{1}{2} \begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix}.$$

A routine check shows that

$$\sqrt{\text{NOT}} \cdot \sqrt{\text{NOT}} = \text{NOT}, \tag{4.8}$$

and

$$\sqrt{\text{NOT}} \cdot \sqrt{\text{NOT}}^\dagger = \frac{1}{4} \begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix} \begin{pmatrix} 1-i & 1+i \\ 1+i & 1-i \end{pmatrix} = I.$$

The square-root of NOT is a typical “quantum” gate in the sense that *it is impossible to have a single-input/single-output classical binary logic gate that satisfies (4.8)*. Indeed, any classical binary

$$\sqrt{\text{NOT}}_{\text{classical}}$$

gate is going to output a 0 or a 1 for each possible input 0/1. Assume that we have such a classical square-root of NOT gate acting as a pair of transformations

$$\sqrt{\text{NOT}}_{\text{classical}}(0) = 1, \sqrt{\text{NOT}}_{\text{classical}}(1) = 0.$$

Then, two consecutive applications of it will *not* flip the input!²⁶

Finally we consider the Hadamard transformation H is defined by

$$H : \begin{aligned} |0\rangle &\rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ |1\rangle &\rightarrow \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \end{aligned}, H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

This transformation has a number of important applications. When applied to $|0\rangle$, H creates a superposition state

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle).$$

Applied to n bits individually, H generates a superposition of all 2^n possible states. To see this we need some rudiments on tensor products.

Consider two operators $A : \mathbf{C}^n \rightarrow \mathbf{C}^m$ and $B : \mathbf{C}^q \rightarrow \mathbf{C}^p$. The tensor product of A and B is the operator $A \otimes B : \mathbf{C}^n \otimes \mathbf{C}^q \rightarrow \mathbf{C}^m \otimes \mathbf{C}^p$, with the property $A \otimes B(x \otimes y) = Ax \otimes By$, for any $x \in \mathbf{C}^n$ and $y \in \mathbf{C}^q$. A convenient way is, again, to work with matrices. Let A be a $(m \times n)$ matrix and B a $(p \times q)$ matrix. The (right) Kronecker product of A and B is the $(mp \times nq)$ matrix defined as follows:

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{pmatrix}.$$

For example, if

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix},$$

are two 2×2 matrices, then we have:

$$A \otimes B = \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{pmatrix}$$

Two important mathematical results are useful:

²⁶See Williams and Clearwater [296], Chapter 4, for a detailed analysis of the square-root of NOT computation.

(a) If A and B are matrices associated to the operators A and B , then the matrix associated to $A \otimes B$ is the Kronecker product of A and B .

(b) The tensor products of two unitary transformations is also unitary.

Consequently, considering the tensor product of n single qubit transformations, we can obtain examples of unitary transformations acting on n qubits.

For instance, let $(|00\rangle, |01\rangle, |10\rangle, |11\rangle)$ be the basis in $\mathbf{C}^2 \otimes \mathbf{C}^2$ and consider the following transformations:

$$I \otimes I : \begin{array}{lcl} |00\rangle & \rightarrow & |00\rangle \\ |01\rangle & \rightarrow & |01\rangle \\ |10\rangle & \rightarrow & |10\rangle \\ |11\rangle & \rightarrow & |11\rangle \end{array}, \quad I \otimes I = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

$$I \otimes \text{NOT} : \begin{array}{lcl} |00\rangle & \rightarrow & |01\rangle \\ |01\rangle & \rightarrow & |00\rangle \\ |10\rangle & \rightarrow & |11\rangle \\ |11\rangle & \rightarrow & |10\rangle \end{array}, \quad I \otimes \text{NOT} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

$$\text{NOT} \otimes I : \begin{array}{lcl} |00\rangle & \rightarrow & |10\rangle \\ |01\rangle & \rightarrow & |11\rangle \\ |10\rangle & \rightarrow & |00\rangle \\ |11\rangle & \rightarrow & |01\rangle \end{array}, \quad \text{NOT} \otimes I = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

$$\text{NOT} \otimes \text{NOT} : \begin{array}{lcl} |00\rangle & \rightarrow & |11\rangle \\ |01\rangle & \rightarrow & |10\rangle \\ |10\rangle & \rightarrow & |01\rangle \\ |11\rangle & \rightarrow & |00\rangle \end{array}, \quad \text{NOT} \otimes \text{NOT} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}.$$

$$Shift \otimes \text{NOT} : \begin{array}{lcl} |00\rangle & \rightarrow & |01\rangle \\ |01\rangle & \rightarrow & |00\rangle \\ |10\rangle & \rightarrow & -|11\rangle \\ |11\rangle & \rightarrow & -|10\rangle \end{array}, \quad Shift \otimes \text{NOT} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

We are ready to come back to binary representations of the numbers from 0 to $2^n - 1$ via Hadamard operator. The Walsh–Hadamard transformation is defined recursively by

$$W_n = H, \text{ if } n = 1 \text{ and } W_n = H \otimes W_{n-1}, \text{ for any } n \geq 2.$$

For example, if $n = 2$ then

$$\begin{aligned} W_2|00\rangle &= (H \otimes H)|00\rangle \\ &= H|0\rangle \otimes H|0\rangle \\ &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ &= \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle), \end{aligned}$$

$$\begin{aligned} W_2|01\rangle &= (H \otimes H)|01\rangle \\ &= H|0\rangle \otimes H|1\rangle \\ &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\ &= \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle), \end{aligned}$$

$$\begin{aligned} W_2|10\rangle &= (H \otimes H)|10\rangle \\ &= H|1\rangle \otimes H|0\rangle \\ &= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ &= \frac{1}{2}(|00\rangle + |01\rangle - |10\rangle - |11\rangle), \end{aligned}$$

$$\begin{aligned} W_2|11\rangle &= (H \otimes H)|11\rangle \\ &= H|1\rangle \otimes H|1\rangle \\ &= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\ &= \frac{1}{2}(|00\rangle - |01\rangle - |10\rangle + |11\rangle), \end{aligned}$$

so the associated matrix is

$$W_2 = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}.$$

For $n = 3$, we have

$$\begin{aligned}
 W_3|000\rangle &= (H \otimes W_2)|000\rangle \\
 &= H|0\rangle \otimes W_2|00\rangle \\
 &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) \\
 &= \frac{1}{2\sqrt{2}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle \\
 &\quad + |111\rangle),
 \end{aligned}$$

$$\begin{aligned}
 W_3|001\rangle &= (H \otimes W_2)|001\rangle \\
 &= H|0\rangle \otimes W_2|01\rangle \\
 &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle) \\
 &= \frac{1}{2\sqrt{2}}(|000\rangle - |001\rangle + |010\rangle - |011\rangle + |100\rangle - |101\rangle + |110\rangle \\
 &\quad - |111\rangle),
 \end{aligned}$$

and so on. The associated matrix is $W_3 = H \otimes W_2$:

$$\begin{aligned}
 W_3 &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \\
 &= \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 \end{pmatrix}.
 \end{aligned}$$

If we apply the Walsh–Hadamard transformation to $|00\dots0\rangle$ we get a superposition of all possible states:

$$\begin{aligned}
W_n|00\dots0\rangle &= (H_2 \otimes H_2 \otimes \dots \otimes H_2)|00\dots0\rangle \\
&= \frac{1}{\sqrt{2^n}}((|0\rangle + |1\rangle) \otimes (|0\rangle + |1\rangle) \otimes \dots \otimes (|0\rangle + |1\rangle)) \\
&= \frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} |i\rangle.
\end{aligned}$$

For many quantum algorithms the state

$$|\psi\rangle = \frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} |i\rangle \quad (4.9)$$

is very convenient to be an “initial state” because it contains an equal weighted distribution of all basis states. An “empty” register can be “set” in the above state by an application of the Walsh–Hadamard transformation. In this way, *using a linear number of operations we can transform one basis state into an exponentially large, equally weighted superposition of all basis states.*

Another useful transformation on $\mathbf{C}^2 \otimes \mathbf{C}^2$ is the “controlled-NOT” gate, C_{NOT} defined as follows:

$$C_{\text{NOT}} : \begin{array}{lcl} |00\rangle & \rightarrow & |00\rangle \\ |01\rangle & \rightarrow & |01\rangle \\ |10\rangle & \rightarrow & |11\rangle \\ |11\rangle & \rightarrow & |10\rangle \end{array}, \quad C_{\text{NOT}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Given the input state $|ij\rangle$, $i, j \in \{0, 1\}$, the output state produced by C_{NOT} is $|ik\rangle$, where $k = i \oplus j \pmod{2}$. The first bit is not disturbed (it is a control bit) and the second one interchanges 0 and 1 if and only if the first bit is 1, which corresponds to the logical exclusive-OR (XOR).

The controlled-NOT gate C_{NOT} can be represented by a circuit of the form specified in Figure 4.2.

The “oplus” sign indicates the control bit; the opposite symbol indicates the conditional negation of the second bit. If the input states at a and b are in base states $|0\rangle$ or $|1\rangle$, then the output state at x is the same as the input state at a , and the output state at y is the exclusive-OR of the two input states.

The transformation C_{NOT} is unitary since $C_{\text{NOT}}^\dagger = C_{\text{NOT}}$ and $C_{\text{NOT}}^2 = I_4$ (the 4×4 identity matrix). On the other hand,

C_{NOT} cannot be written as a tensor product of two operators.

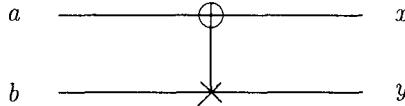


Figure 4.2: The controlled-NOT gate.

Indeed, assume the contrary, and take two operators A, B such that $C_{\text{NOT}} = A \otimes B$. Assume that the associated matrices are

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

so, the matrix $A \otimes B$ corresponds to C_{NOT} and we have

$$\begin{aligned} A \otimes B &= \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{pmatrix} \\ &= C_{\text{NOT}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \end{aligned}$$

Since $a_{11}b_{11} = 1$ and $a_{11}b_{12} = 0$ it follows that $a_{11} \neq 0$ and $b_{12} = 0$, which is impossible because $a_{22}b_{12} = 1$.

Similarly, one can define the “controlled-controlled-NOT” transformation, CC_{NOT} , operating on three qubits, which negates the rightmost bit if and only if the first two are both 1:

$$CC_{\text{NOT}} : \begin{array}{l} |000\rangle \rightarrow |000\rangle \\ |001\rangle \rightarrow |001\rangle \\ |010\rangle \rightarrow |010\rangle \\ |011\rangle \rightarrow |011\rangle \\ |100\rangle \rightarrow |100\rangle \\ |101\rangle \rightarrow |101\rangle \\ |110\rangle \rightarrow |111\rangle \\ |111\rangle \rightarrow |110\rangle \end{array}, \quad CC_{\text{NOT}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

We have $CC_{\text{NOT}} \cdot CC_{\text{NOT}}^\dagger = CC_{\text{NOT}}^\dagger \cdot CC_{\text{NOT}} = I_8$, hence CC_{NOT} is also unitary.

4.10 No Cloning Theorem

Quantum states cannot be cloned, as Wootters and Zurek [301], and Dieks [89] have proved as an application of the linearity of unitary transformations. It is not possible to create the state $(a|0\rangle + b|1\rangle) \otimes (a|0\rangle + b|1\rangle)$ from an *unknown state* $a|0\rangle + b|1\rangle$.

In other words, there is no unitary transformation U such that $U|\varphi 0\rangle = |\varphi\varphi\rangle$ for all quantum states $|\varphi\rangle$.

Indeed, assume the contrary and let $|\varphi\rangle$ and $|\psi\rangle$ be two orthogonal vectors in \mathbf{C}^2 and take $|x\rangle = \frac{1}{\sqrt{2}}(|\varphi\rangle + |\psi\rangle)$. Then, $U|\varphi 0\rangle = |\varphi\varphi\rangle$ and $U|\psi 0\rangle = |\psi\psi\rangle$. On the one hand,

$$\begin{aligned} U|x 0\rangle &= |xx\rangle \\ &= \frac{1}{\sqrt{2}}(|\varphi\rangle + |\psi\rangle) \otimes \frac{1}{\sqrt{2}}(|\varphi\rangle + |\psi\rangle) \\ &= \frac{1}{2}(|\varphi\varphi\rangle + |\varphi\psi\rangle + |\psi\varphi\rangle + |\psi\psi\rangle). \end{aligned}$$

On the other hand,

$$\begin{aligned} U|x 0\rangle &= U\left(\frac{1}{\sqrt{2}}(|\varphi 0\rangle + |\psi 0\rangle)\right) \\ &= \frac{1}{\sqrt{2}}(U|\varphi 0\rangle + U|\psi 0\rangle) \\ &= \frac{1}{\sqrt{2}}(|\varphi\varphi\rangle + |\psi\psi\rangle). \end{aligned}$$

Since the vectors φ and ψ are orthogonal, the vectors $|\varphi\varphi\rangle$, $|\varphi\psi\rangle$, $|\psi\varphi\rangle$, $|\psi\psi\rangle$ constitute a basis in $\mathbf{C}^2 \otimes \mathbf{C}^2$ and the vector $|xx\rangle = U|x 0\rangle$ has been written in two different ways as a linear combination of this basis vectors, an impossibility.

It's important to understand that the no cloning principle states the impossibility of reliably cloning an *unknown* quantum state: it is possible to clone a *known* quantum state. It is possible to obtain n particles in an entangled state $a|00\dots 0\rangle + b|11\dots\rangle$ from an unknown state $a|0\rangle + b|1\rangle$. Each particle will behave in exactly the same way when measured with respect to the basis $\{|00\dots 0\rangle, |00\dots 01\rangle, \dots, |11\dots 1\rangle\}$, but *not* when measured with respect to other bases. It is not possible to create the n particle state

$$(a|0\rangle + b|1\rangle) \otimes (a|0\rangle + b|1\rangle) \otimes \dots (a|0\rangle + b|1\rangle)$$

from an unkown state $a|0\rangle + b|1\rangle$, cf. Rieffel and Polak [250].

In a sense, the no cloning principle seems to announce “bad news”: we lose one of the most important facilities of classical computation, the unlimited possibility to copy. There is “good news” derived from this principle, for example, the possibility of unconditional secure key generation (see Section 6.2 in Gruska [119]). New techniques (see, for example, Bužek, Braunstein, Hillery, Bruß, [40]) open possibilities to produce “approximate” copies of qubits: imperfect, but very close to real copies of qubits can be produced with a “quality” not depending upon the qubits to be copied. Of course, there is a price to be paid: copies produced in this way are entangled.

4.11 Measurements

As we have already seen, the measurement of one or more particles in a quantum system results in a projection of the state of the system prior to measurement onto the subspace of the state space compatible with the measured values. The amplitude of the projection is rescaled to make sure that the resulting state vector has length one. The probability that the result of the measurement is a given value is the sum of the squares of the absolute values of the amplitudes of all components compatible with that value of the measurement.

A simple example of measurement in a two qubit system will illustrate the above points. Let’s fix the basis $\{|0\rangle, |1\rangle\}$, and assume that all measurements of individual qubits will be done with respect to this basis. An arbitrary state of a two qubit system can be written as

$$a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle,$$

where a, b, c and d are complex numbers such that

$$|a|^2 + |b|^2 + |c|^2 + |d|^2 = 1.$$

When the first qubit is measured, then the probability that the result is $|0\rangle$ is $|a|^2 + |b|^2$. Assume now that the measurement gives the first qubit exactly that value, that is, $|0\rangle$. Consequently, the state is projected onto the subspace compatible with the measurement which is the subspace spanned by $|00\rangle$ and $|01\rangle$ and the result of this projection is $a|00\rangle + b|01\rangle$. Renormalizing we get:

$$\frac{1}{\sqrt{|a|^2 + |b|^2}} \cdot (a|00\rangle + b|01\rangle).$$

In general, consider a system containing n qubits ($n \geq 2$). Any state $|x\rangle$ of the system can be expressed as

$$\sum_{i_1, i_2, \dots, i_n=0,1} c_{i_1 i_2 \dots i_n} |i_1 i_2 \dots i_n\rangle,$$

where

$$\sum_{i_1, i_2, \dots, i_n=0,1} |c_{i_1 i_2 \dots i_n}|^2 = 1.$$

When the first qubit is measured with respect to the basis $\{|0\rangle, |1\rangle\}$, then the result $|0\rangle$ is obtained with probability

$$P = \sum_{i_2, \dots, i_n=0,1} |0i_2 \dots i_n|^2.^{27}$$

After rescaling, the new state obtained after the measurement is

$$\frac{1}{\sqrt{\sum_{i_2, \dots, i_n=0,1} |c_{0i_2 \dots i_n}|^2}} \cdot \left(\sum_{i_2, \dots, i_n=0,1} c_{0i_2 \dots i_n} |0i_2 \dots i_n\rangle \right).$$

Similarly, the measurement gives the outcome $|1\rangle$ with the probability

$$1 - P = \sum_{i_2, \dots, i_n=0,1} |c_{1i_2 \dots i_n}|^2,$$

and the state changes correspondingly.

What is the price of measurement? According to Landauer [159],

If it [measurement] is simply information transfer, that is done all the time inside the computer, and can be done with arbitrarily little dissipation.

There are many speculations about the “collapse of the wave function (state)” due to an irreversible interaction of the microphysical quantum system with the macroscopic measurement apparatus. Some authors (see, for example, Greenberg and YaSin [117] or Herzog, Kwiat, Weinfuter and Zeilinger [132]) have argued that it is, in fact, possible to reconstruct the state of the physical system before the measurement, that is, to “reverse the collapse of the wave function” if the process of measurement is reversible. After “reconstruction” no information about the measurement is left.

The act of measurement gives another perspective about entangled particles. Particles are not entangled if the measurement of one has no effect on the other. For instance, the state

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

is entangled since the probability that the first bit is measured to be $|0\rangle$ is $1/2$ if the second bit has not been measured. However, if the second bit had

²⁷We used the projection onto the space spanned by $\{|0i_2 \dots i_n\rangle | i_k \in \{0, 1\}, 2 \leq k \leq n\}$.

been measured, then the probability that the first bit is measured as $|0\rangle$ is different from $1/2$, it is either 1 or 0 , depending on whether the second bit was measured as $|0\rangle$ or $|1\rangle$, respectively. Hence, the probability of measuring the first bit has been changed by the measurement of the second bit.

In contrast, the state

$$\frac{1}{\sqrt{2}}(|00\rangle + |01\rangle) = |0\rangle \otimes \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

is not entangled. Reason: any measure of the first qubit will produce the result $|0\rangle$ independently whether a measurement is performed or not on the second qubit, and the second qubit has probability $\frac{1}{2}$ to be measured to $|0\rangle$ regardless of whether the first qubit was measured or not.

In a sense, entangled states can be equivalently presented in mathematical terms (they cannot be represented as a tensor product of two states) or in physical terms (the measurement on one affects the other); however, the physical meaning is richer than the mathematical formalism.

An important consequence of the existence of entangled states is the fact that if a quantum memory register exists in an entangled state, one can change the state of one part of the register simply by measuring another part of it. This is a unique feature of quantum physics²⁸ which has no parallel in classical physics. *Entanglement is one of the most important features which distinguishes Quantum from Conventional Computing.*

4.12 Zeno Machines

A Zeno machine is a Turing machine which computes with “increased speed”. Two time scales act simultaneously: the *intrinsic time scale of the process of computation* approaches the infinity in a finite *extrinsic (or proper) time of some outside observer*, cf. Svozil [284]. As a consequence, certain uncomputable functions (i.e. functions which cannot be computed by any Turing machine) become Zeno computable. For example, the halting problem – the most notorious unsolvable problem in classical computation theory (see, for example, Odifreddi [198]) is Zeno solvable.

Zeno machines have been introduced by Weyl [290] (see Svozil [284] for a bibliography on this subject). Already Weyl raised the question whether it is kinematically feasible for a machine to carry out an infinite sequence of operations in a finite time. He wrote [290], p. 42:

Yet, if the segment of length 1 really consists of infinitely many subsegments of length $1/2, 1/4, 1/8, \dots$, as of ‘chopped-off’ wholes, then it is incompatible with the character of the infinite

²⁸Which is crucial in many quantum algorithms, teleportation, information transmission, etc.

as the ‘incompletable’ that Achilles should have been able to traverse them all. If one admits this possibility, then there is no reason why a machine should not be capable of completing an infinite sequence of distinct acts of decision within a finite amount of time; say, by supplying the first result after $1/2$ minute, the second after another $1/4$ minute, the third $1/8$ minute later than the second, etc. In this way it would be possible, provided the receptive power of the brain would function similarly, to achieve a traversal of all natural numbers and thereby a sure yes-or-no decision regarding any existential question about natural numbers!

A possible construction of a Zeno machine starts with a normal Turing machine and considers two time scales, τ and t as follows:

- The *proper time* τ measures the physical system time by clocks in an usual way.
- A discrete cycle time $t = 0, 1, 2, \dots$ characterizes an “intrinsic” time scale for a process running on the machine.
- For some unspecified reason we assume that the machine allows us to “squeeze” its intrinsic time t with respect to the proper time τ by a geometric progression. For $k < 1$ we let any time cycle of t , if measured in terms of τ , to be “squeezed” by a factor of k with respect to the foregoing time cycle. More precisely,

$$\tau_0 = 0, \tau_1 = k, \tau_{t+1} - \tau_t = k(\tau_t - \tau_{t-1}),$$

that is

$$\tau_t = \frac{k(k^t - 1)}{k - 1}.$$

In the limit when t approaches the infinity, the proper time τ_∞ approaches $k/(1-k)$, so it *remains finite*.

There is no commonly accepted classical physical principle which would, *a priori*, forbid such a behaviour.²⁹ One might argue that such an “oracle” would require a geometric energy increase resulting in an infinite consumption of energy. Yet, no currently accepted classical physical principle excludes us from assuming that every geometric decrease in cycle time could be associated with a geometric progression in energy consumption, at least up to some limiting (e.g. Planck) scale.

So, classical physics doesn’t forbid the existence of Zeno machines. However, *classical logic does*. A simple diagonalization argument, which mimics the undecidability of the halting problem, shows that Zeno machines are *logically impossible*. Consider an arbitrary algorithm $B(x)$ whose input is a

²⁹Classical mechanics postulates space and time continua as a foundational principle.

binary string x . Assume, for the sake of a contradiction, that there exists an effective halting algorithm HALT , implementable on a Zeno machine, which is able to decide whether B eventually stops on x or not. Using $\text{HALT}(B(x))$ we shall construct another Zeno machine A , which has as an input a program B and which proceeds as follows: Upon reading the program B as an input, A makes a copy of it.³⁰ In the next step, our machine uses the code $\#(B)$ as an input string for B *itself*, that is, A forms $B(\#(B))$, henceforth denoted by $B(B)$. The machine hands $B(B)$ over to its subroutine HALT . Then, A proceeds as follows:

- if $\text{HALT}(B(B))$ decides that $B(B)$ eventually halts, then A does not halt,³¹
- if $\text{HALT}(B(B))$ decides that $B(B)$ never halts, then A halts.

What about using A on its own code as input? Notice that B is arbitrary, so there is no restriction to prevent us for doing this! Consequently, A , which is representable by its code $\#(A)$ will be applied to itself.

Assume that classically A is restricted to classical bits of information. Then, whenever $A(A)$ halts, $\text{HALT}(A(A))$ forces $A(A)$ not to halt, and conversely, whenever $A(A)$ does not halt, then $\text{HALT}(A(A))$ steers $A(A)$ into the halting state. In both cases one arrives at a contradiction, therefore, Zeno machines are logically inconsistent.

What about the case when A is allowed a qubit of information. Assume that $|0\rangle$ and $|1\rangle$ are the halting and nonhalting states, respectively. The computation can be performed if A receives as an input a qubit corresponding to the fixed point state $|\star\rangle$ of the NOT operator³²:

$$\text{NOT} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},$$

$$\text{NOT}|\star\rangle = |\star\rangle.$$

A simple computation shows that

$$|\star\rangle = \left| \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}} \right\rangle.$$

The qubit solution $|\star\rangle$ proves the impossibility of A to control the output as the probability to reach a halting (nonhalting) state is exactly one half. At the level of probability amplitudes, quantum theory permits Zeno machines, but at the level of observable probabilities, this super-power is nullified, as the result of the computation appears to be random.

³⁰This can be readily achieved, since the program B is presented to A in some encoded form $\#(B)$, i.e. as a string of symbols.

³¹This can be realized by an infinite loop.

³²Diagonalization operator.

4.13 Inexhaustible Uncertainty

In 1927 Werner Heisenberg discovered a fundamental limitation of quantum mechanics: a bound on the accuracies with which certain complementary pairs of observables can be measured. The “canonical” understanding of complementarity is expressed in Messiah [190, p. 154]

The description of properties of microscopic objects in classical terms requires pairs of complementary variables; the accuracy in one member of the pair cannot be improved without a corresponding loss in the accuracy of the other member.

:

*It is impossible to perform measurements of position x and momentum p with uncertainties (defined by the root-mean square deviations) Δx and Δp such that the product of $\Delta x \Delta p$ is smaller than a constant unit of action $\frac{\hbar}{2}$.*³³

In Prigogine’s words [244, p. 51],

the world is richer than it is possible to express in any single language.

Next we will follow Moore [196] to illustrate uncertainty using simple automata “Gedanken” experiments (see more in Conway [65], Brauer [38], Svozil [282], Calude [56]).

A (simple) Moore experiment can be described as follows: a copy of the machine will be experimentally observed, i.e. the experimenter will input a string of input symbols to the machine and will observe the sequence of output symbols. The correspondence between input and output symbols depends on the particular chosen machine and on its initial state. The experimenter will study the sequences of input and output symbols and will try to conclude that “the machine being experimented on was in state q at the beginning of the experiment”.³⁴

In what follows we will work with finite deterministic automata with a finite set K of states, an input alphabet V , and a transition function δ :

³³Many other instances of complementarity are well known. A simple example is offered by the so-called two-slit experiment. A source is “shooting” electrons towards a wall which has two tiny holes (slits), each of them just enough for one electron to get through at a time. A second wall has a detector, that can be moved up and down, with the aim to count the number of electrons reaching a given position of the second wall. Experimentally one can determine the probabilities that electrons reach some positions on the second wall, depending upon the number of open slits, one or two. Contrary to common intuition, due to interference, there are places where one counts fewer electrons in the case when both slits are open than in the case when only one slit is open! Detecting through which slit an electron went (a particle measurement) or recording the interference pattern (a wave measurement) cannot be done in the same experiment.

³⁴This is often referred to as a *state identification experiment*.

$K \times V \rightarrow K$. Instead of final states we will consider an output function $f : K \rightarrow \{0, 1\}$. At each time the automaton is in a given state q and is continuously emitting the output $f(q)$. The automaton remains in state q until it receives an input signal σ , when it assumes the state $\delta(q, \sigma)$ and starts emitting $f(\delta(q, \sigma))$. As we will discuss only the simplest case when the alphabet $V = \{0, 1\}$, an automaton will be just a triple $M = (K, \delta, f)$.

The transition function δ can be extended to a function $\bar{\delta} : K \times V^* \rightarrow K$, as follows: $\bar{\delta}(q, \lambda) = q$, for all $q \in K$, and $\bar{\delta}(q, \sigma w) = \bar{\delta}(\delta(q, \sigma), w)$, for all $q \in K, \sigma \in V, w \in V^*$.

The output produced by an experiment started in state q with input sequence $w \in V^*$ is described by $E(q, w)$, where E is the function $E : Q \times V^* \rightarrow V^*$ defined by the following equations:

$$E(q, \lambda) = f(q),$$

$$E(q, \sigma w) = f(q)E(\delta(q, \sigma), w),$$

for all $q \in K, \sigma \in V, w \in V^*$; recall that $f : K \rightarrow V$ is the output function.

Consider, for example, Moore's automaton, in which $K = \{1, 2, 3, 4\}$, the transition is given by the following tables:

q	σ	$\delta(q, \sigma)$	q	σ	$\delta(q, \sigma)$
1	0	4	3	0	4
1	1	3	3	1	4
2	0	1	4	0	2
2	1	3	4	1	2

Table 4.3: Moore's automaton transition.

and the output function is defined by $f(1) = f(2) = f(3) = 0, f(4) = 1$. A graphical display appears in Figure 4.13.

The experiment starting in state 1 with input sequence 000100010 leads to the output 0100010001. Indeed,

$$\begin{aligned} E(1, 000100010) &= f(1)f(4)f(2)f(1)f(3)f(4)f(2)f(1)f(3)f(4) \\ &= 0100010001. \end{aligned}$$

Consider now an automaton $M = (K, \delta, f)$, and following Moore [196] say that a state q is “indistinguishable” from a state q' (with respect to M) if every experiment performed on M starting in state q produces the same outcome as it would starting in state q' . Formally, $E(q, x) = E(q', x)$, for all strings $x \in V^+$.

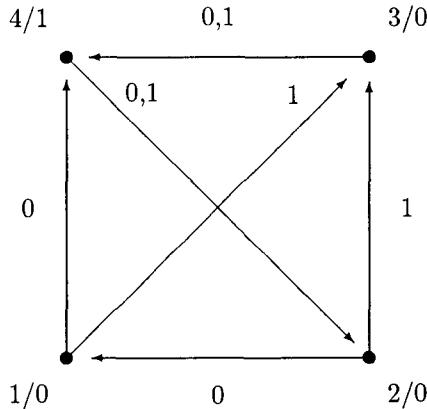


Figure 4.3: Moore's automaton.

An equivalent way to express the indistinguishability of the states q and q' is to require, following Conway [65, p. 3], that for all $w \in V^*$,

$$f(\bar{\delta}(q, w)) = f(\bar{\delta}(q', w)).$$

Indeed,

$$E(q, x_1 x_2 \dots x_n) = f(q)f(\bar{\delta}(q, x_1))f(\bar{\delta}(q, x_1 x_2)) \cdots f(\bar{\delta}(q, x_1 x_2 \dots x_n)),$$

for all $q \in K, x_1 x_2 \dots x_n \in V^*$.

A pair of states will be said to be “distinguishable” if they are not “indistinguishable”, i.e. if there exists a string $x \in V^+$, such that $E(q, x) \neq E(q', x)$.

Moore [196] has proven the following important theorem:

There exists an automaton M such that any pair of its distinct states are distinguishable, but there is no experiment which can determine what state the machine was in at the beginning of the experiment.

Moore used the automaton displayed in Figure 4.13 and his argument is simple. Indeed, each pair of distinct states can be distinguished by an experiment: 1, 2 by $x = 0$, 1, 3 by $x = 1$, 1, 4 by $x = 0$, 2, 3 by $x = 0$, 2, 4 by $x = 0$, and 3, 4 by $x = 0$.

However, there is no (unique) experiment capable to distinguish between every pair of arbitrary distinct states. Two cases have to be examined:

(A) *The experiment starts with 1, i.e. $x = 1u$, $u \in V^*$.* In this case $E(1, x) = E(2, x)$, that is x cannot distinguish between the states 1, 2 as

$$\begin{aligned} E(1, x) = E(1, 1u) &= f(1)f(\delta(1, 1))E(\delta(1, 1), u) \\ &= f(1)f(3)E(3, u) = 00E(3, u), \end{aligned}$$

and

$$\begin{aligned} E(2, x) = E(2, 1u) &= f(2)f(\delta(2, 1))E(\delta(2, 1), u) \\ &= f(2)f(3)E(3, u) = 00E(3, u). \end{aligned}$$

(B) *The experiment starts with 0, i.e. $x = 0v$, $v \in V^*$.* In this case

$$E(1, x) = E(2, x),$$

that is x cannot distinguish between the states 1, 3 as

$$\begin{aligned} E(1, x) = E(1, 0v) &= f(1)f(\delta(1, 0))E(\delta(1, 0), v) \\ &= f(1)f(4)E(4, v) = 01E(4, v), \end{aligned}$$

and

$$\begin{aligned} E(2, x) = E(3, 0v) &= f(3)f(\delta(3, 0))E(\delta(3, 0), v) \\ &= f(3)f(4)E(4, v) = 01E(4, v). \end{aligned}$$

Moore's result can be thought of as being a *discrete analogue* of the Heisenberg uncertainty principle. The state of an electron E is considered specified if both its velocity and its position are known. Experiments can be performed with the aim of answering either of the following:

1. What was the position of E at the beginning of the experiment?
2. What was the velocity of E at the beginning of the experiment?

For an automaton, experiments can be performed with the aim of answering either of the following:

1. Was the automaton in state 1 at the beginning of the experiment?
2. Was the automaton in state 2 at the beginning of the experiment?

In either case, performing the experiment to answer question 1 changes the state of the system, so that the answer to question 2 cannot be obtained. This means that it is only possible to gain partial information about the previous history of the system, since performing experiments causes the system to "forget" about its past.

An exact quantum mechanical analogue has been given by Foulis and Randall [100, Example III]: Consider a device which, from time to time, emits

a particle and projects it along a linear scale. We perform two experiments. In experiment A, the observer determines if there is a particle present. If there is no particle, the observer records the outcome of A as the outcome {4}. If there is a particle, the observer measures its position coordinate x . If $x \geq 1$, the observer records the outcome {2}, otherwise {3}. A similar procedure applies for experiment B: If there is no particle, the observer records the outcome of B as {4}. If there is, the observer measures the x -component p_x of the particle's momentum. If $p_x \geq 1$, the observer records the outcome {1, 2}, otherwise the outcome {1, 3}.³⁵

Moore's automaton is a simple model featuring an “uncertainty principle” (cf. Conway [65, p. 21]), later termed “computational complementarity” by Finkelstein and Finkelstein [99]; for a detailed analysis see Svozil [282], Calude, Calude, Svozil and Yu [43], Calude and Lipponen [53], Jurvanen and Lipponen [145].

4.14 Randomness

Randomness is at the very heart of quantum physics. When a physical state that is in a superposition of states is measured, then it collapses into one of its possible states in a completely unpredictable way – we can only evaluate the probability of obtaining various possible outcomes. An extreme view is to claim with Peres [236] that

in a strict sense quantum theory is a set of rules allowing the computation of probabilities for the outcomes of tests which follow specific preparations.

According to Milburn [193], p. 1, a quantum principle is

physical reality is irreducible random.

We are talking about “true” randomness, not the “randomness” which, at times, nature appears to exhibit and for which classical physics blames our ignorance: meteorologists cannot predict accurately the path of a hurricane,³⁶ for example.

A mathematical definition of randomness is provided by algorithmic information theory, see Chaitin [59, 60], Calude [42]. The idea is to define (algorithmic) randomness as incompressibility. The length of the smallest program (say, for a universal Turing machine³⁷) generating a binary string

³⁵Another quantum mechanical analogue has been proposed by Giuntini [114], pp. 159–162.

³⁶The explanation is not difficult to obtain: the equations governing the motion of the atmosphere are nonlinear and tiny errors in the initial conditions can immensely amplify. This behaviour is known as “deterministic chaos”.

³⁷For technical reasons, we use self-delimiting Turing machines, machines having a “prefix-free” domain: no proper extension of a program that eventually halts has that property.

is the *program-size complexity* of the string. This idea can be extended in an appropriate way to infinite sequences. A random string/sequence is incompressible as the smallest program for generating it is the string/sequence itself! Strings/sequences that can be generated by small programs are deemed to be less random than those requiring longer programs. For example, the digits of $\pi(3.1415926\ldots)$ can be computed one by one; nonetheless, if examined locally, without being aware of their provenance, they appear “random”. People have calculated π out to a billion or more digits. A reason for doing this is the question of whether each digit occurs the same number of times, a symptom of randomness. It seems, but remains unproven, that the digits 0 through 9 each occur 10% of the time in a decimal expansion of π . If this turns out to be true, then π would be a so-called simply normal real number. But although π may be random in so far as it’s “normal”, it is far from (algorithmic) random, because its infinity of digits can be compressed into a concise program for calculating them. The numbers generated by the so-called logistic map,

$$x_{n+1} = rx_n(1 - x_n), \quad (4.10)$$

where r is an arbitrary constant and the process starts at some state $x_0 = c$, may appear “random” for some values, say $x_0 = 0.1$ and $r = 3.98$; however, they are not, because of the succinctness of the rule (4.10) describing them. In general, a long string of pseudo-random bits produced by a program may pass all practical statistical tests for randomness, but it is not (algorithmic) random: its program-size complexity is bounded by the size of the generating program plus a few extra bits which specify the random number seed.

Similarly, a long string of binary bits produced by any classical physical system, of which a Turing machine or a Java program is just an instance, is not (algorithmic) random. The program-size complexity of such a string is bounded by the size of the program generating it, that is, the physical law which governs its evolution, plus the size of the initial conditions on which the law acts. Any classical computer can only feign randomness; thinking otherwise is not only wrong, but as von Neumann said,

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.

Note that human beings are not doing a better job in generating “random” bits as Shannon [268] has argued.³⁸

Is there any physical system that can generate arbitrarily long (algorithmic) random strings?

It is not difficult to destroy randomness. For example, start with a random sequence $x_1x_2\ldots x_n\ldots$ over the alphabet $\{0, 1\}$ and define a new sequence $y_1y_2\ldots y_n\ldots$, over the alphabet $\{0, 1, 2\}$, by

$$y_1 = x_1, y_n = x_{n-1} + x_n, \quad n \geq 2.$$

³⁸Biases observed in people’s preferences for popular lottery numbers are manifest.

Then, the new sequence *is not random*. The motivation is simple: the strings 02 and 20 (and, infinitely many more others) never appear, so the sequence has clear regularities (which can, actually, be detected by simple statistical randomness tests).

It is much more demanding to “generate” a truly random long string starting from an initial state with a simple description. Note that the condition of simplicity of the initial state is *crucial*: starting from a random string one can generate, in a pure algorithmic way, many other random strings. For example, if $x_1x_2 \dots x_{2n-1}x_{2n}$ is a random binary string, then break the string into pairs and then code 00, 01, 10, 11 by a, b, c, d : the result is again a random sequence. So, the problem is to start from an initial state which can be precisely controlled and has a low program-size complexity and produce measurements of unbounded program-size complexity out its natural dynamical evolution.

Quantum mechanics seems capable to produce, with probability one, truly random strings. Here is a way to do it. Consider the operator

$$R_\theta = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix},$$

and recall that it rotates a qubit $a|0\rangle + b|1\rangle$ through an angle θ . In particular, $R_{\frac{\pi}{4}}$ transforms that state $|0\rangle$ into an equally weighted superposition of 0 and 1:

$$\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle. \quad (4.11)$$

So, to make a quantum device to produce random bits one needs to place a 2-state quantum system in the $|0\rangle$ state, apply the operator $R_{\frac{\pi}{4}}$ to rotate the state into the superposition (4.11), and the observe the superposition. The act of observation produces the collapse into either $|0\rangle$ or $|1\rangle$, with equal chances. Consequently, one can use the quantum superposition and indeterminism to simulate, with probability one, a “fair” coin toss. Random digits produced with quantum random generators of the type described above are, with probability one, free of subtle correlations that haunt classical pseudo-random number generators. Of course, the problem of producing algorithmic random strings is still open. Indeed, let’s assume that we have a classical silicon computer that simulates, using a high-quality pseudo-random generator, the quantum mechanics dynamics and quantum measurement of a 2-state quantum system. The simulated world will be statistically almost identical (up to some degree) with the “real” quantum system. However, all simulated bits will be, in the long run, highly compressible. How can we be sure that the “real” quantum system is not just a superpowerful pseudo-random generator?

4.15 The EPR Conundrum and Bell’s Theorem

According to the philosophical view called realism, *reality* exists and has definite properties irrespective of whether they are observed by some agent. Motivated by this view point, Einstein, Podolsky and Rosen [91] suggested a classical argument to “show” that quantum mechanics is incomplete. EPR assumed (a) the non-existence of action-at-a-distance, (b) that some of the statistical predictions of quantum mechanics are correct, and (c) a reasonable criterion defining the existence of “an element of physical reality”.³⁹ They considered a system of two spatially separated but quantum mechanically correlated particles. A “mysterious” feature appears: By counterfactual reasoning, quantum mechanical experiments yield outcomes which cannot be predicted by quantum theory; hence the quantum mechanical description of the system is incomplete!

One possibility to complete the quantum mechanical description is to postulate additional “hidden-variables” in the hope that completeness, determinism and causality will be thus restored. But then, another conundrum occurs: Using basically the same postulates as those of EPR, Bell [20, 21] showed that no deterministic local hidden-variables theory can reproduce all statistical predictions of quantum mechanics.

We will present first Mermin’s [187, 188] two simple devices that explain EPR conundrum (see [44] for an automata-theoretic analysis). Later, we will concentrate on Bell’s result.

4.15.1 Mermin’s EPR Device

Mermin’s EPR device [187] has three “completely unconnected”⁴⁰ parts, two detectors (D1) and (D2) and a source (S) emitting particles. The source is placed between the detectors: whenever a button is pushed on (S), shortly thereafter two particles emerge, moving off toward detectors (D1) and (D2). Each detector has a switch that can be set in one of three possible positions – labelled 1,2,3 – and a bulb that can flash a red (*R*) or a green (*G*) light. The purpose of lights is to “communicate” information to the observer. Each detector flashes either red or green whenever a particle reaches it. Because of the lack of any relevant connections between any parts of the device, the link between the emission of particles by (S), i.e. as a result of pressing a button, and the subsequent flashing of detectors, can only be provided by the passage of particles from (S) to (D1) and (D2). Additional tools can be used to check and confirm the lack of any communication, cf. [187], p. 941.

³⁹If, without in any way disturbing a system, we can predict with certainty (i.e. with probability equal to unity) the value of a physical quantity, then there exists an element of physical reality corresponding to this physica quantity. See [91], p. 777.

⁴⁰There are no relevant connections, neither mechanical nor electromagnetic.

The device is repeatedly operated as follows:

1. the switch of either detector (D1) and (D2) is set randomly to 1 or 2 or 3, i.e. the settings or states 11, 12, 13, 21, 22, 23, 31, 32, 33 are equally likely,
2. pushing a button on (S) determines the emission toward both (D1) and (D2),
3. sometime later, (D1) and (D2) flash one of their lights, *G* or *R*,
4. every run is recorded in the form *ijXY*, meaning that (D1) was set to state *i* and flashed *X* and (D2) was set to *j* and flashed *Y*.

For example, the record 31GR means “(D1) was set to 3 and flashed *G* and (D2) was set to 1 and flashed *R*”.

Long recorded runs show the following pattern:

- (a) For records starting with *ii*, i.e. 11, 22, 33, both (D1) and (D2) flash the same colours, *RR*, *GG*, with equal frequency; *RG* and *GR* are never flashed.
- (b) For records starting with *ij*, *i* ≠ *j*, i.e. 12, 13, 21, 23, 31, 32, both (D1) and (D2) flash the same colour only 1/4 of the time (*RR* and *GG* come with equal frequencies); the other 3/4 of the time, they flash different colours (*RG*, *GR*), occurring again with equal frequencies.

Of course, the above patterns are statistical, that is they are subject to usual fluctuations expected in every statistical prediction: patterns are more and more “visible” as the number of runs becomes larger and larger.

The conundrum posed by the existence of Mermin’s device reveals as soon as we notice that the seemingly simplest physical explanation of the pattern (a) is incompatible with pattern (b). Indeed, as (D1) and (D2) are unconnected there is no way for one detector to “know”, at any time, the state of the other detector or which colour the other is flashing. Consequently, it seems plausible to assume that the colour flashed by detectors is determined only by some property, or group of properties of particles, say speed, size, shape, etc. What properties determine the colour does not really matter; only the fact that each particle carries a “program” which determines which colour a detector will flash in some state is important. So, we are led to the following two hypotheses:

H1 *Particles are classified into eight categories:*

$$GGG, GGR, GRG, GRR, RGG, RGR, RRG, RRR.^{41}$$

⁴¹A particle of type *XYZ* will cause a detector in state 1 to flash *X*; a detector in state 2 will flash *Y* and a detector in state 3 will flash *Z*.

H2 *Two particles produced in a given run carry identical programs.*

According to H1–H2, if particles produced in a run are of type *RGR*, then both detectors will flash *R* in states 1 and 3; they will flash *G* if both are in state 2. Detectors flash the same colours when being in the same states because *particles carry the same programs*.

It is clear that from H1–H2 it follows that *programs carried by particles do not depend in any way on the specific states of detectors*: they are properties of particles not of detectors. Consequently, both particles carry the same program whether or not detectors (D1) and (D2) are in the same states.⁴²

We are ready to argue that

- [L] For each type of particle, *in runs of type (b) both detectors flash the same colour at least one third of the time.*

If both particles are of types *GGG* or *RRR*, then detectors will flash the same colour all the time. For particles carrying programs containing one colour appearing once and the other colour appearing twice, only in two cases out of six possible combinations both detectors will flash the same light. For example, for particles of type *RGR*, both detectors will flash *R* if (D1) is in state 1 and (D2) is in state 3 and vice versa. In all remaining cases detectors will flash different lights. The argument remains the same for all combinations as the conclusion was solely based on the fact that one colour appears once and the other twice. So, the lights are the same one third of the time.

The conundrum reveals as a significant difference appears between the data dictated by particle programs (colours agree at least one third of the time) and the quantum mechanical prediction (colours agree only one quarter of the time):

under H1–H2, the observed pattern (b) is incompatible with [L].

4.15.2 Mermin's GHZ Device

Based on Greenberg, Horne and Zeilinger's [116] version of EPR experiment, Mermin [188] imagined a new device, let's call it GHZ, to show quantum nonlocality. The device has a source and three widely separated detectors (A), (B), (C), each of which has only two switch settings, 1 and 2. Any detector, when triggered, flashes red (*R*) or green (*G*). Again, detectors are

⁴²The emitting source (S) has no knowledge about the states of (D1) and (D2) and there is no communication among any parts of the device.

supposed to be far away from the source and there are no connections between the source and detectors (except those induced by a group of particles flying from the source to each detector).

The experiment runs as follows. Each detector is in a randomly chosen state (1 or 2) and then by pressing a button at the source a trio of particles are released towards detectors; each particle will reach a detector and, consequently, each detector will flash a light, green or red. There are eight possible states, but for the argument we need to take into consideration only those for which the number of 1's is odd, i.e. 111, 122, 212, 221.

According to [116], (a) if one detector is set to 1 (and the others to 2), then an *odd* number of red lights always flash, i.e. RRR, RGG, GRG, GGR , and they are equally likely, (b) if all detectors are set to 1, then an *odd* number of red lights is *never* flashed: GRR, RGR, RRG, GGG .

It is immediate that in case (a) knowing the colour flashed by two detectors, say (A) and (B), *determines uniquely* the colour flashed by the third detector, (C). The explanation can come only because particles are emitted by the same source (there are no connections between detectors). A similar conclusion as in the case of EPR device reveals: *particles carry programs instructing their detectors what colour to flash*. Any particle carries a program of the form XY telling its detector to flash colour X if in state 1 and colour Y if in state 2. There are four types of programs: GG, GR, RG, RR . A run in which programs carried by the trio of particles are of types (RG, GR, GG) will result in RRG if the states were 122, in GGG if the states were 212, and in GRG if the states were 221. This is an *illegal* set of programs as the number of R 's is not odd (in RRG , for example). A *legal* set of programs is (RG, GR, GR) as it produces RRR, GGR, GRG on 122, 212, 221. There are eight legal programs,

$$(RR, RR, RR), (RR, GG, GG), (GG, RR, GG), (GG, GG, RR),$$

$$(RG, GR, GR), (RG, RG, RG), (GR, GR, RG), (GR, RG, GR)$$

out of 64 possible programs.

The conundrum reveals again as none of the above programs respects (b), i.e. it is compatible with the case 111. *A single 111 run suffices to prove inconsistency! Particle programs require an odd number of R's to be flashed on 111, but quantum mechanics prohibits this in every 111 run.*

4.15.3 Bell's Theorem

Bell [20, 21] showed, using basically the same postulates as those of EPR, that no deterministic local hidden-variables theory can reproduce all statistical predictions of quantum mechanics. Initially, Bell's argument has been applied to an EPR-type *Gedanken experiment* of Bohm; later, Bell's analysis was extended to actual systems, and experimental tests were suggested and performed (see, for example, Clauser and Shimony [61]). Essentially, the

particles on either side appear to be “more correlated” than can be expected by a classical analysis assuming locality (i.e. the impossibility of any kind of information or correlation transfer faster than light).

In what follows we will use Odifreddi [199] in presenting an elementary analysis of Bell’s result. The setting is the following. We consider two physical systems; on one two types of measurements are made (A, B), and on the other one two other types (C, D). The results are binary, so they will be denoted by “+” and “−”. We will repeat these measurements to ensure statistically relevant results. *Correlations* appear when measurements give the same outcome, that is, “++” and “−−”. The basic result is that in almost all cases, more “++” and “−−” (and less “+−” and “−+”) coincidences are recorded than one can explain by any local classical analysis.

Let $p(x|i)$ be the probability that, by taking the measure $i \in \{A, B\}$ on the first system, the outcome will be $x \in \{+, -\}$; $p(x|ij)$ is the probability that by taking the measure $i \in \{A, B\}$ on the first system *and* the measure $j \in \{C, D\}$ on the second, the outcome of the first system *alone* will be x ; $p(xy|ij)$ is the probability that by taking the measure i on the first system and measure j on the second system, the outcomes will be respectively, $x \in \{+, -\}$ and $y \in \{+, -\}$; finally, $p(x|ijy)$ is the probability that when taking the measures $i \in \{A, B\}$ on the first system and $j \in \{C, D\}$ on the second one, and having outcome y on the second, the outcome of the first will be x .

The main result can be stated as follows:

If the outcomes of the experiments on both systems are independent, that is

$$p(xy|ij) = p(x|i) \cdot p(y|j),$$

then the lack of correlation in one of the two types of measures cannot exceed the lack of correlation in the remaining types, that is, the following quadrangular inequality holds true:

$$\begin{aligned} p(+ - | AC) + p(- + | AC) &\leq p(+ - | AD) + p(- + | AC) \\ &+ p(+ - | BD) + p(- + | AC) \\ &+ p(+ - | BC) + p(- + | BC). \end{aligned} \quad (4.12)$$

It is remarkable that this inequality⁴³ can be obtained with just an elementary manipulation of binary variables. To see this, let’s denote $p(+|A)$ by a , $p(-|A)$ by $1 - a$ (due to the bivalence nature of measurements we have $p(+|A) + p(-|A) = 1$), and so on. Using the independence hypothesis, that is,

$$p(+ - | AC) = p(+|A) \cdot p(-|C) = a(1 - c),$$

and the like, the inequality (4.12) can be re-written as

⁴³ And, of course, all inequalities obtained by systematic permutations.

$$\begin{aligned} a(1-c) + (1-a)c &\leq a(1-d) + (1-a)d + b(1-d) \\ &\quad + (1-b)d + b(1-c) + (1-b)c, \end{aligned}$$

or, equivalently,

$$ab + bd + bc \leq ac + b + d,$$

where $a, b, c, d \in [0, 1]$. To finish we consider the following three cases:

- if $b \leq a$, then $c(b - a) \leq 0$, so $ad + bd + c(b - a) \leq ad + bd$, and (4.12) follows as $ad \leq d$ and $bd \leq b$;
- if $d \leq c$, then $a(d - c) + bd + bc \leq b + d$, so (4.12) follows;
- if $a \leq b$ and $c \leq d$, then either $b \leq d$ and in this case $d(a+b) + c(b-a) \leq b+d$, or $d \leq b$ and in this case $a(d-c) + b(d+c) \leq b+d$, and in each case we deduce (4.12).

The probabilistic hypothesis of independence can actually be decomposed in the conjunction of two hypotheses with more physical significance (see Jarett [144]):

Separability: The statistical outcomes performed on one system are *independent of the outcomes* performed on the other system:

$$p(x|ijy) = p(x|ij) \text{ and } p(y|ijx) = p(y|ij).$$

Locality: The statistical outcomes performed an experiment on one system are *independent of the types of experiments* performed on the other system:

$$p(x|ij) = p(x|i) \text{ and } p(y|ij) = p(y|j).$$

Separability says that the spatio-temporal separation between the two systems makes them reducible to individual parts, the “whole” is no more than the “sum of parts”; locality forbids any instantaneous interaction.

Separability and locality implies independence as

$$p(xy|ij) = p(xy|ijy) \cdot p(y|ij) = p(x|ij) \cdot p(y|ij) = p(x|i) \cdot p(y|j).$$

Consequently, *if the outcomes of the experiments on both systems are separable and local, then the lack of correlation in one of the two types of measures cannot exceed the lack of correlation in the remaining types*.

Probabilities can be interpreted as truth-values of elementary propositions, so the above analysis can be reformulated in the language of “classical logic”. Indeed, let’s write A for $p(+|A)$ and $\neg A$ for $p(-|A)$, and similarly for B, C . Further on, let’s notice that the elementary operations with probabilities can be reformulated as logical operations, namely, conjunction \wedge will correspond to product, disjunction \vee to sum, and implication \rightarrow to \leq .

A “logical” version of the quadrangular inequality can be deduced:

If the conjunction is distributive with respect to disjunction for all propositions $A, \neg A, B, \neg B, C, \neg C$, that is,

$$\alpha \wedge (\beta \vee \gamma) \rightarrow (\alpha \wedge \beta) \vee (\alpha \wedge \gamma),$$

then the following quadrangular implication holds true:

$$\begin{aligned} (A \wedge \neg C) \vee (\neg A \wedge C) &\rightarrow (A \wedge \neg D) \vee (\neg A \wedge D) \\ &\vee (D \wedge \neg B) \vee (\neg D \wedge B) \\ &\vee (B \wedge \neg C) \vee (\neg B \wedge C). \end{aligned}$$

First, use the following weak form of distributivity

$$\alpha \wedge (\neg \beta \vee \beta) \rightarrow (\alpha \wedge \neg \beta) \vee (\alpha \wedge \beta),$$

for $\alpha = X \wedge \neg Y$, and $\beta = Z$:

$$(X \wedge \neg Y) \wedge (\neg Z \vee Z) \rightarrow (X \wedge \neg Y \wedge \neg Z) \vee (X \wedge \neg Y \wedge Z),$$

so by the law of excluded middle we get:

$$(X \wedge \neg Y) \rightarrow (X \wedge \neg Y \wedge \neg Z) \vee (X \wedge \neg Y \wedge Z).$$

Weakening the conclusion we get:

$$(X \wedge \neg Y) \rightarrow (X \wedge \neg Z) \vee (Z \wedge \neg Y). \quad (4.13)$$

Using (4.13) for the triples $(X, Y, Z) = (A, C, D), (D, C, B)$ we get

$$(A \wedge \neg C) \rightarrow (A \wedge \neg D) \vee (D \wedge \neg C),$$

and

$$(D \wedge \neg C) \rightarrow (D \wedge \neg B) \vee (B \wedge \neg C),$$

which imply

$$(A \wedge \neg C) \rightarrow (A \wedge \neg D) \vee (D \wedge \neg B) \vee (B \wedge \neg C).$$

Similarly, we obtain the implication

$$(\neg A \wedge C) \rightarrow (\neg A \wedge D) \vee (\neg D \wedge B) \vee (\neg B \wedge C),$$

which concludes the argument.

Both quadrangular inequality and implication have been experimentally falsified, hence no theory satisfying their hypotheses can be physically correct. So, *locality* and *separability* cannot be simultaneously adopted.⁴⁴ The failure of independence affects Reichenbach's [248] *causality* principle: two correlated (non independent) events have a common cause, that there exists an event in their "past" with respect to which they are independent. So, we arrive at the idea of *synchronicity* that has important implication for Quantum Computation:

⁴⁴Quantum mechanics has chosen to drop separability.

there exist events which are correlated in a way which is neither causal nor causal.

Finally, the failure of *distributivity* – the “mark” of quantum logic, has been proved to be more pervasive than the universe of quantum mechanics statements: it is excluded from any logic aiming to describe the physical world. Is any hope to rescue classical logic, which seems to be so brutally excluded ...

4.16 Quantum Logic

Quantum logic pioneered by Birkhoff and von Neumann [33] (see also, Mackey [175], Jauch [143], Kalmbach [146], Cohen [62], Pulmannová [245], Svozil [283]) deals with propositions expressing properties of quantum systems. To every physical property P we can associate in a natural way the proposition

“the physical system has property P ”,

which means

“if the observable is measured, then the property P is observed”.

Such a proposition is always true or false.

Technically, quantum logic identifies logical entities with Hilbert space entities. In particular, elementary propositions p, q, \dots are associated with closed linear subspaces of a Hilbert space through the origin (zero vector); the implication relation \leq is associated with the set-theoretical subset relation \subset , and the logical or \vee , and \wedge , and not' operations are associated with the set-theoretic intersection \cap , the linear span \oplus of subspaces and the orthogonal subspace \perp , respectively. The negation of $p \leq q$ is denoted by $p \not\leq q$. The logical statement 1 which is always true is identified with the entire Hilbert space H , and its complement \emptyset with the zero-dimensional subspace (zero vector).

Two propositions p and q are *orthogonal* if $p \leq q'$. Two propositions p, q are *co-measurable* (*compatible, commuting*) if there exist three mutually orthogonal propositions a, b, c such that

$$p = a \vee b \text{ and } q = a \vee c.$$

Intuitively, propositions p and q consist of an “identical part” a and two orthogonal parts b, c . Clearly, orthogonality implies co-measurability, since if p and q are orthogonal, we may take a, b, c to be $0, p, q$, respectively.

A simple example is the propositional structure encountered in the quantum mechanics of spin state measurements of a spin one-half particle. Assume that the associated Hilbert space is two-dimensional and real-valued,

and consider measurements of the spin-component along one particular direction, say the x -axis.⁴⁵ There are two possible spin components of the particle, $-\frac{1}{2}, +\frac{1}{2}$, which can be codified as $-$ and $+$. The corresponding elementary propositions are:

- p_- = “the particle is in state $-$ ” = one-dimensional subspace spanned by the vector $(1, 0) = (1, 0)$, and
- p_+ = “the particle is in state $+$ ” = one-dimensional subspace spanned by the vector $(0, 1) = (0, 1)$.

The tautology 1 is the proposition “the particle is in state $-$ or in state $+$ ”, which is the whole space, and the absurdity 0 is the proposition “the particle is neither in state $-$ nor in state $+$ ”, which is the zero-dimensional subspace of $(0, 0)$ which is $(0, 0)$. The propositional structure obtained in this case is the classical Boolean algebra with two elements; its Hasse diagram appears in Figure 4.4.⁴⁶

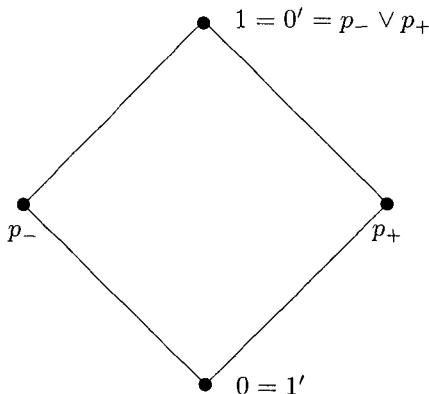


Figure 4.4: Hasse diagram of the spin one-half state co-measurable propositions.

So far we have discussed only about co-measurable observables. The corresponding propositions form a Boolean algebra with 2^n elements, so the so-called *blocks*⁴⁷ in case of a Hilbert space of dimension n . However, non-co-measurable observables should be treated as well! Of course, we may take the view that non-co-measurable observables make no physical sense, so we

⁴⁵This can be obtained with a Stern-Gerlach type of experiment.

⁴⁶Recall that in a Hasse diagram propositions are represented by dots, implication is represented “vertically”, that is q is drawn higher than p if $p \leq q$, and the propositions p and q are connected by a line.

⁴⁷Blocks are maximal in the sense that no additional observable, which is co-measurable with all observables in the block, can be added.

should forget about them (at least, with respect to current day knowledge; maybe, a “more complete” theory could make sense of them!). This legitimate, but somewhat “minority” position goes beyond our aim, so we will concentrate on the mainstream approach which considers that non-co-measurable observables make physical sense at least as theoretical constructions.⁴⁸

A simple formalism to deal with non-co-measurable observables is via the so-called *pasting* construction. Consider a collection of blocks and note that some of them may have a common non-trivial observable. A “logical” structure can be extracted as follows:

- identify all tautologies in all blocks,
- identify all absurdities in all blocks,
- identify identical elements in different blocks,
- keep intact the logical structure of all blocks.

As a simple example let’s paste together observables of the spin one-half systems, see Figure 4.4. Then we have two propositional systems, the first corresponding to the outcomes of a measurement of the spin states along the x -axis

$$L(x) = \{0, p_-, p_+, 1\},$$

and another one, corresponding to the outcomes of a measurement of the spin states along a different spatial direction, say $\bar{x} \neq x \pmod{\pi}$, an identical propositional system,

$$L(\bar{x}) = \{\bar{0}, \bar{p}_-, \bar{p}_+, \bar{1}\}.$$

So, we identify tautologies $1 = \bar{1}$ and absurdities $0 = \bar{0}$ and keep all other propositions intact. The result is the $M\theta_2$ propositional structure⁴⁹ presented in Figure 4.5. It is easy to see that $M\theta_2$ is *not* any longer a Boolean algebra, since distributivity is not satisfied, as the following example shows:

$$p_- = p_- \vee 0 = p_- \vee (\bar{p}_- \wedge \bar{p}'_-) \neq (p_- \vee \bar{p}_-) \wedge (p_- \vee \bar{p}'_-) = 1 \wedge 1 = 1.$$

Algebraically, $M\theta_2$ is an *orthocomplemented lattice*, that is, any two elements have a least upper bound and a greatest lower bound,

$$\text{if } a \leq b, a \leq c, \text{ then } a \leq (b \wedge c),$$

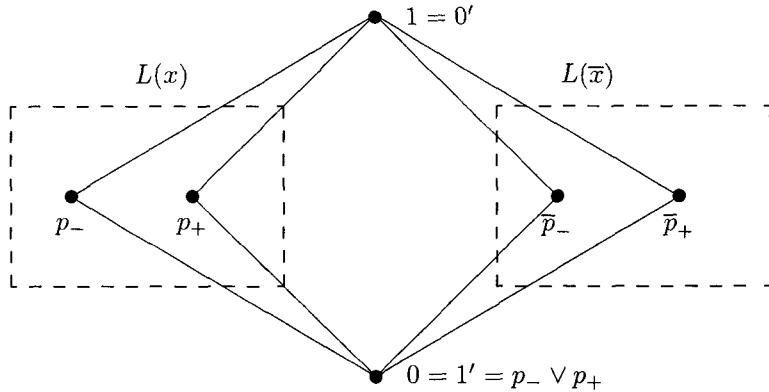
$$\text{if } b \leq a, c \leq a, \text{ then } (b \vee c) \leq a,$$

and for all $a \leq c$, the modular law is satisfied:

$$(a \vee b) \wedge c = a \vee (b \wedge c).$$

⁴⁸This attitude is common also in pure mathematics, where $i = \sqrt{-1}$ makes no “direct” sense, but proves to be extremely useful.

⁴⁹“ θ ” comes from orthocomplemented, “ M ” comes from modular.

Figure 4.5: Hasse diagram of M_{02} .

4.17 Have Quantum Propositions Classical Meaning?

Since measurement destroys quantum information, how do you actually get the results of your calculations? This question leads to the possible “classical meaning” of quantum propositions.

Einstein, Podolsky and Rosen [91] speculated that “elements of physical reality” exist irrespective of whether they are actually observed. Moreover, they conjectured that the quantum formalism can be “completed” or “embedded” into a larger theoretical framework which would reproduce the quantum theoretical results but would otherwise be classical and deterministic from an algebraic and logical point of view.

A proper formalization of the term “element of physical reality” can be given in terms of two-valued states or valuations, which can take on only one of the two values 0 and 1, and which are interpretable as the classical logical truth assignments *false* and *true*, respectively. Kochen and Specker’s results [152] (cf. also Specker [275], Svozil [283]) state that

for quantum systems representable by Hilbert spaces of dimension higher than two, there does not exist any valuation $s : L \rightarrow \{0, 1\}$ defined on the set of closed linear subspaces of the space L^{50} preserving the lattice operations and the orthocomplement, even if one restricts the attention to lattice operations carried out among commuting (orthogonal) elements.

As a consequence,

there exist different quantum propositions which cannot be distinguished by any classical truth assignment.

⁵⁰These subspaces are interpretable as quantum mechanical propositions.

The Kochen and Specker's result, as it is commonly argued, e.g. by Peres [236] and Mermin [189], is directed against the non-contextual hidden parameter program envisaged by [91]. Indeed, if one takes into account the entire Hilbert logic (of dimension larger than two) and if one considers all states thereon, any truth value assignment to quantum propositions prior to the actual measurement yields a contradiction.⁵¹ But, the Kochen-Specker argument continues, it is always possible to prove the existence of separable valuations or truth assignments for classical propositional systems identifiable with Boolean algebras. Hence, there does not exist any injective morphism from a quantum logic into some Boolean algebra.

Are there natural (weaker) conditions under which embeddings do exist? We will show that, if one is willing to *abandon* the preservation of some commonly used logical functions, then it is possible to give a classical meaning to quantum physical statements, thus suggesting a possible “understanding” of quantum mechanics.

4.17.1 A gallery of embeddings

An embedding of a quantum logical structure L of propositions into a classical universe represented by a Boolean algebra B should preserve as much logico-algebraic structure as possible. Such an embedding can be formalized as a mapping $\varphi : L \rightarrow B$ with the following properties.⁵² Let $p, q \in L$.

- (i) *Injectivity*: two different quantum logical propositions are mapped into two different propositions of the Boolean algebra: if $p \neq q$, then $\varphi(p) \neq \varphi(q)$.
- (ii) *Preservation of the order relation*: if $p \leq q$, then $\varphi(p) \leq \varphi(q)$.
- (iii) *Preservation of ortholattice operations*, i.e. preservation of the
 - (ortho-) complement: $\varphi(p') = \varphi(p)'$,
 - or operation: $\varphi(p \vee q) = \varphi(p) \vee \varphi(q)$,
 - and operation: $\varphi(p \wedge q) = \varphi(p) \wedge \varphi(q)$.

We cannot have an embedding from the quantum universe to the classical universe satisfying all three requirements (i)–(iii). In particular, the nonpreservation of ortholattice operations among non-co-measurable propositions is quite evident, because of the nondistributive structure of quantum logics.

⁵¹This can be proven by finitistic means, that is, with a finite number of one-dimensional closed linear subspaces (see Havlicek and Svozil [126]).

⁵²These properties are not independent.

4.17.2 Injective embeddings

We start with the simple fact that there does not exist an injective lattice morphism from any nondistributive lattice into a Boolean algebra. A good example is the lattice $M\theta_2$ drawn in Figure 4.5. Recall that $M\theta_2$ is a non-distributive lattice, in fact, the smallest orthocomplemented nondistributive lattice.

The requirement (iii) that the embedding φ preserves all ortholattice operations (even for non-co-measurable and non-orthogonal propositions) would mean that

$$\varphi(p_-) \wedge (\varphi(q_-) \vee \varphi(q_+)) \neq (\varphi(p_-) \wedge \varphi(q_-)) \vee (\varphi(p_-) \wedge \varphi(q_+)),^{53}$$

which is impossible because the range of φ is a subset of a Boolean algebra and for any Boolean algebra the distributive law is satisfied. Consequently, a lattice embedding in the form of an injective lattice morphism from Hilbert lattices into Boolean algebras is *not* possible even for two-dimensional Hilbert spaces. Could we still hope for a reasonable kind of embedding of a quantum universe into a classical one by weakening our requirements, most notably (iii)?

Let us follow Zierler and Schlessinger [307] and Kochen and Specker [152] and weaken (iii) by requiring that the ortholattice operations need only to be preserved *among co-measurable* propositions.⁵⁴ Again, the result is *negative*; for a proof see Calude, Hertling and Svozil [51].

An even stronger weakening of condition (iii) would be to require preservation of ortholattice operations merely among the centre C , i.e. among those propositions which are co-measurable (commuting) with all other propositions. It is not difficult to prove that in the case of complete Hilbert lattices (and not mere subalgebras thereof), the center consists of just the least lower and the greatest upper bound $C = \{0, 1\}$ and thus is isomorphic to the two-element Boolean algebra $\{0, 1\}$. The requirement is trivially fulfilled and its implications are quite trivial as well.

Is it possible to embed quantum logic into a Boolean algebra when one does not demand preservation of all ortholattice operations but merely of complementation? This is indeed possible and here is a sketch of the argument.

Recall that an *orthoposet*⁵⁵ (or *orthocomplemented poset*) $(L, \leq, 0, 1, {}')$ is a set L which is endowed with a partial ordering \leq ,⁵⁶ contains two distinguished elements 0 and 1 satisfying $0 \leq p \leq 1$, for all $p \in L$, and is endowed with a

⁵³We have written p_- for $(p_+)'$ and q_- for $(q_+)'$.

⁵⁴Mathematically, this is equivalent to the requirement of separability by the set of valuations or two-valued probability measures or truth assignments on L .

⁵⁵Poset is an abbreviation for partial ordered set.

⁵⁶That is, a subset \leq of $L \times L$ satisfying the following three conditions: (1) $p \leq p$, (2) if $p \leq q$ and $q \leq r$, then $p \leq r$, (3) if $p \leq q$ and $q \leq p$, then $p = q$, for all $p, q, r \in L$.

function' (*orthocomplementation*) from L to L satisfying the following three conditions

- (1) $p'' = p$,
- (2) if $p \leq q$, then $q' \leq p'$,
- (3) the least upper bound of p and p' exists and is 1, for all $p, q \in L$.⁵⁷

For example, an arbitrary sublattice of the lattice of all closed linear subspaces of a Hilbert space is an orthoposet if it contains the subspace $\{0\}$ and the full Hilbert space and is closed under the orthogonal complement operation. Namely, the subspace $\{0\}$ is the 0 in the orthoposet, the full Hilbert space is the 1, the set-theoretic inclusion is the ordering \leq , and the orthogonal complement operation is the orthocomplementation'.

Let L be a fixed arbitrary orthoposet. We shall construct a Boolean algebra B and an injective mapping $\varphi : L \rightarrow B$ which preserves the order relation and the orthocomplementation. To this aim we will use the notion of *maximal ideal*, that is, a subset I of L such that for all $p, q \in L$,

1. $p \in I$ if and only if $p' \notin I$,
2. if $p \leq q$ and $q \in I$, then $p \in I$.

Let \mathcal{I} be the set of all maximal ideals in L , and let B be the Boolean algebra which consists of all subsets of \mathcal{I} . The order relation in B is the set-theoretic inclusion, the ortholattice operations complement, or, and are given by the set-theoretic complement, union, and intersection, and the elements 0 and 1 of the Boolean algebra are just the empty set and the full set \mathcal{I} . Consider the map $\varphi : L \rightarrow B$ which maps each element $p \in L$ to the set $\varphi(p) = \{I \in \mathcal{I} \mid p \notin I\}$ of all maximal ideals which do not contain p . One can prove that the map φ has the following three properties (cf. Calude, Hertling and Svozil [51]):

- (i) is injective,
- (ii) preserves the order relation,
- (iii) preserves complementation,

that is,

every orthoposet can be embedded into a Boolean algebra where the embedding preserves the order relation and the complementation.

⁵⁷Note that these conditions imply $0' = 1$, $1' = 0$, and that the greatest lower bound of p and p' exists and is 0, for all $p \in L$.

A different embedding has been suggested by Malhas [177, 178]. Consider an orthocomplemented lattice $(L, \leq, 0, 1, ',)$, i.e. a lattice $(L, \leq, 0, 1)$ with $0 \leq x \leq 1$ for all $x \in L$, with orthocomplementation. Furthermore, assume that L is atomic, that is, for every $x \in L \setminus \{0\}$, there is an atom $a \in L$ such that $a \leq x$. An atom is an element $a \in L$ with the property that if $0 \leq y \leq a$, then $y = 0$ or $y = a$, and satisfies the following additional property: for all $x, y \in L$,

$$x \leq y \text{ if and only if for every atom } a \in L, a \leq x \text{ implies } a \leq y. \quad (4.14)$$

Every atomic Boolean algebra and the lattice of closed subspaces of a separable Hilbert space satisfy the above conditions.

Consider next a set U^{58} and let $W(U)$ be the smallest set of strings over the alphabet $U \cup \{', \rightarrow\}$ which contains U and is closed under negation (if $A \in W(U)$, then $A' \in W(U)$) and implication (if $A, B \in W(U)$, then $A \rightarrow B \in W(U)$).⁵⁹ The elements of U are called *simple propositions* and the elements of $W(U)$ are called *(compound) propositions*.

A *valuation* is a mapping $t : W(U) \rightarrow \{0, 1\}$ such that $t(A) \neq t(A')$ and $t(A \rightarrow B) = 0$ if and only if $t(A) = 1$ and $t(B) = 0$. Clearly, every assignment $s : U \rightarrow \{0, 1\}$ can be extended to a unique valuation t_s .

We continue with some natural definitions. A *tautology* is a proposition A which is true under every possible valuation, i.e. $t(A) = 1$, for every valuation t . A set $\mathcal{K} \subseteq W(U)$ is *consistent* if there is a valuation making true every proposition in \mathcal{K} . Let $A \in W(U)$ and $\mathcal{K} \subseteq W(U)$. We say that A *derives* from \mathcal{K} , and write $\mathcal{K} \models A$, in case $t(A) = 1$ for each valuation t which makes true every proposition in \mathcal{K} (that is, $t(B) = 1$, for all $B \in \mathcal{K}$). Let

$$Con(\mathcal{K}) = \{A \in W(U) \mid \mathcal{K} \models A\}.$$

In fact, the function *Con* is a finitary closure operator, i.e. it satisfies the following four properties:

- $\mathcal{K} \subseteq Con(\mathcal{K})$,
- if $\mathcal{K} \subseteq \tilde{\mathcal{K}}$, then $Con(\mathcal{K}) \subseteq Con(\tilde{\mathcal{K}})$,
- $Con(Con(\mathcal{K})) = Con(\mathcal{K})$,
- $Con(\mathcal{K}) = \bigcup_{\{X \subseteq \mathcal{K}, X \text{ finite}\}} Con(X)$.

Finally, we say that a set \mathcal{K} is a *theory* if \mathcal{K} is a fixed-point of the operator *Con*: $Con(\mathcal{K}) = \mathcal{K}$, that is, *Con*(\mathcal{K}) is the set of all propositions A which can be derived from \mathcal{K} .

⁵⁸Not containing the logical symbols $\cup, ', \rightarrow$.

⁵⁹We define in a natural way $A \cup B = A' \rightarrow B$, $A \cap B = (A \rightarrow B')'$, $A \leftrightarrow B = (A \rightarrow B) \cap (B \rightarrow A)$.

The main example of a theory can be obtained by taking a set X of valuations and constructing the set of all propositions true under all valuations in X :

$$Th(X) = \{A \in W(U) \mid t(A) = 1, \text{ for all } t \in X\}.$$

In fact, every theory is of the above form, that is,

for every theory \mathcal{K} there exists a set of valuations X (depending upon \mathcal{K}) such that $\mathcal{K} = Th(X)$.

In other words, *theories are those sets of propositions which are true under a certain set of valuations (interpretations).*

Let now T be a theory. Two elements $p, q \in U$ are T -equivalent, written $p \equiv_T q$, in case $p \leftrightarrow q \in T$. The relation \equiv_T is an equivalence relation. The equivalence class of p is $[p]_T = \{q \in U \mid p \equiv_T q\}$ and the factor set is denoted by U_{\equiv_T} ; for brevity, we will sometimes write $[p]$ instead of $[p]_T$. The factor set comes with a natural partial order: $[p] \leq [q]$ if $p \rightarrow q \in T$. Note that in general, (U_{\equiv_T}, \leq) is not a Boolean algebra.⁶⁰

In a similar way we can define the \equiv_T -equivalence of two propositions: $A \equiv_T B$ if $A \leftrightarrow B \in T$. Denote by $[[A]]_T$ (shortly, $[[A]]$) the equivalence class of A and note that for every $p \in U$, $[p] = [[p]] \cap U$. The resulting Boolean algebra $W(U)_{\equiv_T}$ is the Lindenbaum algebra of T .

Fix now an atomic orthocomplemented lattice $(L, \leq, 0, 1,')$ satisfying (4.14). Let U be a set of cardinality greater or equal to the cardinality of L and fix a surjective mapping $f : U \rightarrow L$. For every atom $a \in L$, let $s_a : U \rightarrow \{0, 1\}$ be the assignment defined by

$$s_a(p) = 1 \text{ if and only if } a \leq f(p).$$

Take

$$X = \{t_{s_a} \mid a \text{ is an atom of } L\}^{61} \text{ and } T = Th(X).$$

Malhas [177, 178] has proven that

the lattice (U_{\equiv_T}, \leq) is orthocomplemented, and, in fact, isomorphic to L .

In particular,

there exists a theory whose induced orthoposet is isomorphic to the lattice of all closed subspaces of a separable Hilbert space.

How does this relate to the Kochen-Specker impossibility result?

⁶⁰For instance, in case $T = Con(\{p\})$, for some $p \in U$. If U has at least three elements, then (U_{\equiv_T}, \leq) does not have a minimum.

⁶¹Recall that t_s is the unique valuation extending s .

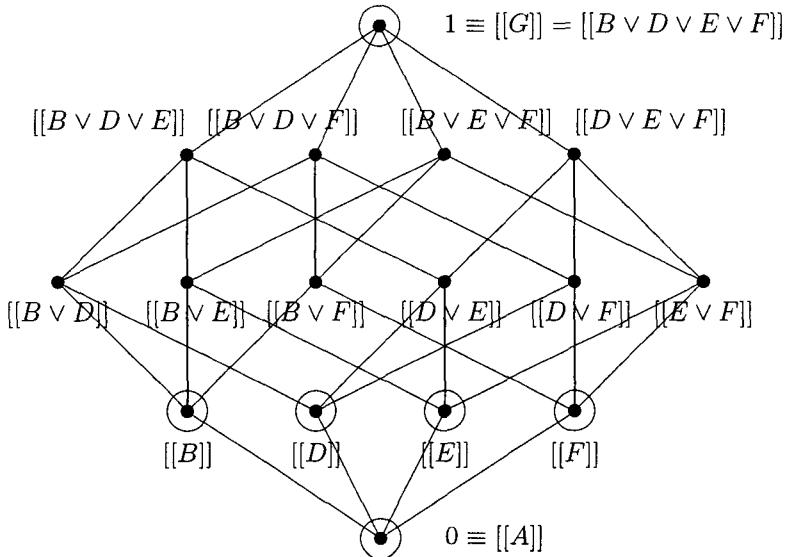


Figure 4.6: Hasse diagram of an embedding of the quantum logic $M0_2$. Concentric circles indicate the embedding.

Let us choose

$$U = \{A, B, C, D, E, F, G, H\}.$$

Since U contains more elements than $M0_2$, we can map U surjectively onto $M0_2$, e.g. $f(A) = 0, f(B) = p_-, f(C) = p_-, f(D) = p_+, f(E) = q_-, f(F) = q_+, f(G) = 1, f(H) = 1$.

For every atom $a \in M0_2$, let us introduce the truth assignment $s_a : U \rightarrow \{0, 1\}$ as defined above (i.e. $s_a(r) = 1$ if and only if $a \leq f(r)$) and thus a valuation on $W(U)$ separating it from the rest of the atoms of $M0_2$. That is, for instance, associate with $p_- \in M0_2$ the function s_{p_-} as follows:

$$\begin{aligned} s_{p_-}(A) &= s_{p_-}(D) = s_{p_-}(E) = s_{p_-}(F) = 0, \\ s_{p_-}(B) &= s_{p_-}(C) = s_{p_-}(G) = s_{p_-}(H) = 1. \end{aligned}$$

The truth assignments associated with all the atoms are listed in Table 4.4.

The theory T we are thus dealing with is determined by the union of all the truth assignments:

$$X = \{t_{s_{p_-}}, t_{s_{p_+}}, t_{s_{q_-}}, t_{s_{q_+}}\} \text{ and } T = Th(X).$$

By virtue of construction, U splits into six equivalence classes with respect to the theory T ; i.e.

$$U_{\equiv_T} = \{[A], [B], [D], [E], [F], [G]\}.$$

	A	B	C	D	E	F	G	H
s_{p-}	0	1	1	0	0	0	1	1
s_{p+}	0	0	0	1	0	0	1	1
s_{q-}	0	0	0	0	1	0	1	1
s_{q+}	0	0	0	0	0	1	1	1

Table 4.4: Truth assignments on U corresponding to atoms $p_-, p_+, q_-, q_+ \in MO_2$.

Since $[p] \rightarrow [q]$ if and only if $(p \rightarrow q) \in T$, we obtain a partial order on U_{\equiv_T} induced by T which isomorphically reflects the original quantum logic MO_2 . The Boolean Lindenbaum algebra $W(U)_{\equiv_T} = \{0, 1\}^4$ is obtained by forming all the compound propositions of U and imposing a partial order with respect to T . It is represented in Figure 4.6. The embedding is given by

$$\begin{aligned}\varphi(0) &= [[A]], \varphi(p_-) = [[B]], \varphi(p_+) = [[D]], \\ \varphi(q_-) &= [[E]], \varphi(q_+) = [[F]], \varphi(1) = [[G]].\end{aligned}$$

It is order-preserving but *does not* preserve operations such as the complement. Although, in this particular example, $f(B) = (f(D))'$ implies $(B \rightarrow D') \in T$, the converse is not true in general. For example, there is no $s \in X$ for which $s(B) = s(E) = 1$. Thus, $(B \rightarrow E') \in T$, but $f(B) \neq (f(E))'$.

4.17.3 Surjective extensions

The original program of completion of quantum mechanics naturally leads to injective embeddings because the physical intuition behind an embedding is that the “actual physics” is a classical one, but because of some yet unknown reason, some part of this “hidden arena” becomes observable while other part remains hidden.

Nevertheless, there exists at least one other alternative to complete quantum mechanics, namely by a *surjective map* $\phi : B \rightarrow L$ of a classical Boolean algebra onto a quantum logic, such that B “has more elements than L ”. An extension would be just a Boolean algebra B such that every element of L corresponds to at least two elements of B , one being the negation of the other. In such a situation the Kochen-Specker argument does not apply, because every element of L can be mapped by ϕ onto either one of its two or more correspondents. That is, ϕ depends upon the context of measurement. See more in Svozil [283].

Let us conclude with an observation about terminology. We have used the term “observables” for quantum propositions. Some “observables” (that is, the complementary ones) might not be co-measurable, so it seems appropriate to look at these “observables” as “potential observables”. After a

particular measurement has been chosen, some of these “potential observables” are actually determined and others (the complementary ones) become “counterfactuals” by quantum mechanical means.

4.18 Quantum Computers

A classical computer bogs down when it is made to simulate a quantum system; Feynman [95] suggested that a quantum computer might do a better job. In fact earlier, Benioff [22] devised an elaborate quantum-mechanical simulation of a Turing machine, the first half-classical, half-quantum Turing machine; his construction was as powerful as a Turing machine. In the same line of research Feynman [95, 96] proposed quantum versions of logical circuits. Albert [5] has described the first “truly” quantum computer, a “quantum automaton”; this machine has properties not shared by any classical automata. Quantum automata are, of course, not universal computers. In 1985 Deutsch [84] published a model of a quantum computer, the first “true” quantum universal Turing machine, one which is relying directly on the interference of quantum states. In the same year, 1985, Peres [235] improved Bennioff and Feynman models. In 1992 Deutsch and Jozsa [88] discussed a few problems that could be solved faster with quantum computers than conventional Turing machines. This was the first indication that quantum computing might be superior to classical computing.

4.18.1 Benioff’s computer

Benioff [22] devised a hybrid Turing machine, in which a classical part co-exists with a quantum part. His model is a Turing machine with a tape consisting of a sequence of qubits (spin states), each one being in one of the basis state $|0\rangle$ or $|1\rangle$. The head of the machine was replaced by a quantum mechanical interaction that could “change” the values of qubits. The transition rules came from a specific Schrödinger equation satisfying the following property: the initial configuration of spins evolves into a final set of spins which, when decoded as bits, will represent the result of the calculation. The program was encoded in the specific form of the Schrödinger equation. The computation was done in steps of fixed duration so that at the end of each step the tape was back in one of the basis states $|0\rangle$ (representing 0) or $|1\rangle$ (representing 1), but during a computational step the machine could temporarily be in a superposition of spin states.

At the end of each step the head measured the state of the tape which destroyed all superpositions on the tape. Consequently, quantum interference was only partially used, in fact it was destroyed at the end of each step. Hence, a classical Turing machine could simulate easily such a computation.

Benioff’s machine faces at least two major problems, (a) the design of the Hamiltonian that mimics a specific Turing machine has to incorporate all

computational paths performed by that specific machine (which, in a sense, amounts to knowing the answer of the problem you want to solve before actually running the computation), (b) the interactions between the head of the machine and its tape are difficult to realize because these physical objects can be far apart. The first problem was theoretically solved by Benioff, but the second one could not be solved.

4.18.2 Feynman's computer

Motivated by the construction of universal reversible Boolean gates, Feynman [95, 96] proposed a quantum universal simulator based on quantum versions of Boolean circuits. A quantum circuit is composed of quantum wires and elementary quantum gates. Initially, Feynman's approach was seen as restrictive (see Deutsch [84]), but in fact these two approaches are equivalent. However, it took almost a decade to get the proof, see Yao [302]:

every computation which can be performed efficiently on a quantum Turing machine can be done by a quantum circuit and conversely.

Feynman's construction uses a serial connection of k quantum gates, each performing a unitary transformation. There are n input/output qubits and $k + 1$ extra qubits as program counter sites, k “creation” operators, c_i and k “annihilation” operators, c_i . A creation operator c_i sets the i th counter qubit to 1 and the annihilation operator c_i sets the i th counter qubit to 0. These extra qubits are used to track the progress of a computation. Only one program counter site is ever occupied, by the “cursor”.

A computation starts by assigning the input bits into the input register and the cursor to site 0. One checks if the site k is empty or it has the cursor. When the cursor is found at the k th site, we know that the entire circuit has been used in the computation. At that time the state of the n qubits has the answer to the computation.

The quantum computer does not take care itself of termination! The time when the measurement is to be performed has to be determined from outside.

Peres [235] has improved Feynman's design in (a) timing the end of calculation, (b) the analysis of possible errors (in the program as well in measurements), and (c) extending the computation with general qubit states $a|0\rangle + b|1\rangle$.

4.18.3 Deutsch's computer

Deutsch's [84] starting point is a challenge to the Church–Turing Thesis. Any classical computation evolves from a set of input states to a set of output states; states are “labelled” in some standard way. For a classical Turing machine, the measured output is a definite function of the prepared input;

the measurement can, at least in principle, be done by an outside observer and will be the same if the measurement is repeated. In this way one can define, in a coherent way, the notion of Turing computable function. Two Turing machines are computationally equivalent when they compute the same function.

Quantum machines (and probabilistic or stochastic machines) *do not* compute functions in the above sense! The output state of a stochastic machine is “random” and only the probability distribution function for possible outputs depends on the input state. The quantum state of a quantum machine, although completely determined by the input state, is not an observable and, consequently, the user has no access to it. There exists, however, various ways to generalize the notion of “computational equivalence” for such machines. One possible way is to look carefully at labels. Like in the classical case, labels should be provided for all possible combinations of input states. As measurements cannot in general determine the output state, labels for output states should be done for the set of pairs consisting of an output observable and a possible measured value of that observable. Such a pair contains the specification of a possible experiment that could in principle carry on the output together with a possible result of the experiment. We can now say that two computing machines are computationally equivalent under given labellings if any possible experiment (or sequence of experiments) in which their inputs were prepared equivalently (with respect to the input labellings) and observables corresponding to each other under the output labellings are measured, the measured values of these observables are statistically indistinguishable. According to the above criterion, a given computing machine “computes” a unique function.

The Church–Turing Thesis is manifestly non-physical. There is, however, a subtle physical principle deriving from it, which, according to Deutsch [84], p. 99, reads:

Every finitely realizable physical system can be perfectly simulated by a universal model computing machine operating by finite means.

This statement asks for some definitions. A finitely realizable physical system includes any physical object on which experimentation is possible. A computing machine M is capable of perfectly simulating a physical system S , under given labelling of inputs and outputs, if there is a program for M that makes M computationally equivalent to S under that labelling. Rephrased, M becomes under a certain program and a fixed labelling, a system “computationally indistinguishable” from S . Deutsch [84], p.100, argued that the above principle is stronger than the Church–Turing Thesis: *it is not satisfied by Turing machines in classical physics, but it is compatible with quantum theory.* This observation motivates the interest, and the urgency, in seeking a “truly quantum” model of computation.

Deutsch's quantum Turing machine has an infinite sequence of qubits (the tape), \mathbf{t} , and a control consisting of a finite sequence of qubits, \mathbf{m} . In addition, Deutsch uses an observable, \mathbf{x} , which has an integer as possible value. The state of the quantum computer is a unit vector in the space spanned by basis vectors $|\mathbf{x}, \mathbf{t}, \mathbf{m}\rangle$. The dynamics is given by a constant unitary operator U . With these ingredients, Deutsch [84] has described a universal quantum computer, that is one capable of simulating perfectly every finitely realizable physical system (hence, every quantum computer). As applications he proposed the random generation procedure described in Section 4.14.

4.18.4 Reversible computation revisited

Recall that a classical computation is irreversible, so dissipative. If a computer operates reversibly, then at least in principle there need be no dissipation, so no power requirement. So, we may compute, in principle, for free. In what follows we will describe quantum realizations of sets of reversible gates which are universal for all Boolean circuits. Recall that a quantum circuit is composed of quantum wires and elementary quantum gates; each wire represents a path of a single qubit and is described by a state in the two dimensional Hilbert space \mathbb{C}^2 .

First we start by observing that

the transformation

$$T = |0\rangle\langle 0| \otimes U_1 + |1\rangle\langle 1| \otimes U_2, \quad (4.15)$$

where U_1 and U_2 are unitary transformations acting on \mathbb{C}^n , is also unitary.

Indeed, by considering the associated matrices, we have:

$$|0\rangle\langle 0| = \begin{pmatrix} 1 \\ 0 \end{pmatrix} (1, 0) = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix},$$

$$|1\rangle\langle 1| = \begin{pmatrix} 0 \\ 1 \end{pmatrix} (0, 1) = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix},$$

hence

$$|0\rangle\langle 0| \otimes U_1 + |1\rangle\langle 1| \otimes U_2 = \begin{pmatrix} U_1 & 0U_1 \\ 0U_1 & 0U_1 \end{pmatrix} + \begin{pmatrix} 0U_2 & 0U_2 \\ 0U_2 & U_2 \end{pmatrix} = \begin{pmatrix} U_1 & 0 \\ 0 & U_2 \end{pmatrix}.$$

On the other hand,

$$\begin{pmatrix} U_1 & 0 \\ 0 & U_2 \end{pmatrix} \begin{pmatrix} U_1 & 0 \\ 0 & U_2 \end{pmatrix}^\dagger = \begin{pmatrix} U_1 U_1^\dagger & 0 \\ 0 & U_2 U_2^\dagger \end{pmatrix} = \begin{pmatrix} I_n & 0 \\ 0 & I_n \end{pmatrix} = I_{2n},$$

where I_k denotes the identity $k \times k$ matrix.

The controlled- U gate

$$|0\rangle\langle 0| \otimes I + |1\rangle\langle 1| \otimes U,$$

where I is the single-qubit identity and U is another single-qubit gate which can be used in the particular case of

$$U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \text{NOT},$$

to obtain the controlled-NOT gate C_{not} .

Let's stop a moment and try to use C_{not} to "copy" a qubit in an unknown state. Assume that the qubit was in the state $|\psi\rangle = a|0\rangle + b|1\rangle$. We use as input $|\psi\rangle$ and $|0\rangle$, that is, $a(|0\rangle + b|1\rangle) \oplus |0\rangle = a|00\rangle + b|10\rangle$, and we apply C_{not} to it. The result is $a|00\rangle + b|11\rangle$. The two qubits are apparently in the same state, so we have contradicted the no cloning theorem discussed in Section 4.10!

The explanation is simple. When we measure one of the qubits we get 0 or 1 with probabilities $|a|^2$ and $|b|^2$. But, once we measure one qubit, the state of the other qubit is completely determined, so no extra information can be obtained about a, b , so the hidden information carried by $|\psi\rangle$ is lost because it was not copied. Although the two qubits appear to be identical, they are not independent copies from $|\psi\rangle$: they are two entangled qubits carrying together only one qubit of information.

The controlled-controlled-NOT (Toffoli) gate can be obtained from (4.15) by taking $U_1 = I \otimes I = I_4$ and $U_2 = C_{not}$:

$$T = |0\rangle\langle 0| \otimes I \otimes I + |1\rangle\langle 1| \otimes C_{not}.$$

The Fredkin gate FREDKIN can be defined by

$$\text{FREDKIN} = |0\rangle\langle 0| \otimes I \otimes I + |1\rangle\langle 1| \otimes S,$$

where S is the swap operation

$$S = |00\rangle\langle 00| + |01\rangle\langle 10| + |10\rangle\langle 01| + |11\rangle\langle 11|.$$

To prove that S is a unitary transformation we note that

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad |01\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix},$$

$$|10\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, |11\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix},$$

and

$$|00\rangle\langle 00| = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} (1, 0, 0, 0) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix},$$

$$|01\rangle\langle 10| = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} (0, 0, 1, 0) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix},$$

$$|10\rangle\langle 01| = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} (0, 1, 0, 0) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix},$$

$$|11\rangle\langle 11| = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} (0, 0, 0, 1) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

It follows that

$$S = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ and } SS^\dagger = S^2 = I_4,$$

hence S is unitary. According to (4.15), the transformation FREDKIN is also unitary. The matrix associated to FREDKIN is

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{aligned}
&= \left(\begin{array}{ccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right) + \left(\begin{array}{ccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \right) \\
&= \left(\begin{array}{ccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \right).
\end{aligned}$$

Any quantum gate can also be represented as a truth table: for each input basis vector we give the output of the gate. In this way, truth tables for Toffoli and Fredkin gates from Table 4.1 in Section 4.3 re-appear in Table 4.5.

Universality is an important consequence of the above analysis:

The transformation T given by (4.15) is universal for all Boolean circuits.

We already know that the gates AND and NOT form a universal set, hence it suffices to represent them in terms of the transformation T given by (4.15).⁶² We have $T|110\rangle = |111\rangle$ and $T|111\rangle = |110\rangle$, hence the third bit is changed, that is

$$T|1, 1, x\rangle = |1, 1, \neg x\rangle.$$

On the other hand, for $x, y \in \{0, 1\}$, $T|xy0\rangle = |xy1\rangle$ if and only if $x = y = 1$, and $T|xy0\rangle = |xy0\rangle$, otherwise. Consequently, the last bit of $T|xy0\rangle$ is $x \wedge y$,

$$T|x, y, 0\rangle = |x, y, x \wedge y\rangle.$$

While the T or F quantum gates are universal for Boolean circuits, they *cannot* achieve any quantum state transformation. Universality for quantum transformation is defined differently as we are dealing with continuous, not discrete, transformations, and the maximum one can hope for is an arbitrarily good approximation. A matrix M is ϵ -close to a unitary matrix U if $\|U - M\| \leq \epsilon$.⁶³ A set of quantum gates S is *universal* for quantum transformations

⁶²Recall that the basis states $|0\rangle$ and $|1\rangle$ encode the classical bit values 0 and 1, respectively.

⁶³Recall that $\|\psi\| = \sqrt{\langle\psi|\psi\rangle}$ is the norm.

Toffoli quantum gate		Fredkin quantum gate	
Input	Output	Input	Output
$ 000\rangle$	$ 000\rangle$	$ 000\rangle$	$ 000\rangle$
$ 010\rangle$	$ 010\rangle$	$ 010\rangle$	$ 010\rangle$
$ 100\rangle$	$ 100\rangle$	$ 100\rangle$	$ 100\rangle$
$ 110\rangle$	$ 111\rangle$	$ 110\rangle$	$ 101\rangle$
$ 001\rangle$	$ 001\rangle$	$ 001\rangle$	$ 001\rangle$
$ 011\rangle$	$ 011\rangle$	$ 011\rangle$	$ 011\rangle$
$ 101\rangle$	$ 101\rangle$	$ 101\rangle$	$ 110\rangle$
$ 111\rangle$	$ 110\rangle$	$ 111\rangle$	$ 111\rangle$

Table 4.5: Toffoli/Fredkin quantum gates.

if every unitary transformation U can be performed with arbitrary precision $\epsilon > 0$ by a quantum circuit $C_{U,\epsilon}$ consisting of gates from \mathcal{S} . Barenco and co-authors (see [13, 14]) have proved that

1. there is no one-qubit universal gate,
2. no classical gate can be universal for quantum computing,
3. C_{not} together with all single-bit quantum gates form a universal set of gates for quantum computing.

Deutsch, Barenco and Ekert [87] and Lloyd [171] have proven that almost any non-trivial two-qubit gate is universal for quantum computing.

4.19 Quantum Algorithms

Is quantum computing offering theoretically any substantial benefit over classical computing?

4.19.1 Deutsch's problem

The simplest way to illustrate the power of quantum computing is to solve the so-called *Deutsch's problem*. Consider a Boolean function $f : \{0, 1\} \rightarrow \{0, 1\}$ and suppose that we have a black box to compute it. We would like to know whether f is constant (that is, $f(0) = f(1)$) or balanced ($f(0) \neq f(1)$). To make this test classically, we need two computations of f , $f(0)$ and $f(1)$ and one comparison. Is it possible to do it better? The answer is affirmative, and here is a possible solution.

Suppose that we have a quantum black box to compute f . Consider the transformation U_f which applies to two qubits, $|x\rangle$ and $|y\rangle$ and produces $|x\rangle|y \oplus f(x)\rangle$.⁶⁴ This transformation flips the second qubit if f acting on the

⁶⁴By \oplus we denote, as usual, the sum modulo 2.

first qubit is 1, and does nothing if f acting on the first qubit is 0.

The black box is “quantum”, so we can choose the input state to be a superposition of $|0\rangle$ and $|1\rangle$. Assume first that the second qubit is initially prepared in the state $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Then,

$$\begin{aligned} U_f \left(|x\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \right) &= |x\rangle \frac{1}{\sqrt{2}}(|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle) \\ &= (-1)^{f(x)} |x\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \end{aligned}$$

Next take the first qubit to be $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. The black box will produce

$$\begin{aligned} U_f \left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \right) &= \frac{1}{\sqrt{2}}((-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle) \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\ &= \frac{1}{2}(-1)^{f(0)}(|0\rangle + (-1)^{f(0) \oplus f(1)}|1\rangle)(|0\rangle - |1\rangle). \end{aligned}$$

Next we will perform a measurement that projects the first qubit onto the basis $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$: we will obtain $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ if the function f is balanced and $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ in the opposite case. So, Deutsch’s problem was solved with only one computation of f . The explanation consists in the ability of a quantum computer to be in a blend of states: we can compute $f(0)$ and $f(1)$, but also, and more importantly, we can extract some information about f which tells us whether $f(0)$ is equal or not to $f(1)$.

Can any function $f : \{0, 1\} \rightarrow \{0, 1\}$ be implemented by a quantum gate array U_f ? The answer is affirmative. Identifying the values 0 and 1 with the kets $|0\rangle$ respectively $|1\rangle$, U_f may be defined as the linear operator $U_f : \mathbf{C}^4 \rightarrow \mathbf{C}^4$, which satisfies, for any $x, y \in \{0, 1\}$, the equality

$$U_f|x, y\rangle = |x, y \oplus f(x)\rangle. \quad (4.16)$$

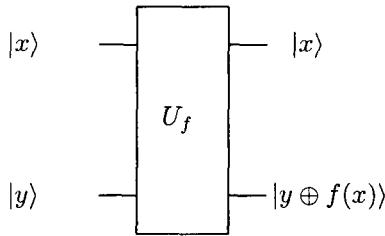
To compute $f(x)$ we apply U_f to $|x0\rangle$. Graphically, the transformation U_f is presented in Figure 4.7. We shall argue that

for any function $f : \{0, 1\} \rightarrow \{0, 1\}$, U_f is a unitary transformation.

We have

$$U_f U_f|x, y\rangle = U_f|x, y \oplus f(x)\rangle = |x, (y \oplus f(x)) \oplus f(x)\rangle = |x, y\rangle,$$

hence, in view of the equality $U_f U_f = I$, it suffices to prove that $U_f^\dagger = U_f$.

Figure 4.7: Quantum gate array U_f .

The function f can be defined in four ways: 1. $f(0) = f(1) = 0$; 2. $f(0) = 0, f(1) = 1$; 3. $f(0) = 1, f(1) = 0$; and 4. $f(0) = f(1) = 1$.

We will investigate the matrix U_f in each situation, taking into account the correspondences:

$$0 \rightarrow |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad 1 \rightarrow |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

In the first case, we have $U_f|x, y\rangle = |x, y \oplus 0\rangle = |x, y\rangle$, hence $U_f = I = U_f^\dagger$. In the second case, $U_f|00\rangle = |00\rangle$, $U_f|01\rangle = |01\rangle$, $U_f|10\rangle = |11\rangle$, $U_f|11\rangle = |10\rangle$, so it follows that

$$U_f = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} = U_f^\dagger.$$

A direct computation shows that in the third case, $U_f|00\rangle = |01\rangle$, $U_f|01\rangle = |00\rangle$, $U_f|10\rangle = |10\rangle$ and $U_f|11\rangle = |11\rangle$, therefore,

$$U_f = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = U_f^\dagger.$$

Finally, $U_f|x, y\rangle = |x, y \oplus 1\rangle$, i.e. $U_f|x0\rangle = |x1\rangle$ and $U_f|x1\rangle = |x0\rangle$, hence

$$U_f = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} = U_f^\dagger.$$

4.19.2 Quantum parallelism

Can the transformation U_f , discussed in the above section, be extended for arbitrary functions $f : \{0, 1\}^n \rightarrow \{0, 1\}$? Note that to compute a complete set of values for such a function we need to calculate f in all 2^n points, a huge task if n is big (say, $n = 100$).

The answer is mathematically easy, but computationally extremely powerful, it points out to one of the most important sources of strength of Quantum Computing.

Each vector $x = (i_1, i_2, \dots, i_n) \in \{0, 1\}^n$ can be identified with the state

$$|x\rangle = |i_1\rangle \otimes |i_2\rangle \otimes \dots \otimes |i_n\rangle = |i_1 i_2 \dots i_n\rangle,$$

so keeping in mind that U_f is a linear transformation, define U_f to be the linear operator acting on \mathbf{C}^{2n+2} which satisfies the equality (4.16),

$$U_f|x, y\rangle = |x, y \oplus f(x)\rangle,$$

for any $|x\rangle = |i_1 i_2 \dots i_n\rangle$, $i_k \in \{0, 1\}$, $1 \leq k \leq n$ and $y \in \{0, 1\}$. One can prove that

$$U_f U_f^\dagger = U_f U_f = I \text{ and } U_f|x, 0\rangle = |x, f(x)\rangle.$$

If the transformation U_f is applied to an input which is in a superposition then, taking into account the linearity, U_f is applied *simultaneously* to all basis vectors in the superposition and generates a superposition of the results. Thus it is possible to compute $f(x)$ for all the 2^n values of x in a *single* application of U_f . This effect is called *quantum parallelism*.

Typically, a quantum algorithm starts by preparing the function of interest in a superposition on all values. Starting with an n -qubit state $|00\dots 0\rangle$ and applying the Walsh–Hadamard transformation W , we get the superposition

$$\frac{1}{\sqrt{2^n}}(|00\dots 0\rangle + |00\dots 1\rangle + \dots + |11\dots 1\rangle) = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle.$$

This superposition is a compact “encoding” of all integers x in the interval $[0, 2^n]$. Using linearity, we get, again, a compact “encoding” of all values of f , the function we are interested in:

$$U_f \left(\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x0\rangle \right) = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} U_f|x0\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x, f(x)\rangle.$$

Consequently, n qubits are enough to work simultaneously with 2^n states, in a “strange” compact form; this gives quantum computing the *ability to perform an exponential amount of computation in a linear amount of physical space*.

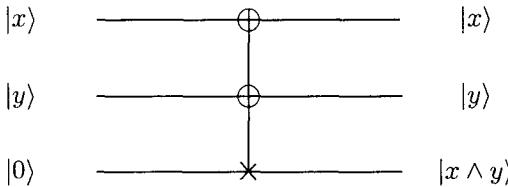


Figure 4.8: Quantum parallel computation of controlled-controlled-NOT.

Let's apply the above technique to the simple example of controlled-controlled-NOT gate TOFOLLI that computes the conjunction.

We have:

$$W_2|00\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$$

and

$$\begin{aligned} \text{TOFOLLI}(W|00\rangle \otimes |0\rangle) &= \frac{1}{2} \text{TOFOLLI}(|000\rangle + |010\rangle + |100\rangle + |110\rangle) \\ &= \frac{1}{2}(|000\rangle + |010\rangle + |100\rangle + |111\rangle). \end{aligned}$$

The resulting superposition can be viewed as a truth table for conjunction.

4.19.3 Quantum implementations

Recall that for any function $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$ there exists a quantum function $F : \mathbf{C}^{2^{n+m}} \rightarrow \mathbf{C}^{2^{n+m}}$ working on an n -qubits input and an m -qubits output register with $F|x, 0\rangle = |x, f(x)\rangle$. Invertible functions $f : \{0, 1\}^n \rightarrow \{0, 1\}^n$ can be directly realized by quantum functions $F : |i\rangle \rightarrow |f(i)\rangle$. While a direct implementation of F is possible with any universal set of quantum gates, the implementation can be substantially more efficient. For example, if we have efficient implementations of the quantum functions $U_f : |i, 0\rangle \rightarrow |i, f(i)\rangle$ and $U_{f^{-1}} : |i, 0\rangle \rightarrow |i, f^{-1}(i)\rangle$, then an overwriting operator F' can be constructed by using an n -qubit scratch register:

$$F' : |i, 0\rangle \xrightarrow{U_f} |i, f(i)\rangle \xrightarrow{\text{SWAP}} |f(i), i\rangle \xrightarrow{U_{f^{-1}}^\dagger} |f(i), 0\rangle.$$

Bennett's “uncomputing” procedure to control the amount of “junk” bits necessary to compute reversibly non-reversible functions can be presented as follows:

$$|x, 0, 0\rangle \xrightarrow{G} |x, g(x), 0\rangle \xrightarrow{H} |x, g(x), h(g(x))\rangle \xrightarrow{G^\dagger} |x, 0, f(x)\rangle.$$

Here f is the composition of two non-reversible functions $f(x) = h(g(x))$ (G and H are the quantum functions for g and h). The last step is merely the inversion of the first step and uncomputes the intermediate result. The second register can then be reused for further computations.

If the computation of a function $f(x)$ fills a scratch register with the junk bits $j(x)$ (i.e. $|x, 0, 0\rangle \rightarrow |x, f(x), j(x)\rangle$), a similar procedure can free the register again:

$$|x, 0, 0, 0\rangle \longrightarrow |x, f(x), j(x), 0\rangle \xrightarrow{FANOUT} |x, f(x), j(x), f(x)\rangle \longrightarrow |x, 0, 0, f(x)\rangle.$$

Again, the last step is the inversion of the first. The intermediate step is a *FANOUT* operation which copies the function result into an additional empty register. Possible implementations of *FANOUT* operation are,

$$|x, x \oplus y\rangle,$$

or

$$|x, y\rangle \rightarrow |x, (x + y)(\bmod 2^n)\rangle.$$

Conditional branching is a powerful tool used by conventional programs. A unitary operator, on the other hand, is static and has no internal flow-control. Nevertheless, we can conditionally apply an n qubit operator U to a quantum register by using an *enable* qubit and define an $n+1$ qubit operator U'

$$U' = \begin{pmatrix} I_n & 0 \\ 0 & U \end{pmatrix}.$$

So U is only applied to basis vectors where the enable bit is set. This can be easily extended to enable-registers of arbitrary length.

A conditional operator $U_{[[\mathbf{e}]]}$ – with enable register \mathbf{e} – is a unitary operator of the form

$$U_{[[\mathbf{e}]}} : |i, \epsilon\rangle = |i\rangle |\epsilon\rangle_{\mathbf{e}} \rightarrow \begin{cases} (U|i\rangle) |\epsilon\rangle_{\mathbf{e}}, & \text{if } \epsilon = 111\dots \\ |i\rangle |\epsilon\rangle_{\mathbf{e}}, & \text{otherwise.} \end{cases}$$

If the architecture allows the efficient implementation of the controlled-NOT gate

$$C_{\text{NOT}} : |x, y_1, y_2 \dots\rangle \rightarrow |(x \oplus \bigwedge_i y_i), y_1, y_2 \dots\rangle,$$

then conditional operators can be realized by simply adding the enable string to the control register of all controlled-not operations.

4.19.4 Quantum programming

Many quantum algorithms seem difficult to understand because of the heavy physics formalism (Dirac notation, matrices, gates, operators). In what follows we will briefly introduce the programming language QCL⁶⁵ invented by Ömer [202] and we will use it to present a few important quantum algorithms. QCL is a high level, architecture independent programming language for quantum computing; its syntax is derived from classical procedural languages, like C or Pascal. QCL allows a complete implementation and simulation of quantum algorithms (including classical components) in a single, consistent formalism. The interpreter qcl can simulate quantum computers with arbitrary numbers of qubits. All numerical simulations are handled by the QC library, cf. Ömer [202]. The command `include "filename"` tells the interpreter to process the file `filename.qcl`, before continuing with the current input file or command line; qcl looks for the file in the current directory and in the default include directory, which can be changed with the option `include-path`. In interactive use `include "filename"` can be abbreviated by `<<filename`.

The syntactic structure of a QCL program is described by a context free grammar. Syntactic expressions are defined as:

$$\begin{aligned} \text{expression-name} &\leftarrow \text{expression-def}_1 \\ &\leftarrow \text{expression-def}_2 \\ &\dots \quad \dots \end{aligned}$$

We will use *keywords* and other literal text in *Courier*, *subexpressions* in *Italic*, *optional expressions*, repeated 0 or 1 times, in square brackets [,], *multiple expressions*, repeated 0, 1 or n times, in braces { , }, *alternatives* written as $\text{alt}_1 | \text{alt}_2 | \dots$, *grouping* of expressions forced by round brackets (,), *character classes* including digits, *digit* \leftarrow decimal digit from 0 to 9, letters, *letter* \leftarrow alphabetic letter form a to z or A to Z, and characters, *char* \leftarrow printable character except ''.

A QCL Program is a sequence of *statements* and *definitions*:⁶⁶

$$\text{qcl-input} \leftarrow \{ \text{stmt} | \text{def} \}$$

Statements, from simple commands to complex control-structures, are executed as they are encountered. For example,

```
qcl> if random()>=0.5 { print "red"; } else { print "black"; }
: red
```

⁶⁵QCL stands for Quantum Computation Language.

⁶⁶Read from a file or from the shell prompt. In the latter case, input is restricted to one line which is implicitly terminated by ','.

Definitions are not executed but bind a value (variable or constant) or a block of code (routine-definition) to a *symbol* (identifier). Consequently, each symbol has an associated *type*, which can either be a *data type* or a *routine type*. The type defines the symbol access mode, by reference or by call. Here are two examples:

```
qcl> int counter=5;
qcl> int fac(int n) {if n<=0 {return 1;} else {return n*fac(n-1);}}
```

Expressions can be composed of *literals*, variable references and sub-expressions combined by operators and function calls. Some examples are:

```
qcl> print "5 out of 10:",fac(10)/fac(5)^2,"combinations."
: 5 out of 10: 252 combinations.
```

Classic data-types are: arithmetic types `int`, `real` and `complex` and the general types `boolean` and `string`.

QCL defines two unconventional operators, which are mainly used with quantum expressions:

Concatenation The concatenation operator & combines two quantum registers or two strings. Its precedence is equal to the arithmetic operators + and -.

Size The size-of operator # gives the length (i.e. the number of qubits) of any quantum expression. This is the operator with the highest precedence.

QCL provides external operators for general unitary 2×2 , 4×4 and 8×8 matrices, which can be used directly to implement sets of 1, 2 and 3 qubit gates.

```
extern operator Matrix2x2(
    complex u00,complex u01,
    complex u10,complex u11,
    qureg q);
extern operator Matrix4x4(...,qureg q);
extern operator Matrix8x8(...,qureg q);
```

Matrix operators are checked for unitarity before they are applied:

```
qcl> const i=(0,1);
qcl> qureg q[1];
qcl> Matrix2x2(i*cos(pi/6),i*sin(pi/6),(0,0),(1,0),q);
! external error: matrix operator is not unitary
```

We continue with some details about the implementation of a few important transformations.

The rotation of a single qubit defined by the transformation matrix Rot_θ

$$Rot_\theta = \begin{pmatrix} \cos \frac{\theta}{2} & \sin \frac{\theta}{2} \\ -\sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{pmatrix}$$

is implemented by

```
extern operator Rot(real theta, qureg q);
```

Hadamard gate of n -qubit registers is defined by⁶⁷

$$H : |i\rangle \rightarrow \sum_{j \in \{0,1\}^n} (-1)^{(i,j)} |j\rangle.$$

The vectors $\mathcal{B}' = \{i \in \{0,1\}^n \mid |i'\rangle = H|i\rangle\}$ form the *Hadamard (dual, parity) base* of $\mathcal{B} = \{i \in \{0,1\}^n \mid |i\rangle\}$. Since \mathcal{B}' only contains uniform superpositions that just differ by the signs of the basis vectors, the external implementation of H is called **Mix**:

```
extern operator Mix(quareg q);
```

The *conditional phase gate* is a conditional operator for the zero-qubit phase operator $e^{i\phi}$:

$$V(\phi) : |\epsilon\rangle \rightarrow \begin{cases} e^{i\phi}, |\epsilon\rangle, & \text{if } \epsilon = 111\dots, \\ |\epsilon\rangle, & \text{otherwise,} \end{cases}$$

```
extern operator CPhase(real phi, qureg q);
```

The external FANOUT operator of QCL is defined by

$$\text{FANOUT} : |x, y\rangle \rightarrow |x, x \oplus y\rangle,$$

```
extern qufunct Fanout(quconst a, quvoid b);
```

The *SWAP* operator, changing two qubits of equal sized registers ($\text{SWAP} : |x, y\rangle \rightarrow |y, x\rangle$) is defined by the transformation matrix:

$$\text{SWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

and implemented as:

```
extern qufunct Swap(quareg a, qureg b);
```

The controlled-NOT operator $C_{[[e]]}$ is the conditional operator to C ⁶⁸ with the enabled register **e**:

$$C_{[[e]]} : |b\rangle|\epsilon\rangle_e \rightarrow \begin{cases} |1-b\rangle|\epsilon\rangle_e, & \text{if } \epsilon = 111\dots, \\ |b\rangle|\epsilon\rangle_e, & \text{otherwise,} \end{cases}$$

is implemented as:

⁶⁷Recall that $H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$.

⁶⁸Recall that the NOT operator is defined by the matrix $C = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

```
extern qfunct Not(quareg q);
extern qfunct CNot(quareg q,quconst c);
```

The QCL versions of NOT and controlled-NOT work also on target registers:

```
qc1> qureg q[4]; qureg p[4];
qc1> Not(q);
[8/8] 1 |00001111>
qc1> CNot(p,q);
[8/8] 1 |11111111>
```

4.19.5 Quantum Fourier Transform

A common mathematical trick in solving a problem given by a “function” is to transform the function, work with the transform, and finally “un-transform” the result to deduce properties of the original function. Fourier transform is such a significant transformation (see Kronsjö [157] for more details). Fourier transform linearity is particular useful for Quantum Computing.

In general, Fourier transforms map from the time domain to the frequency domain. So, Fourier transforms map functions of period r to functions which have non-zero period values only at multiples of the frequency $\frac{1}{r}$. The Discrete Fourier Transform maps a q dimensional vector $\{f(0), f(1), \dots, f(q-1)\}$ into the vector $\{\bar{f}(0), \bar{f}(1), \dots, \bar{f}(q-1)\}$ as follows:

$$DFT : \bar{f}(c) = \frac{1}{\sqrt{q}} \sum_{y=0}^{q-1} e^{\frac{2\pi i}{q} cy} f(y), \quad (4.17)$$

for all $c \in \{0, 1, \dots, q-1\}$.

The fastest algorithm for computing DFT is the Fast Fourier Transform (FFT) invented by Cooley and Tukey [66]; it uses a binary decomposition of the exponent to perform the transformation in $O(n2^n)$ steps, where $q = 2^n$.

In analogy, the Quantum Fourier Transform QFT can be defined by:

$$QFT_q : |x\rangle \rightarrow \frac{1}{\sqrt{q}} \sum_{y=0}^{q-1} e^{\frac{2\pi i}{q} xy} |y\rangle.$$

When applied to a quantum superposition, QFT_q transforms the state

$$\frac{1}{\sqrt{q}} \sum_{y=0}^{q-1} f(y) |y\rangle,$$

into the state

$$QFT_q : \sum_{y=0}^{q-1} f(y) |y\rangle \rightarrow \sum_{c=0}^{q-1} \bar{f}(c) |c\rangle,$$

where $\bar{f}(c)$ comes from (4.17).

Note that

$$QFT_q : |0\rangle \rightarrow \frac{1}{\sqrt{q}} \sum_{c=0}^{q-1} |c\rangle,$$

so it has the same effect as a Hadamard transformation. Coppersmith [67] has used a combination of Hadamard transformations H and conditional phase gates V ⁶⁹ to obtain the transformation:

$$DFT' = \prod_{i=1}^{n-1} \left(H_{n-i-1} \left(\frac{\pi}{2} \right) \prod_{j=0}^{i-1} V_{n-i-1, n-j-1} \left(\frac{2\pi}{2^{i-j+1}} \right) \right) H_{n-1}.$$

The basis vectors of the transformed state $|\tilde{\psi}'\rangle = DFT'_q |\psi\rangle$ are given in reverse bit order, so to get the actual DFT_q , the bits have to be flipped. Here is a QCL program implementing this transformation:

```
operator dft(qureg q) { // main operator
    const n=#q;           // set n to length of input
    int i; int j;          // declare loop counters
    for i=0 to n-1 {
        for j=0 to i-1 {   // apply conditional phase gates
            CPhase(2*pi/2^(i-j+1),q[n-i-1] & q[n-j-1]);
        }
        Mix(q[n-i-1]);    // qubit rotation
    }
    flip(q);              // swap bit order of the output
}
```

The program contains two loops: the outer loop performs a Hadamard transformation from the highest to the lowest qubit (`Mix`), while the inner loop performs conditional phase shifts (`CPhase`) between the qubits.

The `dft` operator takes a quantum register (`quareg`) `q` as argument.⁷⁰ The number of qubits of a register can be determined at runtime by the size operator `#`; this permits register size independent operator definitions. Inside the operator definition, sub-operators are called just as sub-procedures in conventional languages: the actual sequence of operators is fully determined at runtime.⁷¹

The execution is context dependent: `DFT(q)` called from top level works as expected; the difference appears when we have a `!` call, `!DFT(q)`: *all operators within DFT are inverted and applied in reverse order*. Inverse execution can also take place implicitly, e.g. Bennett's trick is used.

For illustration, let's start the QCL interpreter and prepare a test state (see [202]):

⁶⁹Indices indicate the qubits operated on.

⁷⁰A quantum register is not a quantum state by itself, but a pointer indicating the target qubits in the overall machine state.

⁷¹This includes loops, in our specific example, `for`-loops, conditional statements, etc.

```
$ qcl -b5 -i dft.qcl
[0/5] 1 |00000>
qc1> qureg q[5];           // allocate a 5 qubit register
qc1> Mix(q[3:4]);         // rotate qubits 3 and 4
[5/5] 0.5 |00000> + 0.5 |10000> + 0.5 |01000> + 0.5 |11000>
qc1> Not(q[0]);           // invert first qubit
[5/5] 0.5 |00001> + 0.5 |10001> + 0.5 |01001> + 0.5 |11001>
```

We now have a periodic state with period $2^3 = 8$ and an offset of 1 composed of 4 basis vectors to which we can apply the DFT:

```
qc1> dft(q);
[5/5] 0.353553 |00000> + -0.353553 |10000> + 0.353553i |01000> +
-0.353553i |11000> + (0.25,0.25) |00100> + (-0.25,-0.25) |10100> +
(-0.25,0.25) |01100> + (0.25,-0.25) |11100>
```

The DFT “inverts” the period to $2^5/8 = 4$ and a periodic distribution with offset 0 is obtained. The information about the original offset is in the phase factors, and has no influence on the probability distribution:

```
qc1> dump q;
: SPECTRUM q: |143210>
0.125 |00000> + 0.125 |00100> + 0.125 |01000> + 0.125 |01100> +
0.125 |10000> + 0.125 |10100> + 0.125 |11000> + 0.125 |11100>
```

“Uncomputing” the DFT, that is calling `!DFT(q)`, brings back the initial configuration:

```
qc1> !dft(q);
[5/5] 0.5 |00001> + 0.5 |10001> + 0.5 |01001> + 0.5 |11001>
qc1> exit;
```

4.19.6 Shor’s algorithm

The factoring problem requires *finding a non-trivial factor of a given a composite integer*. More precisely, if N is given and known to be composite, find a non-trivial factor of it, that is, $1 < n_1 < N$ such that $n_1|N$.⁷²

Multiplying large numbers is computationally easy. In contrast, no conventional polynomial⁷³ algorithm for the factorization of large numbers is known.

The best known⁷⁴ conventional algorithm, the so-called *quadratic sieve*, needs

$$O\left(e^{(\frac{64}{9})^{\frac{1}{2}}(\ln N)^{\frac{1}{2}}(\ln \ln N)^{\frac{2}{3}}}\right)$$

⁷² n_1 divides N .

⁷³In the length of its input measured in bits.

⁷⁴Published; cf. Lenstra and Lenstra [164], Gruska [119].

operations for factoring a binary number of N bits;⁷⁵ this is exponential with the input size. There exist sub-exponential $O(2^{(\ln N)^{\frac{1}{3}}})$ randomized algorithms for factoring.⁷⁶

Why are we concerned with these subtleties regarding such a “simple” problem as factoring of integers? The multiplication of large prime numbers is a “one-way function” i.e. a function which can easily be computed, while computing its inversion is “thought” (but not proven) to be practically impossible. One-way functions play a major role in digital cryptography and are essential to public key crypto-systems where the key for encoding is public and only the key for decoding remains secret. An example is the RSA method invented in Rivest, Shamir and Adleman [251]. Their cryptographic algorithm is based on the “one-way” character of multiplying two large prime numbers p, q (typically of 512 or 1024 digits). Encryption seems to be secure provided it is not feasible to get p, q from their product $N = pq$.⁷⁷ The RSA method is one of the most popular public key systems and is implemented in many communication programs.

According to Hughes [141] conventional factoring perspectives with the best available method in 1997 (quadratic sieve) on a network of 1,000 work-stations is presented in Table 4.6.

Year/Number of bits	1024	2048	4096
2006	10^{15} years	5×10^{15} years	3×10^{29} years
2024	38 years	10^{12} years	7×10^{25} years
2042	3 days	3×10^8 years	2×10^{22} years

Table 4.6: Perspectives of factoring on conventional computers.

While it is generally believed (although not proven) that efficient prime factorization on a conventional computer is impossible, an efficient algorithm for quantum computers has been proposed in 1994 by Shor [269]; see also Beckman, Shari, Devabhaktuni and Preskill [19], Ekert and Jozsa [94].

A key idea is to reduce factoring to the problem of finding the period of an integer function. Consider first a composite N and let's find an integer x such that $x^2 \equiv 1 \pmod{N}$, and $x \not\equiv 1 \pmod{N}$, $x \not\equiv -1 \pmod{N}$. If x is an integer satisfying the above conditions, then $x^2 - 1 = (x - 1)(x + 1)$ is a multiple of N , but $x - 1$ is not divisible by N nor is $x + 1$, so there exists a non-trivial factor n_1 of N that divides say $x + 1$ or $x - 1$. A factor of

⁷⁵ $(\frac{64}{9})^{\frac{1}{3}} \approx 1.9$.

⁷⁶ Cf. Vazirani [292].

⁷⁷ It is not known whether breaking RSA is as hard as factoring.

N can then be found by Euclid's algorithm⁷⁸ in $O(\log n)$ time: we compute $\gcd(N, x+1)$ and $\gcd(N, x-1)$.

For example, if $N = 15$, then $x = 4$ is a solution of $x^2 \equiv 1 \pmod{15}$ that verifies the above requirements; $\gcd(15, 4+1) = 5$, $\gcd(15, 4-1) = 3$, so 5 and 3 are non-trivial divisors of 15.

Let G be the set of all integers x in the interval $[1, N]$ co-primes with N , that is $\gcd(x, N) = 1$.⁷⁹ Pick x in G and define the function

$$f_{N,x}(a) = x^a \pmod{N}.$$

The function $f_{N,x}$ is *periodic*, that is there exists an integer r such that $f_{N,x}(a) = f_{N,x}(a+r)$, for every $a \in G$. The smallest r such that $x^r \equiv 1 \pmod{N}$ is called the order of x and is denoted by $\text{ord}(x)$.

If the period r is even, then the equation

$$x^r \equiv 1 \pmod{N}$$

can be written in the following equivalent forms:

$$(x^{\frac{r}{2}})^2 \equiv 1 \pmod{N},$$

$$(x^{\frac{r}{2}})^2 - 1^2 \equiv 0 \pmod{N},$$

$$(x^{\frac{r}{2}} - 1)(x^{\frac{r}{2}} + 1) \equiv 0 \pmod{N}.$$

The product $(x^{\frac{r}{2}} - 1)(x^{\frac{r}{2}} + 1)$ is a multiple of N , hence, unless $x^{\frac{r}{2}} \equiv 1 \pmod{N}$, or $x^{\frac{r}{2}} \equiv -1 \pmod{N}$, both factors of the product *must have a non-trivial factor in common with N* . Consequently, the following algorithm can be used:

- Step 1. Choose a uniformly distributed integer $1 \leq x \leq N$.
- Step 2. If $\gcd(N, x) > 1$, then we have a factor of N and stop.
- Step 3. Compute the period r of the function $f_{N,x}$.
- Step 4. If r is odd or $x^{\frac{r}{2}} \equiv 1 \pmod{N}$ or $x^{\frac{r}{2}} \equiv -1 \pmod{N}$, then go to Step 1; otherwise, compute $\gcd(N, x+1)$, $\gcd(N, x-1)$ to get a factor of N , and stop.

Note that in case $N = p^i$, where p is a prime and $i > 1$, if $\gcd(x, p) = 1$ and $r = \text{ord}(x)$, then r is odd or $x^{\frac{r}{2}} \equiv 1 \pmod{N}$ or $x^{\frac{r}{2}} \equiv -1 \pmod{N}$.

⁷⁸The greatest common divisor \gcd of a, b can be computed by the scheme:

$$\gcd(a, b) = \begin{cases} b, & \text{if } a = 0 \pmod{b}, \\ \gcd(a, a \bmod b), & \text{if } a \neq 0 \pmod{b}, a > b. \end{cases}$$

⁷⁹This set is a group under the multiplication $(\bmod N)$.

So, this case is excluded by the above algorithm. However, we don't lose generality because power of primes can be factorized by conventional computers in polynomial time.

What are the chances of finding a factor of N by computing $\gcd(N, x+1)$ and $\gcd(N, x-1)$? This question is answered by the following evaluations (see, for example, Shor [269] or Gruska [119]):

- Assume that N is not a power of a prime. If x is selected uniformly distributed in G ,⁸⁰ then the probability to find a even r such that $x^{\frac{r}{2}} \not\equiv 1 \pmod{N}$ and $x^{\frac{r}{2}} \not\equiv -1 \pmod{N}$ is greater or equal to $\frac{1}{4}$.
- If N is odd and x is selected uniformly distributed in G , then the probability that $\text{ord}(x)$ is even is greater or equal to $\frac{1}{2}$.
- The probability of picking an element x , uniformly distributed in G , such that $r = \text{ord}(x)$ is even, $x^{\frac{r}{2}} \not\equiv 1 \pmod{N}$ and $x^{\frac{r}{2}} \not\equiv -1 \pmod{N}$ is greater or equal to $\frac{1}{2}$.

This analysis leads to the following conclusion:

If there is a polynomial time deterministic/probabilistic/quantum algorithm to compute the period of the function $f_{N,x}$, for every N, x , then there exists a deterministic/probabilistic/quantum algorithm to factorize any integer N .

Unfortunately, there is no *known* polynomial time deterministic/probabilistic algorithm to compute the period of the function $f_{N,x}$! Next we will follow Shor [269] in presenting a polynomial time quantum algorithm to compute the period of the function $f_{N,x}$.

Let F be quantum function $F : |x, 0\rangle \rightarrow |x, f(x)\rangle$ of the integer function $f : \mathbf{Z} \rightarrow \{1, 2, \dots, 2^m\}$ with the unknown period $r < 2^n$. To determine r , we need two registers, with the sizes of $2n$ and m qubits, which should be reset to $|0, 0\rangle$.

As a first step we produce, using the Hadamard transform H , a homogeneous superposition of all basis vectors in the first register by applying an operator U :

$$U|0, 0\rangle = \sum_{i=0}^{2^{2n}-1} c_i |i, 0\rangle \quad \text{with} \quad |c_i| = \frac{1}{2^n}.$$

Applying F to the resulting state gives:

$$|\psi\rangle = F \frac{1}{2^n} \sum_{i=0}^{2^{2n}-1} |i, 0\rangle = \frac{1}{2^n} \sum_{i=0}^{2^{2n}-1} |i, f(i)\rangle.$$

⁸⁰Recall that G is the set of all integers in the interval $[1, N]$ co-primes with N .

A measurement of the second register with the result $k = f(s)$ with $s < r$ reduces the state to

$$|\psi'\rangle = \sum_{j=0}^{\lceil q/r \rceil - 1} c'_j |rj + s, k\rangle \quad \text{with} \quad q = 2^{2n} \quad \text{and} \quad c'_j = \sqrt{\left\lceil \frac{r}{q} \right\rceil}.$$

The measurement selects the values $s, s+r, s+2r, \dots, s+r\alpha$, where α is the largest integer such that $s+r\alpha \leq q$, and $s \leq r$ is chosen randomly (by measurement). So, $\alpha \approx \frac{q}{r}$. The post-measurement state $|\psi'\rangle$ of the first register consists only of basis vectors of the form $|rj+s\rangle$. Reason: $f(rj+s) = f(s)$, for all j .

Assume now that $\alpha = \frac{q}{r} - 1$; later we will briefly discuss the general case. It is not possible to directly extract the period r (or a multiple of it) by measurement of the first register because of the random offset s . The result of a Fourier transform, however, is invariant⁸¹ to offsets of a periodic distribution. We have:

$$\begin{aligned} |\tilde{\psi}'\rangle &= QFT_q |\psi'\rangle \\ &= \frac{1}{\sqrt{q}} \sum_{y=0}^{q-1} \sqrt{\frac{r}{q}} \sum_{j=0}^{\lceil \frac{q}{r} \rceil - 1} e^{2\pi i \frac{y(rj+s)}{q}} |y, k\rangle \\ &= \sum_{y=0}^{q-1} \tilde{c}'_y |y, k\rangle, \end{aligned}$$

where

$$\tilde{c}'_y = \frac{\sqrt{r}}{q} \sum_{j=0}^{p-1} e^{\left(\frac{2\pi i}{q} y(jr+s)\right)} = \frac{\sqrt{r}}{q} e^{\phi_y} \sum_{j=0}^{p-1} e^{\left(2\pi i \frac{yj\tau}{q}\right)},$$

$$\phi_y = 2\pi i \frac{ys}{q} \quad \text{and} \quad p = \left\lceil \frac{q}{r} \right\rceil.$$

If $q = 2^{2n}$ is a multiple of r , then $\tilde{c}'_y = e^{\phi_y}/\sqrt{r}$ if y is a multiple of q/r , and 0 otherwise. But even if r is not a power of 2, the spectrum of $|\tilde{\psi}'\rangle$ shows distinct peaks with a period of q/r because

$$\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{k=0}^{n-1} e^{2\pi i k \alpha} = \begin{cases} 1, & \text{if } \alpha \in \mathbf{Z}, \\ 0, & \text{if } \alpha \notin \mathbf{Z}. \end{cases}$$

This is also the reason we use a register of $2n$ qubits when $r < 2^n$: it guarantees at least 2^n elements in the above sum and thus a peak width of order $O(1)$.

⁸¹Except for phase factors which don't effect the probability.

Now measure the first register: we will get a value c close to $\lambda q/r$ with $\lambda \in \mathbb{Z}_r$. This can be written as $c/q = c \cdot 2^{-2n} \approx \lambda/r$. We can think of this result as a rational approximation a/b with $a, b < 2^n$ for dyadic number $c \cdot 2^{-2n}$. An efficient classical algorithm for solving this problem using continued fractions is described in Hardy and Wright [122] and is implemented in the denominator function.

Since the form of a rational number is not unique, λ and r are only determined by $a/b = \lambda/r$ if $\gcd(\lambda, r) = 1$. The probability that λ and r are co-prime is greater than $1/\ln r$, so only $O(n)$ trials are necessary to achieve a probability of success as close to one as desired; in fact, as observed by Shor [269], the expected number of trials can be decreased to a constant.⁸²

The implementation of the Shor algorithm uses the following functions:

- boolean testprime(int n): Tests whether n is a prime number.⁸³
- boolean testprimepower(int n): Tests whether n is a prime power.
- int powmod(int x, int a, int n): Calculates $x^a \pmod{n}$.
- int denominator(real x, int qmax): Returns the denominator q of the best rational approximation $\frac{p}{q} \approx x$ with $p, q < q_{\max}$.

The procedure `shor` checks whether the integer `number` is suitable for quantum factorization, and then repeats Shor's algorithm until a factor has been found ([202]).

```

procedure shor(int number) {
    int width=ceil(log(number,2));      // size of number in bits
    qureg reg1[2*width];               // first register
    qureg reg2[width];                 // second register
    int qmax=2^width;
    int factor;                       // found factor
    int m; real c;                    // measured value
    int x;                            // base of exponentiation
    int p; int q;                     // rational approximation p/q
    int a; int b;                     // possible factors of number
    int e;                            // e=x^(q/2) mod number

    if number % 2 == 0 { exit "number must be odd"; }
    if testprime(number) { exit "prime number"; }
    if testprimepower(number) { exit "prime power"; };
}

```

⁸²If the supposed period $r' = b$ derived from the rational approximation $a/b \approx c 2^{-2m}$ is odd or $\gcd(x^{r'/2} \pm 1, n) = 1$, then one could try to expand a/b by some integer factor k in order to guess the actual period $r = kb$.

⁸³Since both test functions are not part of the algorithm itself, short but inefficient implementations with $O(\sqrt{n})$ have been used in [202].

```

{
    {
        // generate random base
        x=floor(random()*(number-3))+2;
    } until gcd(x,number) == 1;
    print "chosen random x =",x;
    Mix(reg1); // Hadamard transform
    expn(x,number,reg1,reg2); // modular exponentiation
    measure reg2; // measure 2nd register
    dft(reg1); // Fourier transform
    measure reg1,m; // measure 2st register
    reset; // clear local registers
    if m==0 { // failed if measured 0
        print "measured zero in 1st register. trying again ...";
    } else {
        c=m*0.5^(2*width); // fixed point form of m
        q=denominator(c,qmax); // find rational approximation
        p=pfloor(q*m*c+0.5);
        print "measured",m," approximation for",c,"is",p,"/",q;
        if q (mod 2) == 1 and 2*q<qmax { // odd q ? try expanding p/q
            print "odd denominator, expanding by 2";
            p=2*p; q=2*q;
        }
        if q (mod 2) == 1 { // failed if odd q
            print "odd period. trying again ...";
        } else {
            print "possible period is",q;
            e=powmod(x,q/2,number); // calculate candidates for
            a=(e+1) mod number; // possible common factors
            b=(e+number-1) mod number; // with number
            print x,"^",q/2," + 1 mod",number,"=",a,",",
                  x,"^",q/2," - 1 mod",number,"=",b,
                  factor=max(gcd(number,a),gcd(number,b));
        }
    }
} until factor>1 and factor<number;
print number,"=",factor,"*",number/factor;
}

```

Shor's algorithm is probabilistic, so it may fail. The smallest number that can be factorized with Shor's algorithm is 15, as it's the product of smallest odd prime numbers 3 and 5. Ömer's [202] implementation of the modular exponentiation needs $2l + 1$ qubits scratch space with $l = \lceil \log(15 + 1) \rceil = 4$. The algorithm itself needs $3l$ qubits, so a total of 21 qubits must be provided.

```

$ qcl -b21 -i shor.qcl
qcl> shor(15)
: chosen random x = 4
: measured zero in 1st register. trying again ...
: chosen random x = 11
: measured 128 , approximation for 0.500000 is 1 / 2
: possible period is 2
: 11 ^ 1 + 1 (mod 15) = 12 , 11 ^ 1 - 1 (mod 15) = 10
: 15 = 5 * 3

```

The first try failed because 0 was measured in the first register of $|\psi\rangle$ and

$\lambda/r = 0$ which gives no information about the period r .

One might argue that this is not likely to happen, since the first register has 8 qubits and 256 possible basis vectors; however, if a number n is to be factored, one might expect a period about \sqrt{n} assuming that the prime factors of n are of the same order of magnitude. This leads to a period q/\sqrt{n} after the DFT and the probability $p = 1/\sqrt{n}$ to accidentally pick the vector $|0\rangle$ is about $p = 25.8\%$.

The second try also had the same probability of failure since $11^2 \pmod{15} = 1$, but this time, the measurement picked the second peak in the spectrum at $|128\rangle$. With $128/2^8 = 1/2 = \lambda/r$, the period $r = 2$ was correctly identified and the factors $\gcd(11^{2/2} \pm 1, 15) = \{3, 5\}$ to 15 have been found.

Shor's algorithm shows that factoring can be checked in polynomial time by a quantum algorithm. However, there is *no* proof that factoring cannot be checked in polynomial time by a deterministic conventional algorithm. If efficient ways to simulate quantum computers on conventional machines will be found, then Shor's algorithm will be transformed into an efficient conventional procedure for factoring. However, this possibility seems very improbable.

We finish this section by citing again Hughes [141] with estimations displayed in Table 4.7 on factoring on quantum computers (with minimal clock speed of 100 MHz).

Number of bits	1,024	2,048	4,096
number of qubits	5,124 years	10,244 years	20,484 years
number of gates	3×10^9 years	2×10^{11} years	2×10^{12} years
factoring time	4.5 mins	36 min	4.8 hours

Table 4.7: Perspectives of factoring on a (hypothetical) quantum computer.

So, according to Tables 4.6, 4.7, using RSA with 2,048-bit numbers may be safe for the next 50 years if no quantum computer will be constructed by then; in the opposite case, even working with 4,096-bit numbers may not be safe if quantum computers become available!

4.19.7 Grover's algorithm

Many problems, ranging from sorting and graph colouring to database search, can be phrased as search problems of the form “find some x such that $P(x)$ is true”, where P is an appropriate predicate. For example:

- (a) Given an n element vector A , find a permutation π of $\{1, 2, \dots, n\}$ such that for all $1 \leq i < n$ we have $A_{\pi(i)} < A_{\pi(i+1)}$.
- (b) Given a graph $(V; E)$ with n vertices V and e edges $E \subset V \times V$, and a set of k colours C , find a colouring map c from V to C such that for all (v_1, v_2) in E , $c(v_1) \neq c(v_2)$.

Some problems, such as 3-SAT or graph colourability, have an “inner structure” that can be exploited to construct full solutions from (smaller) partial solutions; in such cases, efficient algorithms are known. But, in general, the search space has no special structure. To see the difference between a structured and an unstructured search space think, with Brassard [36], of the task of finding someone’s (you know her name) phone number in a city’s directory versus the task of finding the name of a stranger whose phone number you happen to know. To search a simple unstructured file, a computer would have to run through, on average, half of the data to locate an entry x satisfying $P(x)$. It is easily proven that there can be, in general, no shortcuts, so randomly testing the predicate P seems the best strategy that can be adopted on a conventional computer. For a search space of size N , the general unstructured search problem is of complexity $O(N)$, once the time it takes to test the predicate P is factored out. When we said “no shortcuts are possible in general” we meant “no shortcuts are possible if we use conventional computers”. However, Grover [118] showed that the unstructured search problem *can be solved* with bounded probability within $O(\sqrt{N})$ time with a quantum computer.

This advantage is not dramatic for our example of locating a name in the phone directory of a city; however, things are different if we are interested in databases that are so large that they would not fit in the memory of all the world’s (conventional) computers put altogether. Such databases cannot “exist” explicitly, of course, but they can be specified by a rule that allows the construction of any required record on demand. As an example consider one of the most widely used systems to protect the confidentiality of information, the Data Encryption Standard (DES). Encipher and decipher are controlled by a 56-bit key, which the legitimate participants must share in secret. The goal of an eavesdropper, that has intercepted matching pairs of clear and enciphered text, is to find the key that maps one into the other. This problem can be described by a virtual “phone directory” in which each possible key is a name and the enciphered text with that key is the corresponding phone number. Given the intercepted enciphered text, our name-finding problem corresponds to searching for the required secret key to decode the rest of the enciphered text. An exhaustive search needs to try an average of 2^{55} keys before hitting the right one, an operation that takes more than 1 year even if one billion keys are checked every second. By comparison, Grover’s algorithm can solve the problem, after quantum-DES enciphering the known clear text, in just 185 million times. Hence, Grover’s quantum searching algorithm can “in principle” be used to break classical cryptographic systems such as DES.

Grover's search algorithm is more efficient than any known conventional algorithm searching a completely unstructured solution space. More precisely, *Grover's algorithm is optimal for completely unstructured searches*, see Zalka [305]. But, many search problems involve searching a structured solution space. One would expect that this extra information would enable more efficient searching strategies. Hogg [139] has developed quantum algorithms that use the problem structure in a similar way to classical heuristic search algorithms. Small cases indicate that Hogg's algorithms are more efficient than Grover's algorithm applied to structured search problems, but the speed up is likely to be only polynomial.

As we already pointed out, Grover's algorithm searches an unstructured list of size N to find one item satisfying a given condition. It is assumed that it is easy to check whether an arbitrary element satisfies the given condition. Let n be such that $2^n \geq N$. Assume that predicate P is implemented by a quantum gate

$$U_P |x, 0\rangle \rightarrow |x, P(x)\rangle,$$

where true is encoded as 1. Grover's algorithm consists of the following steps:

Step 1. Prepare a register containing a superposition of all of the possible values $x_i \in \{0, 1, \dots, 2^n - 1\}$.

Step 2. Compute $P(x_i)$ on this register.

Step 3. Change amplitude a_j to $-a_j$, for all x_j such that $P(x_j) = 1$.

Step 4. Apply inversion about the average to increase amplitude of x_j with $P(x_j) = 1$.

Step 5. Repeat steps 2 through 4 $\frac{\pi}{4} 2^{\frac{n}{2}}$ times.

Step 6. Read the result.

Computing P for all possible inputs x_i can be done by applying U_P to a register containing the superposition

$$\frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle$$

with a register set to 0:

$$U_P : \frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle \rightarrow \frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x, P(x)\rangle.$$

This is done with the classical technique in steps 1 and 2.

The difficulty is, as usual, to obtain a useful result from this superposition. For any x_i such that $P(x_i)$ is true, $|x_i, 1\rangle$ will be part of the result superposition, but chances are very slim to get it via measurement, 2^{-n} , since its amplitude is $\frac{1}{2^{n/2}}$. The "trick" is to change the resulting quantum

state in such a way to greatly increase the amplitude of all vectors $|x_i, 1\rangle$, for which the predicate is true, and decrease the amplitude of vectors $|x_j, 0\rangle$, for which the predicate is false. This change in amplitudes is obtained using the inversion about the average transformation; it can be accomplished with $O(\log(N))$ quantum gates.

The transformation

$$\sum_{i=0}^{N-1} a_i |x_i\rangle \rightarrow \sum_{i=0}^{N-1} (2A - a_i) |x_i\rangle$$

is performed by the $N \times N$ unitary matrix

$$D = \begin{pmatrix} \frac{2}{N} - 1 & \frac{2}{N} & \dots & \frac{2}{N} \\ \frac{2}{N} & \frac{2}{N} - 1 & \dots & \frac{2}{N} \\ \dots & \dots & \dots & \dots \\ \frac{2}{N} & \frac{2}{N} & \dots & \frac{2}{N} - 1 \end{pmatrix}.$$

The matrix D can be defined as $D = WRW$ where W is the Walsh–Hadamard transformation and R is the matrix:

$$R = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & 0 & 1 \end{pmatrix}.$$

Next we explain how the inversion of amplitudes works. Consider the gate array

$$U_P : |x, b\rangle \rightarrow |x, b \oplus P(x)\rangle.$$

Apply to U_P the superposition

$$|\psi\rangle = \frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle$$

and choose

$$b = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

to obtain a state where the sign of all x with $P(x) = 1$ has been changed, but b is unchanged. Indeed, let $X_i = \{x \mid P(x) = i\}$, $i = 0, 1$ and let's compute $U_P|\psi, b\rangle$:

$$\begin{aligned} & \frac{1}{2^{(n+1)/2}} U_P \left(\sum_{x \in X_0} |x, 0\rangle + \sum_{x \in X_1} |x, 1\rangle - \sum_{x \in X_0} |x, 1\rangle - \sum_{x \in X_1} |x, 1\rangle \right) \\ &= \frac{1}{2^{(n+1)/2}} \left(\sum_{x \in X_0} |x, 0 \oplus 0\rangle + \sum_{x \in X_1} |x, 0 \oplus 1\rangle - \sum_{x \in X_0} |x, 1 \oplus 0\rangle \right) \end{aligned}$$

$$\begin{aligned}
& - \sum_{x \in X_1} |x, 1 \oplus 1\rangle) \\
= & \frac{1}{2^{(n+1)/2}} (\sum_{x \in X_0} |x, 0\rangle + \sum_{x \in X_1} |x, 1\rangle - \sum_{x \in X_0} |x, 1\rangle - \sum_{x \in X_1} |x, 0\rangle) \\
= & \frac{1}{2^{n/2}} (\sum_{x \in X_0} |x\rangle - \sum_{x \in X_1} |x\rangle) \oplus b,
\end{aligned}$$

so the amplitude of the states in X_1 has been inverted.

Grover's algorithm is optimal up to a constant factor; no quantum algorithm can perform an unstructured search faster. If there is only a unique x_0 such that $P(x_0)$ is true, then after $\frac{\pi}{8}2^{\frac{n}{2}}$ iterations of steps 2 through 4 the failure rate is 1/2. After iterating $\frac{\pi}{4}2^{\frac{n}{2}}$ times the failure rate drops to 2^{-n} . For a more detailed analysis see Gruska [119].

Additional iterations will increase the failure rate: for example, after $\frac{\pi}{2}2^{\frac{n}{2}}$ iterations, the failure rate is close to 1. This is an important feature of many quantum algorithms, which has little counterpart in conventional computing. Repeating quantum procedures may improve results for a while, but after some repetitions the results will get worse again. Quantum procedures are unitary transformations, which are rotations of complex space; repeated applications of a quantum transform may rotate the state closer and closer to the desired state for a while, but eventually it will rotate past the desired state to get further and further from the desired state. Thus to obtain useful results from a repeated application of a quantum transformation, it is paramount to know when to stop.⁸⁴

4.20 Quantum Complexity

4.20.1 Probabilistic computation

Probabilistic methods of computation have become increasingly popular, not only in theoretical studies (where they have been successfully applied to the study of average case complexity of deterministic algorithms), but in many practical applications. A probabilistic algorithm is a procedure that behaves deterministically except in some cases when it takes "decisions" pseudo-randomly, according to a fixed probability distribution. To illustrate this type of computation consider Rabin's [246] test, a polynomial probabilistic algorithm for checking the primality of an integer:

```

input n
choose pseudo-randomly m integers, x1, x2, ..., xm,
such that 1 ≤ xj ≤ n

```

⁸⁴This is a sensible attitude, in general.

```

for each  $x_i$  perform the test  $W(x_i, n)$ 
if  $W(x_j, n)$  holds true for some  $j$ , then accept
else reject
end

```

For an integer $1 \leq x < n$, the test $W(x, n)$ is passed if either $x^{n-1} \not\equiv 1 \pmod{n}$, or there exists an integer $m = (n-1)2^{-i}$ such that $x^m - 1$ and n have a common divisor different from 1 and n . Rabin [246] proved that if $W(x, n)$ holds true, then n is composite, and, if n is composite, then at least half of integers $1 \leq x < n$ satisfy $W(x, n)$. So, if Rabin's algorithm accepts, then n is composite; if it rejects, then n is "probably" prime, as n is composite only in case every chosen integer x_1, x_2, \dots, x_m is not a witness to the compositeness of n . As no less than half of the integers less than n are such witnesses, picking m pseudo-randomly chosen integers and not finding among them a witness to the compositeness of n implies a mistake whose probability is smaller than 2^{-m} .

Consider a non-deterministic Turing machine and assume that for every $s \in K, a \in V$, $\text{card}(\delta(s, a)) = 2$, for every input x the machine M executes the same number of steps, and every computation ends in a final state, which accepts or rejects. In this way, the set of all possible computations of M on an input x is a complete binary tree. The *error probability* of a probabilistic Turing machine M is the function which assigns to every input x the ratio of the number of computations of M on x giving the wrong answer to the total number of computations of M on x .

A *probabilistic Turing machine* M is a non-deterministic Turing machine satisfying the above conditions *plus* a rule for acceptance. There are two important rules: (a) the simple majority rule, and (b) the clear majority rule.

According to the *simple majority rule* M accepts the input x when more than half of the computations of M on x end in an accepting state. Two equivalent ways to state this rule are: M accepts the input x if (1) the probability of finding an accepting computation starting with x is greater than one half;⁸⁵ (2) the ratio of accepting computations starting with x to the total number of computations starting with x is greater than one half.

As an example we transform the non-deterministic algorithm solving SAT into a probabilistic algorithm testing whether a Boolean formula F is satisfied by more than half of the possible assignments:⁸⁶

```

input  $e(F)$ 
check that  $e(F)$  encodes a correct Boolean formula
for each variable  $x$  occurring in  $F$  do
    choose in a non-deterministic manner
     $F = F|_{x=0}$  or

```

⁸⁵Note that all computations have the same probability.

⁸⁶This problem is called MAJ.

```

 $F = F|_{x=1}$ 
simplify the resulting formula without variables
if the result is 1, then accept and halt
end

```

The above machine accepts exactly those Boolean formulae for which more than half of the possible computations are accepted. Each computation corresponds to a unique assignment satisfying F , thus F probabilistically accepts x if and only if F is satisfied by more than half of the possible assignments.

Let's turn our attention to the clear majority accepting rule, the one when the error probability is bounded. A non-deterministic machine M works with the clear majority rule if there exists an $\epsilon > 0$ such that M accepts the input x when more than half plus ϵ of the computations of M on x end in an accepting state. Equivalently: M accepts the input x if the probability of finding an accepting computation starting with x is greater than one half plus ϵ . Note that Rabin's test of primality is of this form as the error probability is at most 2^{-m} .

Probabilistic machines can be simulated by deterministic Turing machines. As one can expect, the fact that the full computation tree must be constructed results in a “blow up” of the running time of the simulation: the running time is exponential. So, we turn our attention to *polynomial time bounded probabilistic machines*, that is, probabilistic machines whose running times are polynomials. Every computation of such a machine halts in *exactly* $p(n)$ steps on inputs of length n . By PP⁸⁷ we denote the class of languages accepted by polynomial time probabilistic Turing machines. It is easy to see that (cf. Balcázar, Díaz, Gabarró [11]):

$$\text{NP} \subseteq \text{PP} \subseteq \text{PSPACE}.$$

Let BPP⁸⁸ denote the class of languages accepted by polynomial time probabilistic Turing machines whose error probability is bounded from above by some constant $\epsilon > 0$. It is clear that

$$\text{BPP} \subseteq \text{PP},$$

but it's open whether the inclusion is proper. The importance of BPP consists in the possibility of *iterating* the algorithm as many times as you need. The result is the reduction of the error probability; the error can be made as small as you need and the time cost for this reduction is reasonably low.

There is an alternative, equivalent, way to look at probabilistic Turing machines. The root corresponds to the machine's starting configuration, and each node corresponds to a different configuration reachable, with non-zero probability, from the configuration represented by its parent node, in

⁸⁷PP stands for Probabilistic Polynomial time.

⁸⁸BPP stands for Bounded error Probabilistic Polynomial time.

one computation step. Each edge from parent to child is associated with a probability, and the probability of following a particular path from the root to a node is the product of probabilities along the edge. Probabilities depend only on the configuration associated with the parent node, regardless of the node's position in the tree; they have to obey the *local probability condition* stating that

the sum of probabilities on edges leaving any single node is always one.

In this way we can associate a probability to each node: simply compute the product of probabilities assigned to the edges on the path from the root (having probability 1) to the node. This is the probability that a computation starting from the root reaches the node.

It may happen that at a certain level of a computation tree the same configuration duplicates, or even appears several times. The probability that a particular configuration is reached at a certain step in the computation is the sum of the probabilities of all nodes corresponding to that configuration at that step. For example, the probability of a particular final configuration is the sum of the probabilities of all leaf nodes corresponding to that configuration. So, if c_1, c_2, \dots, c_k are all distinct configurations at a certain level of the computation tree and p_1, p_2, \dots, p_k are their “global” probabilities of occurrence at that level, then the following *global probability condition* has to be satisfied:

$$\sum_{j=1}^k p_j = 1.$$

It is easy to show that if the local probability condition is satisfied, then the global probability condition is also satisfied.

4.20.2 Simon’s problem

As we have already mentioned, Deutsch and Jozsa [88] imagined a simple “promise problem” that can be solved “efficiently” without error on a quantum Turing machine, but, classically, one can perform very “inefficiently”. Unfortunately, this problem, as well as some other related ones suggested by various authors, can be efficiently solved by classical probabilistic Turing machines with exponential small error probability.

In 1994 Simon [270] imagined a simple problem that can be solved in polynomial time on a quantum Turing machine, but cannot be solved in polynomial time on *any* probabilistic Turing machine. Here is Simon’s problem: Suppose we are given a function $f : \{0,1\}^n \rightarrow \{0,1\}^n$ and we are promised that either f is one-to-one or there exists a non-trivial n -bit string s such that for all distinct n -bit strings x, x' we have

$$f(x) = f(x') \text{ if and only if } x' = x \oplus s,$$

that is, $f(x) = f(x')$ if and only if the bits of x and x' differ in exactly those positions where the bits of s are 1.

We wish to determine which of these two conditions holds for f , and, in the second case, to find s .

Let $a = (a_1, \dots, a_n), b = (b_1, \dots, b_n)$ be two n -bit strings regarded as n -bit vectors in the $\mathbf{Z}_2^n = \{0, 1\}^n$. We say that $a < b$ in case

$$\sum_{i=1}^n a_i 2^{i-1} > \sum_{i=1}^n b_i 2^{i-1}.$$

The inner product of a, b is

$$a \cdot b = \sum_{i=1}^n a_i b_i \pmod{2}.$$

A set $B \subset \{0, 1\}^n$ is linearly independent if for every $b \in B$ and every subset $B' \subset B \setminus \{b\}$ we have

$$(0, \dots, 0) \neq \bigoplus_{b' \in B'} b'.$$

Recall the quantum gate array U_f and the Walsh–Hadamard transformation W :

$$U_f(|x, y\rangle) = |x, f(x) \oplus y\rangle,$$

$$W(|x\rangle) = \frac{1}{\sqrt{2^n}} \sum_{y \in \{0, 1\}^n} (-1)^{x \cdot y} |y\rangle.$$

Simon's solution is the following. Use two quantum registers, both with n qubits and the initial states $|0, \dots, 0\rangle, |0, \dots, 0\rangle$. Then, apply the Walsh–Hadamard transformation on the first register, then apply U_f , then again the Walsh–Hadamard transformation on the first register, and, finally, observe the resulting pair of states to get a pair $(y, f(x))$. More formally, the algorithm can be presented in the following form:

$$\begin{aligned} |(0, \dots, 0), (0, \dots, 0)\rangle &\xrightarrow{W} \frac{1}{\sqrt{2^n}} \sum_{x \in \{0, 1\}^n} |x, (0, \dots, 0)\rangle \\ &\xrightarrow{U_f} \frac{1}{\sqrt{2^n}} \sum_{x \in \{0, 1\}^n} |x, f(x)\rangle \\ &\xrightarrow{W} \left(\frac{1}{\sqrt{2^n}}\right)^2 \sum_{x, y \in \{0, 1\}^n} (-1)^{x \cdot y} |y, f(x)\rangle \\ &= \frac{1}{2^n} \sum_{x, y \in \{0, 1\}^n} (-1)^{x \cdot y} |y, f(x)\rangle. \end{aligned}$$

If f is one-to-one, then f is bijective so all possible states $|y, f(x)\rangle$ are *distinct*, so the result of applying the above scheme $n - 1$ times consists of $n - 1$ pairs $(y_1, f(x_1)), \dots, (y_{n-1}, f(x_{n-1}))$, uniformly distributed over all pairs $(y, f(x))$.

However, if there is a non-trivial n -bit string s such that $f(x) = f(x')$ if and only if $x' = x \oplus s$, for all $x \neq x'$, then for each y, x we have

$$|y, f(x)\rangle = |y, f(x \oplus s)\rangle.$$

In this case we have:

$$\begin{aligned} & \frac{1}{2^n} \sum_{x,y \in \{0,1\}^n} (-1)^{x \cdot y} |y, f(x)\rangle \\ &= \frac{1}{2^{n+1}} \sum_{x,y \in \{0,1\}^n} \left((-1)^{x \cdot y} + (-1)^{(x \oplus s) \cdot y} \right) |y, f(x)\rangle \\ &= \frac{1}{2^{n+1}} \sum_{x,y \in \{0,1\}^n} (-1)^{x \cdot y} (1 + (-1)^{s \cdot y}) |y, f(x)\rangle \end{aligned}$$

Consequently, after $n - 1$ independent applications of the scheme we will get $n - 1$ independent pairs $(y_1, f(x_1)), \dots, (y_{n-1}, f(x_{n-1}))$, such that for every $1 \leq i \leq n - 1$, $y_i \cdot s \equiv 0 \pmod{2}$.

In both cases, after $n - 1$ repetitions of the scheme we will get $n - 1$ vectors y_i , $i = 1, 2, \dots, n - 1$. There are two possibilities according to whether the set $\{y_i \mid 1 \leq i \leq n - 1\}$ is linearly independent or not.

In the first case, the linear system of $n - 1$ equations $y_i \cdot s = 0$ can be solved in \mathbf{Z}_2 to obtain s . There are two cases:

- if f is one-to-one, then the solution s is irrelevant.
- if f is two-to-one, then the solution s is the one required by Simon's problem.

To distinguish between these two cases we need to compute and compare the values of $f(0, \dots, 0)$ and $f(s)$.

In the second case, that is when the set $\{y_i \mid 1 \leq i \leq n - 1\}$ is not linearly independent, we have to repeat the whole process. However, with probability at least $\frac{1}{4}$, the set of vectors $\{y_i \mid 1 \leq i \leq n - 1\}$ is linearly independent.⁸⁹ So, after an expected $O(n)$ repetitions, sufficiently many linearly independent vectors y_i will have been collected such that s is uniquely determined.

Consequently, the time of the computation is $O(nT_f(n) + G(n))$, where $T_f(n)$ is the time required to compute f on an n -bit string and $G(n)$ is the

⁸⁹If u is a non-zero n -bit vector, then by choosing $n - 1$ uniformly distributed n -bit vectors y_i , $1 \leq i \leq n$, such that $y_i \cdot u \equiv 0 \pmod{2}$ we obtain a linearly independent set of vectors $\{y_i \mid 1 \leq i \leq n\}$ with probability at least $\frac{1}{4}$.

time required to solve an $n \times n$ linear system of equations in \mathbf{Z}_2 . Can we do it equally better with a probabilistic Turing machine? The answer is negative as Simon [270] has shown. Let us construct an oracle, corresponding to a “hard probability distribution”, as follows: for each n uniformly generated two bit strings $s(n) \in \{0, 1\}^n, b(n) \in \{0, 1\}$. If $b(n) = 0$, then the function $f_n : \{0, 1\}^n \rightarrow \{0, 1\}^n$ is uniformly generated from the set of all permutations of $\{0, 1\}^n$; if $b(n) = 1$, then f_n is uniformly generated from the set of two-to-one functions such that $f_n(x) = f_n(x \oplus s(n))$, for all n -bit strings x . Then, any probabilistic Turing machine that queries the above oracle no more than $2^{-n/4}$ times cannot correctly guess $b(n)$ with probability greater than $\frac{1}{2} + 2^{-n/2}$. So,

any probabilistic Turing machine needs an exponential time to solve Simon’s problem on infinitely many inputs.

A recent paper by Hemaspaandra, Hemaspaandra and Zimand [130] shows that a variant of the Simon’s problem that is still solvable in quantum polynomial time needs on a classical machine an exponential time on almost every input.

Simon’s quantum algorithm works in polynomial time in the *expected time*⁹⁰ and there is no upper bound for the time required in the worst case. Brassard and Høyer [37] improved Simon’s algorithm (using Grover’s database search algorithm) and showed that Simon’s problem can be solved in polynomial time in the worst case. We will follow Mihara and Sung [191] to present a simpler polynomial time algorithm (in the worst case) for Simon’s problem.

First let f be as in Simon’s problem and let g be a non-zero n -bit vector different from s . The following quantum algorithm returns an n -bit vector y such that

$$s \cdot y = 0, \text{ and } g \cdot y = 1.$$

The idea is to use Simon’s algorithm with U_F instead of U_f , where F is an appropriately constructed function. To define F we construct two functions

$$\phi_g(x) = \max\{f(x), f(x \oplus g)\},$$

$$\psi_g(x) = \begin{cases} 0, & \text{if } f(x) > f(x \oplus g), \\ 1, & \text{otherwise,} \end{cases}$$

and we put

$$F(x, y) = (-1)^{\psi_g(x)} |x, \phi_g(x) \oplus y\rangle.$$

It is seen that $\phi_g(x) = \phi_g(x \oplus g)$, and $f(x) \neq f(x \oplus g)$ if and only if $\psi_g(x) \neq \psi_g(x \oplus g)$.

The algorithm is the following:

⁹⁰Recall, we need an expected $O(n)$ repetitions to collect the linearly independent vectors y_i .

$$\begin{aligned}
|(0, \dots, 0), (0, \dots, 0)\rangle &\xrightarrow{W} \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x, (0, \dots, 0)\rangle \\
&\xrightarrow{U_F} \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} (-1)^{\psi_g(x)} |y, \phi_g(x)\rangle \\
&\xrightarrow{W} \left(\frac{1}{\sqrt{2^n}} \right)^2 \sum_{x,y \in \{0,1\}^n} (-1)^{x \cdot y} (-1)^{\psi_g(x)} |y, \phi_g(x)\rangle \\
&= \frac{1}{2^{n+2}} \sum_{x,y \in \{0,1\}^n} (-1)^{x \cdot y} (-1)^{\psi_g(x)} ((-1)^{x \cdot y} \\
&\quad + (-1)^{(x \oplus s) \cdot y} - (-1)^{(x \oplus g) \cdot y} \\
&\quad + (-1)^{(x \oplus g \oplus s) \cdot y}) |y, \phi_g(x)\rangle \\
&= \frac{1}{2^{n+2}} \sum_{x,y \in \{0,1\}^n} (-1)^{x \cdot y} (1 + (-1)^{s \cdot y} \\
&\quad - (-1)^{g \cdot y} - (-1)^{(g \oplus s) \cdot y}) |y, \phi_g(x)\rangle \\
&= \frac{1}{2^{n+2}} \sum_{x,y \in \{0,1\}^n} (-1)^{x \cdot y} (1 + (-1)^{s \cdot y}) (1 - \\
&\quad (-1)^{g \cdot y}) |y, \phi_g(x)\rangle.
\end{aligned}$$

Hence, $(1 + (-1)^{s \cdot y})(1 - (-1)^{g \cdot y}) \neq 0$ if and only if $s \cdot y = 0$ and $g \cdot y = 1$. So, by measuring the first register we can obtain y such that $s \cdot y = 0$ and $g \cdot y = 1$. The total running time of the algorithm is $O(n + T_f(n))$, where $T_f(n)$ is the time required to compute f on an n -bit string.

We could solve Simon's problem if one could produce enough y 's such that $s \cdot y = 0$. The above algorithm shows how to produce such a y when we use a different, non-zero g . We next show that with *certitude* we can find a g for obtaining y . To this aim we need two simple mathematical facts:

- (1) There exists a polynomial time algorithm that for every linearly independent set $B \subset \{0,1\}^n$, returns a non-zero n -bit string $g \in \{0,1\}^n$ such that $g \cdot y = 0$, for every $y \in B$. If B has exactly $n - 1$ elements, then g is unique.
- (2) Let B be a linearly independent set and $g \in \{0,1\}^n$ such that $g \cdot y = 0$, for every $y \in B$. Then, for every y' such that $g \cdot y' = 1$, the set $B \cup \{y'\}$ is linearly independent.

Using the above facts we can write the following quantum algorithm:

Put $B = \emptyset$.

Select a non-zero n -bit vector g .

Repeat the following steps while $g \neq s$:

Use the above quantum algorithm to find a non-zero
 n -bit vector y such that $s \cdot y = 0$
 and $g \cdot y = 1$ and put $B = B \cup \{y\}$.

Use 1) to find a non-zero g' such that $g' \cdot y = 0$,
 for every $y \in B$ and set $g = g'$.

Return g .

In view of (2), the set B is linearly independent, therefore we can find a non-zero s such that $s \cdot y = 0$, for all $y \in B$. The running time of the algorithm is $O(n^2 + nT_f(n) + nG(n))$ in the worst case: here $T_f(n)$ is the time required to compute f on an n -bit string and $G(n)$ is the time required to produce g as in (1), a polynomial in n .

4.20.3 Complexity

A computation on a quantum Turing machine QTM as described by Deutsch [84] can be represented by a tree in much the same way we did it for probabilistic computation. The major change required by quantum mechanics is to replace probabilities with amplitudes. For complexity issues it is enough to consider only real amplitudes in the interval $[-1, 1]$. The amplitude of a node is the product of the amplitudes of the edges on the path from the root to that node. The amplitude of a configuration at any step in the computation is the sum of the amplitudes of all nodes corresponding to that configuration at the level in the tree corresponding to that step. When an observation is made, the probability associated with each configuration is not the configuration's amplitude in the superposition, but rather the squared magnitude of its amplitude. Hence, the probability of a configuration at any step is the square of its amplitude. For example, the probability of a configuration is the square of the sum of the amplitudes of all leaf nodes corresponding to that configuration. Some specific properties follow. For instance, a particular configuration c may correspond to two leaf nodes with conjugate amplitudes, α and $-\alpha$, and the probability of c being the final configuration will be zero. Still the parent nodes of these two nodes might both have non-zero probabilities.

The computation produces c with probability α^2 if the configuration of one leaf is different from the other. If both leaves have amplitude α , then the probability of c being the final configuration is $4\alpha^2$ (not just $2\alpha^2$). This mutual influence between branches is a consequence of quantum interference.

A quantum computation tree must obey the property that the sum of the probabilities of configurations at any level add up to one. Note that it is not enough to ask that for each node the sum of the squares of the amplitudes on edges leading to its children is one! Computation steps should be unitary, so reversible. A quantum computation results, in just one single step, in a superposition of all branches of its tree simultaneously.

A classical probabilistic computation tree has to be well-defined and local, with probabilities adding up to one. A quantum computation tree has to be well-defined, local, and unitary.

Quantum variations of time and complexity classes have been intensively studied. For time complexity, one-tape multitrack QTM are considered; for space complexity, off-line multitape QTM with one-way, read-only, input tape, a working tape, and one-way, write-only, out-tape are used. For time complexity it is enough to consider computations in which the measurement is done only after the machine halts; to study space complexity, a measurement is done each time a symbol is written on the output tape. Many variations of models and approaches have been considered; see Gruska [119].

There are no essential differences between conventional and quantum computation as concerns the space efficiency. Things are different for time complexity. Quantum versions of classes P and BPP are classes EQP and BQP. The class BQP, which is regarded as the class of languages (problems) that can be decided efficiently on QTMs, is defined as the family of languages L such that there exists a QTM that can decide, with probability at least $2/3$, for each string x whether $x \in L$.

The following basic relations hold true:

$$P \subseteq EQP \subseteq BQP,$$

and

$$BPP \subseteq BQP \subseteq PP \subseteq PSPACE.$$

It is an open problem to decide which inclusion is proper. Quantum complexity classes are intimately related to conventional complexity classes; in particular, showing that QTMs are more powerful than PTMs needs a breakthrough result in classical complexity theory.

4.21 Quantum Cryptography

Quantum systems can be used to achieve cryptographic tasks, such as secret (secure) communication. In cryptography (see, for example, Salomaa [263]) it is very difficult, if not impossible, to prove by experiment that a cryptographic protocol is secure: who knows whether an eavesdropper (spy, competitor) managed to beat the system? For example, the bit-commitment method, thought for a while to be secure through quantum methods, was proven to be insecure, cf. Mayers [186] and Lo, Chau [173].⁹¹ The only confidence one can hope to achieve relies on mathematical arguments, the so-called proofs of security.

⁹¹Cheating is possible through a clever use of quantum entanglement.

As usual in quantum mechanics scenarios, Alice and Bob are widely separated and wish to communicate. They are connected by an ordinary bi-directional open channel and a uni-directional quantum channel, directed from Alice to Bob. The quantum channel allows Alice to send single qubits (e.g. photons) to Bob who can measure their quantum state. An eavesdropper, Eve, is able to intercept and measure the qubits, then pass them on to Bob.

Given two orthonormal bases

$$\{|\uparrow\rangle, |\rightarrow\rangle\}, \{|\nwarrow\rangle, |\nearrow\rangle\},$$

where $|\nwarrow\rangle = \frac{1}{\sqrt{2}}(|\uparrow\rangle - |\rightarrow\rangle)$ and $|\nearrow\rangle = \frac{1}{\sqrt{2}}(|\uparrow\rangle + |\rightarrow\rangle)$, Alice and Bob can agree to associate $|\uparrow\rangle$ and $|\nwarrow\rangle$ with 0, $|\rightarrow\rangle$ and $|\nearrow\rangle$ with 1. For each bit, Alice pseudo-randomly uses one of these bases and Bob also pseudo-randomly selects a basis for measuring the received qubit. After the bits have been transmitted, Alice and Bob inform each other (using the open channel) of the basis they used to prepare and measure each qubit. In this way, they find out *when* they used the same basis, which happens on average half of the time, and retain only those results.

If Eve measures the qubits transmitted by Alice, then she uses the correct basis on average half of the time. Therefore, assuming that $2n$ qubits are sent by Alice, n bits are received by Bob without any disturbance. In $n/2$ cases, Bob will also use the correct basis.

What about the other n qubits sent by Alice? Since

$$|\nwarrow\rangle = \frac{1}{\sqrt{2}}(|\uparrow\rangle - |\rightarrow\rangle),$$

and

$$|\nearrow\rangle = \frac{1}{\sqrt{2}}(|\uparrow\rangle + |\rightarrow\rangle),$$

the probability to find a qubit represented by the state $|\nwarrow\rangle$ or $|\nearrow\rangle$ in the state $|x\rangle \in \{|\uparrow\rangle, |\rightarrow\rangle\}$ is

$$P = \left(\frac{1}{\sqrt{2}}\right)^2 + \left(\frac{1}{\sqrt{2}}\right)^2 = 1.$$

Taking into account that

$$|\uparrow\rangle = \frac{1}{\sqrt{2}}(|\nearrow\rangle + |\nwarrow\rangle) \text{ and } |\rightarrow\rangle = \frac{1}{\sqrt{2}}(|\nearrow\rangle - |\nwarrow\rangle),$$

the same result will be obtained by interchanging the two bases. Consequently, if n qubits are disturbed by Eve, then half of them are measured by Bob with the correct basis and $n/4$ qubits will be projected by Bob's measurement back onto original state. Eve's tentative interception disturbs

only a quarter of the message retained by Bob. Alice and Bob can now detect Eve's presence by pseudo-randomly choosing $n/2$ bits of the string and announcing over the open channel the values they have. If they agree on all these bits, then the probability that no eavesdropper was present is $(3/4)^{n/2}$. The undisclosed bits represent the “secret key”.

The above scenario is extremely “theoretical”, as it assumes that one possible strategy for Eve – she may deliberately not intercept all qubits (after all, she knows everything about quantum key distribution, doesn't she?). Noise may influence the trio “communication” as well. There are various subtle methods to address these issues, but we are not going to enter into details (see, for example, Gruska [119]). It is important to observe that variants of *the quantum key distribution are feasible with current technology*.

4.22 Information and Teleportation

4.22.1 Dense coding

Dense coding and quantum teleportation can be used to further illustrate the use of quantum gates. The initial scenario is the same for both processes: Alice and Bob wish to communicate. They use a pair of entangled qubits,

$$\psi_0 = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle).$$

Alice and Bob need never have communicated: just use a “facility” generating entangled pairs of qubits, and then sending one qubit to Alice and one qubit to Bob, which they store. Bennett and Wiesner [30] observed that Alice can communicate *two* classical bits by sending Bob just a *single* qubit, that is, her qubit of the entangled pair. This is the reason the method is called *dense coding*.

This entangled state can be obtained from $|00\rangle$ by applying the unitary transformation $C_{not} \circ (R_{\pi/4} \otimes I)$. We have

$$(R_{\pi/4} \otimes I)|00\rangle = R_{\pi/4}|0\rangle \otimes I|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$$

and

$$C_{not}(R_{\pi/4} \otimes I)|00\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle).$$

Alice keeps the first qubit and gives the second particle to Bob. So until a particle is transmitted, only Alice can perform transformations on her qubit, and only Bob can perform transformations on his.

Alice receives two classical bits, encoding the numbers 0, 1, 2 and 3. Depending on this number Alice performs one of the unitary transformations I , X , Y or Z , on her qubit of the entangled pair ψ_0 . If a single qubit transformation T acts on the first bit and the second one is left unchanged,

then the transformation $T \otimes I$ performs on the pair ψ_0 . The results for $(T \otimes I)|\psi_0\rangle$ are presented in Table 4.8.

Encoded value	T	New state $(T \otimes I) \psi_0\rangle$
0	I	$\psi_0 = \frac{1}{\sqrt{2}}(00\rangle + 11\rangle)$
1	X	$\psi_1 = \frac{1}{\sqrt{2}}(10\rangle + 01\rangle)$
2	Y	$\psi_2 = \frac{1}{\sqrt{2}}(- 10\rangle + 01\rangle)$
3	Z	$\psi_3 = \frac{1}{\sqrt{2}}(00\rangle - 11\rangle)$

Table 4.8: Application of $(T \otimes I)|\psi_0\rangle$.

Alice sends her qubit to Bob who applies the controlled-NOT transformation to the pair ψ_i ($i \in \{0, 1, 2, 3\}$). The resulting state is shown in Table 4.9.

Initial state	Result
$\psi_0 = \frac{1}{\sqrt{2}}(00\rangle + 11\rangle)$	$\frac{1}{\sqrt{2}}(00\rangle + 10\rangle) = \frac{1}{\sqrt{2}}(0\rangle + 1\rangle) \otimes 0\rangle$
$\psi_1 = \frac{1}{\sqrt{2}}(10\rangle + 01\rangle)$	$\frac{1}{\sqrt{2}}(11\rangle + 01\rangle) = \frac{1}{\sqrt{2}}(0\rangle + 1\rangle) \otimes 1\rangle$
$\psi_2 = \frac{1}{\sqrt{2}}(- 10\rangle + 01\rangle)$	$\frac{1}{\sqrt{2}}(- 11\rangle + 01\rangle) = \frac{1}{\sqrt{2}}(0\rangle - 1\rangle) \otimes 1\rangle$
$\psi_3 = \frac{1}{\sqrt{2}}(00\rangle - 11\rangle)$	$\frac{1}{\sqrt{2}}(00\rangle - 10\rangle) = \frac{1}{\sqrt{2}}(0\rangle - 1\rangle) \otimes 0\rangle$

Table 4.9: Application of controlled-NOT transformation.

Since the resulting state is not entangled, Bob can now measure the second qubit without disturbing the quantum state. If the measurement returns $|0\rangle$ then the encoded value was either 0 or 3. If the result of the measurement is $|1\rangle$, then the encoded value was either 1 or 2. Bob now applies the transformation H to the first qubit (see Table 4.10) and finally Bob measures the

resulting bit. Now he is able to distinguish between 0 and 3, respectively 1 and 2, as it is shown in Table 4.11.

Initial state	First bit	Result
ψ_0 or ψ_1	$\frac{1}{\sqrt{2}}(0\rangle + 1\rangle)$	$\frac{1}{\sqrt{2}}(\frac{1}{\sqrt{2}}(0\rangle + 1\rangle) + \frac{1}{\sqrt{2}}(0\rangle - 1\rangle)) = 0\rangle$
ψ_2 or ψ_3	$\frac{1}{\sqrt{2}}(0\rangle - 1\rangle)$	$\frac{1}{\sqrt{2}}(\frac{1}{\sqrt{2}}(0\rangle + 1\rangle) - \frac{1}{\sqrt{2}}(0\rangle - 1\rangle)) = 1\rangle$

Table 4.10: Application of transformation H .

First measurement	Second measurement	Encoded value
$ 0\rangle$	$ 0\rangle$	0
$ 0\rangle$	$ 1\rangle$	3
$ 1\rangle$	$ 0\rangle$	1
$ 1\rangle$	$ 1\rangle$	2

Table 4.11: Final results.

Dense coding is difficult to implement, so has very little practical utility (at least at the time of writing). However, it has the advantage of revealing in a particular simple way the relation between classical information, qubits and the information provided by a pair of entangled qubits.

4.22.2 Quantum teleportation

It is possible to transmit qubits without sending qubits!

What does this mean? Is it a pun? According to Bennett,⁹² “It’s a means by which you can take apart an unknown quantum state into classical information and purely quantum information, send them through two separate channels, put them back together, and get back the original quantum state”.

Teleportation, as it is commonly understood, is a fictional procedure of transferring an object from one location to another location in a three stage process: (a) dissociation, (b) information transmission, (c) reconstitution. The point is that, in contrast with fax transmission – where the original object remains intact at the initial location, only an approximate replica

⁹²A co-author of a 1993 paper that proposed quantum teleportation, [28].

is constructed at destination,⁹³ in teleportation the original object is destroyed after enough information about it has been extracted, the object is not traversing in any way the space between locations, but it is reconstructed, as an exact replica, at the destination.

Quantum teleportation allows for the transmission of quantum information to a distant location. The objective is to transmit the quantum state of a particle using classical bits and reconstruct the state at the receiver.

Let's assume that Alice wishes to communicate with Bob a single qubit in an unknown state $\varphi = a|0\rangle + b|1\rangle$; she wants to make the transmission through classical channels. Alice cannot know with certainty the state as any measurement she may perform may change it; she cannot clone it because of the no cloning result! So, it seems that the only way to send Bob the qubit is to send him the *physical qubit*, or to swap the state into another quantum system and then send Bob that system.

As with dense coding, they use an entangled pair

$$\psi_0 = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle).$$

Alice controls the first half of the pair and Bob controls the second one. The input state is

$$\begin{aligned} \varphi \otimes \psi_0 &= (a|0\rangle + b|1\rangle) \otimes \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \\ &= \frac{1}{\sqrt{2}}(a|0\rangle \otimes |00\rangle + a|0\rangle \otimes |11\rangle + b|1\rangle \otimes |00\rangle + b|1\rangle \otimes |11\rangle) \\ &= \frac{1}{\sqrt{2}}(a|000\rangle + a|011\rangle + b|100\rangle + b|111\rangle). \end{aligned}$$

Alice now applies the transformation $(H \otimes I \otimes I) \circ (C_{not} \otimes I)$ to this state. The third bit is left unchanged; only the first two bits belong to Alice and the rightmost one belongs to Bob.

Applying now $H \otimes I \otimes I$, we have:

$$\begin{aligned} &(H \otimes I \otimes I) \circ (C_{not} \otimes I)(\varphi \otimes \psi_0) \\ &= \frac{1}{\sqrt{2}}H \otimes I \otimes I(a|000\rangle + a|011\rangle + b|110\rangle + b|101\rangle) \\ &= \frac{1}{\sqrt{2}}(aH|0\rangle \otimes (I \otimes I)|00\rangle + aH|0\rangle \otimes (I \otimes I)|11\rangle \\ &\quad + bH|1\rangle \otimes (I \otimes I)|10\rangle + bH|1\rangle \otimes (I \otimes I)|01\rangle) \\ &= \frac{1}{\sqrt{2}}(a\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |00\rangle + a\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |11\rangle) \end{aligned}$$

⁹³At the end, two “identical” versions of the original object result.

$$= \frac{1}{2}(a(|000\rangle + |100\rangle + |011\rangle + |111\rangle) + b(|010\rangle - |110\rangle + |001\rangle - |101\rangle)).$$

This state may be re-written by regrouping terms:

$$\begin{aligned} & (H \otimes I \otimes I) \circ (C_{not} \otimes I)(\varphi \otimes \psi_0) \\ = & \frac{1}{2}(|00\rangle(a|0\rangle + b|1\rangle) \\ & + |01\rangle(a|1\rangle + b|0\rangle) + |10\rangle(a|0\rangle - b|1\rangle) \\ & + |11\rangle(a|1\rangle - b|0\rangle)). \end{aligned}$$

Alice then measures her two qubits, obtaining four possible results: $|00\rangle$, $|01\rangle$, $|10\rangle$, or $|11\rangle$ with equal probability $1/4$. Depending on the result of the measurement, the quantum state of Bob's qubit is projected to $a|0\rangle + b|1\rangle$, $a|1\rangle + b|0\rangle$, $a|0\rangle - b|1\rangle$, $a|1\rangle - b|0\rangle$, respectively. Alice sends the result of her measurement as two classical bits to Bob. He will know what has happened, and can apply the decoding transformation $T \in \{I, X, Y, Z\}$ to fix his qubit.

Received bits	State	Transformation	Result
00	$a 0\rangle + b 1\rangle$	I	$a 0\rangle + b 1\rangle$
01	$a 1\rangle + b 0\rangle$	X	$a 0\rangle + b 1\rangle$
10	$a 0\rangle - b 1\rangle$	Z	$a 0\rangle + b 1\rangle$
11	$a 1\rangle - b 0\rangle$	Y	$a 0\rangle + b 1\rangle$

Table 4.12: An illustration of quantum teleportation.

The final output state is $\varphi = a|0\rangle + b|1\rangle$, which, as desired, is the unknown qubit that Alice wanted to send.

Recently, important teleportation experiments have been performed in Innsbruck and Caltech: for a recent report see Bouwmeester, Pan, Weinfurter, Zeilinger [35]. There is a lot of controversy about the nature of quantum teleportation and what criteria should be met by a successful experiment. The following criteria for evaluating a quantum teleportation procedure have been proposed in [35]:

- How well can it teleport any arbitrary quantum state it is intended to teleport? (fidelity of teleportation)
- How often does it succeed to teleport, when it is given an input state within the set of states it is designed to teleport? (efficiency of teleportation)

- If given a state the scheme is not intended to teleport, how well does it reject such a state? (cross-talk rejection efficiency)

Let us close this section with another controversial statement of the same Bennett: “I think it’s quite clear that anything approximating teleportation of complex living beings, even bacteria, is so far away technologically that it’s not really worth thinking about it.”

4.23 Computing the Uncomputable?

One fundamental result of theoretical computer science is Alan Turing’s proof (in [289]) that it is undecidable to determine whether a computer program will halt or not. This is formally known as the *halting problem*. We can restrict our attention to Turing machines since they are equivalent in computational power to any “conventional” computer [41, 16]. In what follows we present an attempt to trespass the Turing barrier (see Calude, Dinneen and Svozil [50]). The method discussed might in principle allow to “solve” the halting problem (for another proposal see Mitchison and Josza [195]; for an approach based on counterfactual computation (cf. [297]) see [49]). Thereby we are well aware of the fact that for all practical purposes (Bell [20]) this goal will remain unreachable, at least within Quantum Computing.

Assume that it is possible to design a *halting qubit* which indicates whether a computation has actually reached a state associated with a halting condition. Assume further that the halting qubit starts in its non-halting state and, since the evolution is unitary, the buildup of the amplitude is continuous in time.

In such a case, the halting qubit acquires a halting component which is non-zero even in *finite time*. Therefore, a detection of a halting computation at small time scales is conceivable even if the associated classical computation lasts “very” long. The price to be paid is the “very small” amplitude and, associated with it, a corresponding chance of detection.

To be a little bit more precise we need some rudiments of algorithmic information theory (see Chaitin [57, 60], Calude [42]). We will work with programs with no input which produce binary strings as outputs. For any n we denote by P_n a program of length n that halts and produces the longest string among all outputs produced by all programs of length n that eventually stop. We denote by $\Sigma(n)$ the length of the output produced by P_n . Here Σ is the busy beaver function [247, 58]: it grows faster than every computable function of n . Let H be the program-size complexity, that is the length of the smallest universal program generating a binary string.

Assume that any program which halts requires a running time at least proportional to the length of its output. If an n -bit program p halts, then the time t it takes to halt satisfies $H(t) \leq n + c$. So if p has run for time T without halting, and T has the property that $t \geq T \implies H(t) > n + c$, then

p will never halt. This shows that the running times of the programs in the sequence $P_1, P_2, \dots, P_n, \dots$ grow faster than any computable function.

We are now ready to present the argument. Let us assume the halting qubit is represented by

$$|Halt\rangle = c_h(t)|h\rangle + c_n(t)|n\rangle,$$

where $|h\rangle$, $|n\rangle$ represent the halting state and non-halting state and $c_h(t)$, $c_n(t)$ are time dependent amplitudes thereof, respectively.

Initially, let $|c_h(t)| = |c_n(t)| - 1 = 0$. As a worst-case scenario derived from the above analysis, for a linear buildup of the amplitude we obtain

$$|c_h(t)|^2 \propto (\Sigma(H(n) + O(1)))^{-1}.$$

The setup of a detection of $|Halt\rangle$ is a simple transmission measurement of the halting qubit. Although the buildup may be very slow, there is a non-vanishing chance to obtain a solution of the halting problem in finite time. Of course, the solution is probabilistic (one can argue that all mathematical proofs or computer programs are ultimately probabilistic, see Davis [80], De Millo, Lipton and Perllis [83]), but goes beyond the capability of any classical computation: even the best probabilistic algorithms are not able to achieve this computational power (by a classical result [82], probabilistic algorithms are equivalent to Turing machines).

Let us finally notice that by virtue of the same information-theoretic argument, the possibility of time-travel (see, for example, Nahin [197]) would not solve the halting problem unless one could travel back and at a pace exceeding the growth of any computable function.

4.24 Bibliographical Notes

We made no attempt to be comprehensive in any sense of the word.

Feynman [98] contains a reprint of the lecture “Quantum Mechanical Computers” [96] which began the field of Quantum Computing; an important continuation is [134] which includes papers written by many authors who pioneered the field.

Williams and Clearwater’s book [296] is the first monograph on the subject; Mathematica programs simulate a few quantum algorithms including Shor’s algorithm. Other books of interest are Milburn [193] and Berman, Doolen, Mainieri, Tsifrinovich [31]. Calude, Dinneen and Casti [47], Brooks [39], Lo, Spiller, Popescu [174], Macchiavello, Palma, Zeilinger [176] contain recent papers in the subject. The most comprehensive treatment of Quantum Computing is Gruska [119].

The articles referenced in this chapter, and many more, have been announced at the Los Alamos preprint server:

<http://xxx.lanl.gov/archive/quant-ph>.

Most papers on Quantum Computing can be found on the web, for example at the Caltech-MIT-USC Quantum Information and Computation Project

<http://theory.caltech.edu/quic/index.html>,

the Centre for Quantum Computation at Oxford University

<http://www.qubit.org/>,

the Quantum Computation-Cryptography at Los Alamos

<http://qso.lanl.gov/qc/>,

the Southwest Quantum Information and Technology (SQuInT) Network

<http://www.squint.org/>.

These sites have a fair amount of information plus lots of links to other sites of interest.

Ömer's Quantum Computation Language can be found at
<http://tph.tuwien.ac.at/oemer/>. Senko Corporation offers a Quantum Computing Simulator: <http://www.senko-corp.co.jp>.

Chapter 5

Final Remarks

The idea of unconventional Computing has fired many imaginations and many researchers regard it as a new revolution in information processing. The typical advertisement for unconventional Computing includes two items: (a) miniaturization, as a basis for massive parallelism and huge data bases, (b) Moore's law.¹ The fact that some day the gate is going to be too small for the carrier to tunnel through does not imply that the only solution will come from biology or quantum mechanics. But even the hypothesis that computational components will get smaller and smaller is not indisputable! According to Landauer [162] miniaturization is slowing down for economic not physical reasons (of course, the semiconductor industry will not remain still, unconventional Computing has to fight a moving target) and attention could move together new directions, e.g., three dimensional integration. What will happen in 5, 10 or 20 years? People have drastically different opinions. For example, Dennis Bushill, chief scientist at Langley Research Centre of NASA is cited² on the front page of the site <http://www.eiqc.org/> as saying that

NASA are now planning on the basis that Quantum Computing will be mainstream within five years.

In February 1999, during the constituting meeting of the European Molecular Computing Consortium³, Erik Winfree, one of the first computer scientists who have completed a PhD dissertation in DNA Computing, declared that he expects that in a few years real life applications will be reported – maybe based on his two-dimensional self-assembly procedure. The most promising area is cryptography, where some degree of error is acceptable.⁴

For Landauer [162],

¹The energetic efficiency is also mentioned with respect to bio-computers, but this criterion is not an urgent one.

²On 28 November 1999.

³Leiden, The Netherlands.

⁴The problem with cryptography is that we may learn about applications only ... a dozen of years later, when the matter is no longer classified...

... the thrust of much of what has been discussed in quantum computation is pet inventions... I think that there are important questions that get suppressed by premature pitching of what we might be able to do in five or ten years with various approaches.

We tend to agree with the last opinion. For example, Quantum Computing is not going to replace classical Computing for similar reasons that quantum physics does not replace classical physics: no one takes into consideration Heisenberg in order to build a house, and cars are not mended by quantum mechanical garages. If large quantum/DNA/membrane computers are ever constructed, they will probably be used to address just special tasks which benefit from quantum/DNA/membrane information processing, in an intertwined way with conventional methods.

There are several more solid reasons to be excited about unconventional Computing, not directly dealing with applications. We mention some of them: (a) new methods to design algorithms for conventional computers or hybrid computers (for comparison, remember the success of neural networks and genetic algorithms, both of them inspired from nature/life and implemented *in silicon*), (b) new and insightful ways to think about the fundamental laws of physics, (c) new measures of information, (d) new methods of data encryption and security, (e) the challenge of Turing barrier, (f) the cultural quest to bring together Turing machines, information, biology, number theory and quantum physics. At least from the point of view of fundamental research, of general human knowledge, unconventional Computing is already a successful adventure of the spirit.

Bibliography

- [1] L. M. Adleman, Molecular computation of solutions to combinatorial problems, *Science*, 226 (November 1994), 1021–1024.
- [2] L. M. Adleman, On constructing a molecular computer, in [170], 1–22.
- [3] L. M. Adleman, P. W. K. Rothemund, S. Roweiss, E. Winfree, On applying molecular computation to the Data Encryption Standard, in [18], 31–44.
- [4] A. V. Aho, J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Prentice Hall, Englewood Cliffs, N.J., Vol. I: 1971, Vol. II: 1973.
- [5] D. Z. Albert, On quantum-mechanical automata, *Physical Letters*, A 98 (1983), 249–252.
- [6] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, J. D. Watson, *Molecular Biology of the Cell*, 3rd ed., Garland Publishing, New York, 1994.
- [7] R. B. Altman, A. K. Dunker, L. Hunter, T. E. Klein, eds., *Pacific Symp. on Biocomputing*, Hawaii, 1998, World Sci., Singapore, 1998.
- [8] M. Amos, *DNA Computing*, PhD Thesis, Univ. of Warwick, Dept. of Computer Sci., 1997.
- [9] M. Amos, A. Gibbons, D. Hodgson, Error-resistant implementation of DNA computations, in L. F. Landweber, E. B. Baum, eds., *DNA Based Computers II*, American Mathematical Society, 1998, 151–161.
- [10] M. Amos, S. Wilson, D. A. Hodgson, G. Owenson, A. Gibbons, Practical implementation of DNA computations, in [47], 1–18.
- [11] J. L. Balcázar, J. Díaz, J. Gabarró, *Structural Complexity I*, Springer-Verlag, Berlin, 1988.

- [12] J. P. Banâtre, A. Coutant, D. Le Metayer, A parallel machine for multi-set transformation and its programming style, *Future Generation Computer Systems*, 4 (1988), 133–144.
- [13] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolous, P. W. Schnor, T. Sleator, J. A. Smolin, H. Weinfurter, Elementary gates of quantum computation, *Physical Review A* 52 (1995), 3457–3467.
- [14] A. Barenco, *Quantum Computation*, PhD Thesis, Oxford University, 1996.
- [15] J. M. Barreiro, J. Rodrigo, A. Rodriguez-Paton, Evolutionary biomolecular computing, *Romanian J. of Information Sci. and Technology*, 1, 4 (1998), 289–294.
- [16] J. Barrow, *Impossibility—The Limits of Science and the Science of Limits*, Oxford University Press, Oxford, 1998.
- [17] E. B. Baum, A DNA associative memory potentially larger than the brain, in [170], 23–28.
- [18] E. Baum, D. Boneh, P. Kaplan, R. Lipton, J. Reif, N. Seeman, eds., *DNA Based Computers*, Proc. of the Second Annual Meeting, Princeton, 1996.
- [19] D. Beckman, A. N. Shari, S. Devabhaktuni, J. Preskill, Efficient networks for quantum factoring, *Physical Review A*, 54 (1996), 1034–1063.
- [20] J. S. Bell, On the Einstein Podolsky Rosen paradox, *Physics*, 1 (1964), 195–200. Reprinted in [291], 403–408, and in [21], 14–21.
- [21] J. S. Bell, *Speakable and Unspeakable in Quantum Mechanics*, Cambridge University Press, Cambridge, 1987.
- [22] P. Benioff, The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines, *Journal of Statistical Physics*, 22 (1980), 563–591.
- [23] C. H. Bennett, Logical reversibility of computation, *IBM J. Res. Dev.*, 17 (1973), 525–532.
- [24] C. H. Bennett, The thermodynamics of computation, *International Journal of Theoretical Physics*, 21 (1982), 905–940.
- [25] C. H. Bennett, Demons, engines and the second law, *Scientific American*, November (1987), 108–116.
- [26] C. H. Bennett, Notes on the history of reversible computation, *IBM J. Res. Dev.*, 32 (1988), 281–288.

- [27] C. H. Bennett, Time/space trade-offs for reversible computation, *SIAM J. on Computing*, 18 (1989), 766–776.
- [28] C. H. Bennett, G. Brassard, C. Crepeau, R. Jozsa, A. Peres, W. K. Wootters, Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels, *Phys. Rev. Lett.*, 70 (1993), 1895–1898.
- [29] C. H. Bennett, R. Landauer, The fundamental physical limits of computation, *Scientific American*, July (1985), 48–56.
- [30] C. H. Bennett, S. J. Wiesner, Communication via one- and two-particle operations on Einstein-Podolsky-Rosen states, *Phys. Rev. Lett.*, 69 (1992), 2881–2884.
- [31] G. P. Berman, G. D. Doolen, R. Mainieri, V. I. Tsifrinovich, *Introduction to Quantum Computers*, World Scientific, Singapore, 1993.
- [32] G. Berry, G. Boudol, The chemical abstract machine, *Theoretical Computer Sci.*, 96 (1992), 217–248.
- [33] G. Birkhoff, J. von Neumann, The logic of quantum mechanics, *Annals of Mathematics*, 37(4) (1936), 823–843.
- [34] D. Boneh, C. Dunworth, R. J. Lipton, Breaking DES using a molecular computing, in [170], 37–66.
- [35] D. Bouwmeester, J.-W. Pan, H. Weinfurter, A. Zeilinger, High-fidelity teleportation of independent qubits, *quant-ph/9910043*.
- [36] G. Brassard, Searching a quantum phone book, *Science*, 31 January (1997), 627–628.
- [37] G. Brassard, P. Høyer, An exact quantum polynomial-time algorithm for Simon’s problem, *Proceedings of Israeli Symposium on Theory of Computing and System*, 1997, 12–23.
- [38] W. Brauer, *Automatentheorie*, Teubner, Stuttgart, 1984.
- [39] M. Brooks, ed., *Quantum Computing and Communications*, Springer-Verlag, Berlin, 1999.
- [40] V. Bužek, S. L. Braunstein, M. Hillery, D. Bruß, Quantum copying: a network, *Physical Review A*, 56,5 (1997), 3446–3452.
- [41] C. Calude, *Theories of Computational Complexity*, North-Holland, Amsterdam, 1988.
- [42] C. Calude, *Information and Randomness. An Algorithmic Perspective*, Springer-Verlag, Berlin, 1994.

- [43] C. S. Calude, E. Calude, K. Svozil, S. Yu, Physical versus computational complementarity I, *International Journal of Theoretical Physics*, 36, 7 (1997), 1495–1523.
- [44] C. S. Calude, E. Calude, K. Svozil. Quantum correlations conundrum: An automaton-theoretic approach, in G. Păun, ed., *Recent Topics in Mathematical and Computational Linguistics*, Romanian Academy Publishing Company, Bucharest, 2000, in press.
- [45] C. S. Calude, J. L. Casti, Parallel thinking, *Nature* 392, 9 April (1998), 549–551.
- [46] C. S. Calude, J. L. Casti, Silicon, molecules, or photons? *Complexity*, 4, 1 (1998), 13.
- [47] C. S. Calude, J. Casti, M. J. Dinneen, eds., *Unconventional Models of Computation*, Springer-Verlag, Singapore, 1998.
- [48] C. S. Calude, G. J. Chaitin, Randomness everywhere, *Nature*, 400, 22 July (1999), 319–320.
- [49] C.S. Calude, M. J. Dinneen, K. Svozil, Counterfactual Effect, the Halting Problem, and the Busy Beaver Function (Preliminary Version), *CDMTCS Research Report* 107, 1999, (www.cs.auckland.ac.nz/CDMTCS).
- [50] C. S. Calude, M. J. Dinneen, K. Svozil, Reflections on Quantum Computing, *CDMTCS Research Report* 130, 2000, (www.cs.auckland.ac.nz/CDMTCS).
- [51] C. S. Calude, P. H. Hertling, K. Svozil, Embedding quantum universes into classical ones, *Foundations of Physics*, 29, 3 (1999), 349–379.
- [52] C. S. Calude, J. Hromkovic, Complexity: A language-theoretic point of view, in [258], Vol. 2, 1–60.
- [53] C. S. Calude, M. Lipponen, Computational complementarity and sofic shifts, in X. Lin, ed., *Theory of Computing 98, Proceedings of the 4th Australasian Theory Symposium, CATS'98*, Springer-Verlag, Singapore, 1998, 277–290.
- [54] C. S. Calude, Gh. Păun, Global syntax and semantics for recursively enumerable languages, *Fundamenta Informaticae*, 4, 2 (1981), 245–254.
- [55] C. S. Calude, F. W. Meyerstein, Is the universe lawful? *Chaos, Solitons & Fractals*, 10, 6 (1999), 1075–1084.
- [56] E. Calude, *Automata-Theoretic Models for Computational Complementarity*, PhD Thesis, The Univ. of Auckland, 1998.

- [57] G. J. Chaitin, *Information, Randomness and Incompleteness, Papers on Algorithmic Information Theory*, World Scientific, Singapore, 1987. (2nd ed., 1990).
- [58] G. J. Chaitin, A. Arslanov, C. Calude, Program-size complexity computes the halting problem, *Bulletin of the EATCS* 57 (1995), 198–200.
- [59] G. J. Chaitin, *The Limits of Mathematics*, Springer-Verlag, Singapore, 1997.
- [60] G. J. Chaitin, *The Unknowable*, Springer-Verlag, Singapore, 1999.
- [61] J. F. Clauser, A. Shimony, Bell’s theorem: experimental tests and implications, *Rep. Prog. Phys.*, 41 (1978), 1821–1927.
- [62] D. W. Cohen, *An Introduction to Hilbert Space and Quantum Logic*, Springer-Verlag, New York, 1989.
- [63] M. Conrad, Information processing in molecular systems, *Currents in Modern Biology*, 5 (1972), 1–14.
- [64] M. Conrad, The price of programmability, in R. Herken, ed., *The Universal Turing Machine: A Half-Century Survey*, Kammerer and Unverzagt, Hamburg, 1988, 285–307.
- [65] J. H. Conway, *Regular Algebra and Finite Machines*, Chapman and Hall, London, 1971.
- [66] J. W. Cooley, J. W. Tukey, An algorithm for the machine calculation of complex Fourier series, *Math. Comput.*, 19 (1965), 297–301.
- [67] D. Coppersmith, An approximate Fourier transform useful in quantum factoring, *IBM Research Report No. RC19642*, 1994.
- [68] E. Csuha-j-Varju, J. Dassow, On cooperating distributed grammar systems, *J. Inf. Process. Cybern., EIK*, 26, 1–2 (1990), 49–63.
- [69] E. Csuha-j-Varju, J. Dassow, J. Kelemen, Gh. Păun, *Grammar Systems. A Grammatical Approach to Distribution and Cooperation*, Gordon and Breach, London, 1994.
- [70] E. Csuha-j-Varju, L. Kari, Gh. Păun, Test tube distributed systems based on splicing, *Computers and AI*, 15, 2–3 (1996), 211–232.
- [71] E. Csuha-j-Varju, A. Salomaa, Networks of language processors. Parallel communicating systems, *Bulletin of the EATCS*, 66 (October 1998), 122–138.
- [72] K. Culik II, A purely homomorphic characterization of recursively enumerable sets, *Journal of the ACM*, 26 (1979), 345–350.

- [73] K. Culik II, T. Harju, Splicing semigroups of dominoes and DNA, *Discrete Appl. Math.*, 31 (1991), 261–277.
- [74] J. Dassow, V. Mitrana, Splicing grammar systems, *Computers and AI*, 15, 2–3 (1996), 109–122.
- [75] J. Dassow, Gh. Păun, *Regulated Rewriting in Formal Language Theory*, Springer-Verlag, Berlin, 1989.
- [76] J. Dassow, Gh. Păun, On the power of membrane computing, *J. Universal Computer Sci.*, 5, 2 (1999), 33–49, (<http://www.iicm.edu/jucs>).
- [77] J. Dassow, Gh. Păun, Concentration controlled P systems, submitted, 1999.
- [78] M. Davis, *Computability and Unsolvability*, McGraw-Hill, New York, 1958.
- [79] M. D. Davis, E. J. Weyuker, *Computability, Complexity, and Languages*, Academic Press, New York, 1983.
- [80] P. J. Davis, Fidelity in mathematical discourse: Is one and one really two? *Amer. Math. Monthly*, 79 (1972), 252–263.
- [81] M. T. Dawson, R. Powell, F. Gannon, *Gene Technology*, BIOS Scientific Publishers, Oxford, 1996.
- [82] K. De Leeuw, E. F. Moore, C. E. Shannon, N. Shapiro. Computability by probabilistic machines, in C. E. Shannon, J. McCarthy (eds.), *Automata Studies*, Princeton University Press, Princeton, N.J., 1956, 183–212.
- [83] R. De Millo, R. Lipton, A. Perlis, Social processes and proofs of theorems and programs, *Comm. ACM*, 22 (1979), 271–280.
- [84] D. Deutsch, Quantum theory, the Church-Turing principle and the universal quantum computer, *Proceedings of the Royal Society London*, A 400 (1985), 97–119.
- [85] D. Deutsch, Quantum computation, *Physics World*, 5 (1992), 57–61.
- [86] D. Deutsch, *The Fabric of Reality*, Allen Lane, Penguin Press, 1997.
- [87] D. Deutsch, A. Barenco, A. Ekert, Universality in quantum computation, *Proceedings of the Royal Society of London*, A 449 (1995), 669–677.
- [88] D. Deutsch, R. Josza, Rapid solution of problems by quantum computation, *Proceedings of the Royal Society London*, A 439 (1992), 553–558.

- [89] D. Dieks, Communication by EPR devices, *Physical Letters A*, 92 (1982), 271–272.
- [90] P. Dirac, *The Principles of Quantum Mechanics*, 4th ed., Oxford University Press, Oxford, 1958.
- [91] A. Einstein, B. Podolsky, N. Rosen, Can quantum-mechanical description of physical reality be considered complete? *Physical Review*, 47 (1935), 777–780. Reprinted in [291], 138–141.
- [92] J. Engelfriet, Reverse twin-shuffles, *Bulletin of the EATCS*, 60 (1996), 144.
- [93] J. Engelfriet, G. Rozenberg, Fixed point languages, equality languages, and representations of recursively enumerable languages, *Journal of the ACM*, 27 (1980), 499–518.
- [94] A. Ekert, R. Jozsa, Shor’s quantum algorithm for factoring numbers, *Rev. Modern Physics* 68 (3), (1996), 733–753.
- [95] R. P. Feynman, Simulating physics with computers, *International Journal of Theoretical Physics*, 11 (1985), 11–20.
- [96] R. P. Feynman, Quantum mechanical computers, *Optics News* 21 (1982), 467–488.
- [97] R. P. Feynman, in D. H. Gilbert, ed., *Miniatrization*, Reinhold, New York, 1961, 282–296.
- [98] *Feynman Lectures on Computation*, J. G. Hey and R. W. Allen, eds., Addison-Wesley, Reading, Massachusetts, 1996.
- [99] D. Finkelstein, S. R. Finkelstein, Computational complementarity, *International Journal of Theoretical Physics*, 22, 8 (1983), 753–779.
- [100] D. J. Foulis, C. H. Randall, Operational statistics. I. Basic concepts, *Journal of Mathematical Physics*, 13 (1972), 1667–1675.
- [101] E. Fredkin, T. Toffoli, Conservative logic, *International Journal of Theoretical Physics*, 21 (1982), 219–253.
- [102] R. Freund, Generalized P systems, *Proc. of FCT’99*, Iași, Romania (G. Ciobanu, Gh. Păun, eds.), *Lecture Notes in Computer Science*, 1684, Springer-Verlag, 1999, 281–292.
- [103] R. Freund, Generalized P systems with splicing and cutting/recombination, *Grammars*, 2, 3 (1999), 189–199.
- [104] R. Freund, L. Kari, Gh. Păun, DNA computing based on splicing. The existence of universal computers, *Theory of Computing Systems*, 32 (1999), 69–112.

- [105] R. Freund, Gh. Păun, G. Rozenberg, A. Salomaa, Bidirectional sticker systems, in [7], 535–546.
- [106] R. Freund, Gh. Păun, G. Rozenberg, A. Salomaa, Watson-Crick finite automata, in [300], 305–317.
- [107] R. Freund, Gh. Păun, G. Rozenberg, A. Salomaa, Watson-Crick automata, *Techn. Report 97-13*, Dept. of Computer Sci., Leiden Univ., 1997.
- [108] B. Fu, R. Beigel, On molecular approximation algorithms for NP-optimization problems, in [300], 93–101.
- [109] T. J. Fu, N. C. Seeman, DNA double-crossover molecules, *Biochemistry*, 32 (1993), 3211–3220.
- [110] V. Geffert, Normal forms for phrase-structure grammars, *RAIRO. Th. Inform. and Appl.*, 25 (1991), 473–496.
- [111] G. Georgescu, On the generative capacity of splicing grammar systems, in [228], 330–345.
- [112] A. Gibon, M. Amos, D. Hodgson, Models of DNA computing, *Proc. of 21st MFCS Conf.*, 1996, Cracow, *Lect. Notes in Computer Sci.* 1113, Springer-Verlag, Berlin, 1996, 18–36.
- [113] S. Ginsburg, *The Mathematical Theory of Context-free Languages*, McGraw Hill, New York, 1966.
- [114] R. Giuntini, *Quantum Logic and Hidden Variables*, BI Wissenschaftsverlag, Mannheim, 1991.
- [115] T. Gramss, S. Bornholdt, M. Gross, M. Mitchell, Th. Pellizzari, *Non-Standard Computation. Molecular Computation, Cellular Automata, Evolutionary Algorithms, Quantum Computers*, Willey-VCH, Weinheim, New York, etc., 1998.
- [116] D. M. Greenberger, M. A. Horne, A. Zeilinger, Going beyond Bell’s theorem, in M. Kafatos, ed., *Bell’s Theorem, Quantum Theory, and Conceptions of the Universe*, Kluwer Academic Publishers, Dordrecht, 1989, 73–76.
- [117] D. M. Greenberger, A. YaSin, “Haunted” measurements in quantum theory, *Foundation of Physics*, 19, 6 (1989), 679–704.
- [118] L. K. Grover, A fast quantum mechanical algorithm for database search, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, 1996, 212–219.
- [119] J. Gruska, *Quantum Computing*, McGraw-Hill, London, 1999.

- [120] F. Guarnieri, M. Fliss, C. Bancroft, Making DNA add, *Science*, 273 (July 1996), 220–223.
- [121] V. Gupta, S. Parthasarathy, M. J. Zaki, Arithmetic and logic operations with DNA, in [300], 212–220.
- [122] G. H. Hardy, E. M. Wright, *An Introduction to the Theory of Numbers*, Oxford University Press, Oxford (4th edition), 1965.
- [123] M. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, Mass., 1978.
- [124] J. Hartmanis, About the nature of computer science, *Bulletin of the EATCS*, 53 (June 1994), 170–190.
- [125] J. Hartmanis, On the weight of computation, *Bulletin of the EATCS*, 55 (1995), 136–138.
- [126] H. Havlicek, K. Svozil, Density conditions for quantum propositions, *Journal of Mathematical Physics*, 37(11) (1996), 5337–5341.
- [127] T. Head, Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors, *Bulletin of Mathematical Biology*, 49 (1987), 737–759.
- [128] T. Head, Hamiltonian paths and double stranded DNA, in [217], 80–92.
- [129] T. Head, Gh. Păun, D. Pixton, Language theory and molecular genetics. Generative mechanisms suggested by DNA recombination, in [258], Vol. 2, 295–360.
- [130] E. Hemaspaandra, L. A. Hemaspaandra, M. Zimand, Almost-everywhere superiority for quantum polynomial time, *quant-ph/9910033*.
- [131] G. T. Herman, G. Rozenberg, *Developmental Systems and Languages*, North-Holland, Amsterdam, 1975.
- [132] T. J. Herzog, P. G. Kwiat, H. Weinfurter, A. Zeilinger, Complementarity and the quantum eraser, *Physical Review Letters*, 75, 17 (1995), 3034–3037.
- [133] A. Heyting, ed., *Constructivity in Mathematics*, North-Holland, Amsterdam, 1959.
- [134] J. G. Hey, ed., *Feynman and Computation. Exploring the Limits of Computers*, Perseus Books, Reading, Massachusetts, 1999.
- [135] M. Hirvensalo, Copying quantum computer makes NP-complete problems tractable, in *Proc. of MCU Conference*, Metz, 1998 (M. Margenstern, ed), also as a TUCS Report No 161 (1998).

- [136] M. Hirvensalo, An introduction to quantum computing, *Bulletin of the EATCS*, 66 (1998), 100–121.
- [137] J. Hoffmeyer, Semiosis and living membranes, *1º Seminario Avançado de Comunicação e Semiotica: Biosemiotica e Semiotica Cognitiva*, São Paulo, Brasil, 1998, 9–23.
- [138] M. Hogarth, Predicting the future in relativistic spacetimes, *Studies in History and Philosophy of Science. Studies in History and Philosophy of Modern Physics* 24, 5 (1993), 721–739.
- [139] T. Hogg, Highly structured searches with quantum computers, *Physical Review Letters*, 80 (1998) 2473–2476.
- [140] J. E. Hopcroft, J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, Mass., 1979.
- [141] R. J. Hughes, Cryptography, quantum computation and trapped ions, *Technical Report LA-UR-97-4986*, Los Alamos National Laboratory, 1997.
- [142] L. Hunter, Molecular biology for computer scientists, in L. Hunter, ed., *Artificial Intelligence and Molecular Biology*, AAAI Press/MIT Press, Menlo Park, Calif., 1993, 1–46.
- [143] J. M. Jauch, *Foundations of Quantum Mechanics*, Addison-Wesley, Reading, MA., 1968.
- [144] J. Jarett, On the physical significance of the locality condition in Bell argument, *Noûs*, 18 (1984), 569–589.
- [145] E. Jurvanen, M. Lipponen, Distinguishability, simulation and universality of Moore tree automata, *Fundamenta Informaticae*, 34 (1999), 1–13.
- [146] Gudrun Kalmbach, *Orthomodular Lattices*, Academic Press, New York, 1983.
- [147] L. Kari, DNA computing: tomorrow's reality, *Bulletin of the EATCS*, 59 (June 1996), 256–266.
- [148] L. Kari, G. Gloor, S. Yu, Using DNA to solve the bounded Post correspondence problem, *Proc. of Second Intern. Colloq. Universal Machines and Computations*, Metz, 1998, vol. I, 51–65.
- [149] L. Kari, Gh. Păun, G. Rozenberg, A. Salomaa, S. Yu, DNA computing, sticker systems, and universality, *Acta Informatica*, 35, 5 (1998), 401–420.

- [150] L. Kari, Gh. Păun, G. Thierrin, S. Yu, At the crossroads of DNA computing and formal languages: Characterizing RE using insertion-deletion systems, in [300], 318–333.
- [151] L. Kari, H. Rubin, D. H. Wood, eds., *Preliminary Proceedings of the Fourth International Meeting on DNA Based Computing*, Univ. of Pennsylvania, Philadelphia, June 1998.
- [152] S. Kochen, E. P. Specker, The problem of hidden variables in quantum mechanics, *Journal of Mathematics and Mechanics*, 17(1) (1967), 59–87. Reprinted in [276], 235–263.
- [153] L. Kalmar, An argument against the plausibility of Church’s thesis, in [133], 72–80.
- [154] G. Kreisel, A notion of mechanistic theory, *Synthese*, 29 (1974), 11–16.
- [155] S. N. Krishna, R. Rama, A variant of P systems with active membranes. Solving NP-complete problems, *Romanian J. of Information Science and Technology*, 2, 4 (1999).
- [156] S. N. Krishna, R. Rama, Computing with P systems, submitted, 2000.
- [157] L. Kronsjö, *Algorithms: Their Complexity and Efficiency*, 2nd ed., Wiley, New York, 1987.
- [158] R. Landauer, Irreversibility and heat generation in the computing process, *IBM J. Res. Develop.*, 5 (1961), 183–191.
- [159] R. Landauer, Computation, measurement, communication and energy dissipation, in S. Haykin, ed., *Selected Topics in Signal Processing*, Prentice Hall, Englewood Cliffs, New Jersey, 1989, 18.
- [160] R. Landauer, Information is physical, *Physics Today*, 44 (1991), 23–29.
- [161] R. Landauer, Information is inevitably physical, in [134], 76–92.
- [162] R. Landauer, Information is physical, but slippery, in [39], 59–62.
- [163] H. S. Leff, A. F. Rex, *Maxwell’s Demon: Entropy, Information, Computing*, Princeton University Press, Princeton, 1990.
- [164] A. Lenstra, H. Lenstra, eds, *The Development of the Number Field Sieve*, Springer-Verlag, New York, 1993.
- [165] L. F. Landweber, R. J. Lipton, DNA to DNA: A potential “killer app”? , in [300], 59–68.
- [166] E. Laun, K. J. Reddy, Wet splicing systems, in [300], 115–126.

- [167] W.-H. Li, D. Graur, *Fundamentals of Molecular Evolution*, Sinauer Ass., Sunderland, Mass., 1991.
- [168] R. J. Lipton, Using DNA to solve NP-complete problems. *Science*, 268 (April 1995), 542–545.
- [169] R. J. Lipton, Speeding up computations via molecular biology, in [170], 67–74.
- [170] R. J. Lipton, E. B. Baum, eds., *DNA Based Computers*, Proc. of a DIMACS Workshop, Princeton, 1995, Amer. Math. Soc., 1996.
- [171] S. Lloyd, Almost any quantum gate is universal. *Phys. Rev. Letters*, 75 (1995), 346–349.
- [172] S. Lloyd, Quantum-mechanical Maxwell’s demon, *Phys. Rev.*, A56 (1997), 3374–3382.
- [173] H.-K. Lo, H. F. Chau, Is quantum bit commitment really possible? *Phys. Rev. Lett.*, 78 (1997), 3410–3413.
- [174] H.-K. Lo, T. Spiller, S. Popescu, eds., *Introduction to Quantum Computation and Information*, World Scientific, Singapore, 1999.
- [175] G. W. Mackey, Quantum mechanics and Hilbert space, *Amer. Math. Monthly, Supplement*, 64 (1957), 45–57.
- [176] C. Macchiavello, G. M. Palma, A. Zeilinger, eds., *Quantum Computation and Quantum Information Theory, Collected Papers and Notes*, World Scientific, Singapore, 1999.
- [177] O. Q. Malhas, Quantum logic and the classical propositional calculus, *Journal of Symbolic Logic*, 52(3) (1987), 834–841.
- [178] O. Q. Malhas, Quantum theory as a theory in a classical propositional calculus, *International Journal of Theoretical Physics*, 31(9) (1992), 1699–1714.
- [179] M. Măriță, Membrane Computing in Prolog, in C. S. Calude, M. J. Dinneen, G. Păun, eds., Pre-Proceedings of the Workshop on Multiset Processing, *CDMTCS Research Report* 140, 2000, (www.cs.auckland.ac.nz/CDMTCS).
- [180] V. Manca, String rewriting and metabolism: A logical perspective, in [217], 36–60.
- [181] V. Manca, C. Martin-Vide, Gh. Păun, New computing paradigms suggested by DNA computing: Computing by carving, in [151], 41–56.

- [182] V. Manca, C. Martin-Vide, Gh. Păun, Iterated GSM mappings: A collapsing hierarchy, in J. Karhumaki, H. Maurer, Gh. Păun, G. Rozenberg, eds., *Jewels are Forever*, Springer-Verlag, Berlin, 1999, 182–193.
- [183] M. Margenstern, Y. Rogozhin, An universal time-varying distributed H system of degree 2, in L. Kari, H. Rubin, D. H. Wood, eds., *Proc. of the 4th Intern. Meeting on DNA Based Computers*, Pennsylvania Univ., June 1998, and *BioSystems*, 52, 1–3 (1999).
- [184] C. Martin-Vide, Gh. Păun, Cooperating distributed splicing systems, *J. Automata, Languages, Combinatorics*, 4, 1 (1999), 3–16.
- [185] J. C. Maxwell, *Theory of Heat*, Longman's, Green, & Co., London, 1985, 328–329.
- [186] D. Mayers, Unconditionally secure quantum bit commitment is impossible, *Phys. Rev. Lett.*, 78 (1997), 3414–3417.
- [187] N. D. Mermin, Bringing home the atomic world: Quantum mysteries for anybody, *American Journal of Physics*, 49 (1981), 940–943.
- [188] N. D. Mermin, Quantum mysteries revisited, *American Journal of Physics*, 58 (1990), 731–734.
- [189] N. D. Mermin, Hidden variables and the two theorems of John Bell, *Reviews of Modern Physics*, 65 (1993), 803–815.
- [190] A. Messiah, *Quantum Mechanics*, Volume 1, North-Holland, Amsterdam, 1961.
- [191] T. Mihara, S. S. Sung, A quantum polynomial time algorithm in worst case for Simon's problem (extended abstract), *Proc. 9th Annual Symposium on Algorithms and Computation*, Taejon, 1998, 229–236.
- [192] G. Milburn, Quantum optical Fredkin gate, *Physical Review*, 62 (1989), 2124–2127.
- [193] G. Milburn, *The Feynmann Processor. An Introduction to Quantum Computation*, Allen & Unwin, St. Leonards, 1998.
- [194] M. Minsky, *Computation. Finite and Infinite Machines*, Prentice Hall, Englewood Cliffs, N.J., 1967.
- [195] G. Mitchison, R. Josza, Counterfactual computation, *quant-ph/9907007*.
- [196] E. F. Moore, Gedanken-experiments on sequential machines, in C. E. Shannon and J. McCarthy, eds., *Automata Studies*, Princeton University Press, Princeton, 1956, 128–153.

- [197] P. J. Nahin, *Time Machines*, Springer-Verlag, New York, 1999.
- [198] P. Odifreddi, *Classical Recursion Theory*, North-Holland, Amsterdam, New York, Vol. 1, 1989, Vol. 2, 1999.
- [199] P. Odifreddi, Indiscrete applications of discrete mathematics, in D. S. Bridges, C. S. Calude, J. Gibbons, S. Reeves, I. Witten, eds., *Combinatorics, Complexity, Logic*, Springer-Verlag, Singapore, 1996, 52–65.
- [200] M. Ogihara, A. Ray, DNA-based parallel computation by “counting”, in [300], 265–274.
- [201] M. Ogihara, A. Ray, The minimum DNA computation model and its computational power, in [47], 309–322.
- [202] B. Ömer, *A Procedural Formalism for Quantum Computing*, Masters Thesis, Department of Theoretical Physics, Technical University of Vienna, 1998, (<http://tph.tuwien.ac.at/~oemer>).
- [203] Q. Ouyang, P. D. Kaplan, S. Liu, A. Libchaber, DNA solution of the maximal clique problem, *Science*, 278 (1997), 446–449.
- [204] A. Păun, Extended H systems with permitting contexts of small radius, *Fundamenta Informaticae*, 31, 2 (1997), 185–193.
- [205] A. Păun, On time-varying H systems, *Bulletin of the EATCS*, 67 (1999), 157–164.
- [206] A. Păun, On the diameter of extended H systems, in J. Dassow, D. Wotschke, eds., *Workshop on Descriptive Complexity of Automata, Grammars and Related Structures*, Magdeburg, 1999, 165–174.
- [207] Gh. Păun, On the iteration of gsm mappings, *Revue Roum. Math. Pures Appl.*, 23, 4 (1978), 921–937.
- [208] Gh. Păun, On the power of the splicing operation, *Intern. J. Computer Math.*, 59 (1995), 27–35.
- [209] Gh. Păun, Regular extended H systems are computationally universal, *J. Automata, Languages, Combinatorics*, 1, 1 (1996), 27–36.
- [210] Gh. Păun, On the power of splicing grammar systems, *Ann. Univ. Buc., Matem.-Inform. Series*, 45, 1 (1996), 93–106.
- [211] Gh. Păun, DNA computing; Distributed splicing systems, in J. Mycielski, G. Rozenberg, A. Salomaa, eds., *Structures in Logic and Computer Science. A Selection of Essays in Honor of A. Ehrenfeucht, Lect. Notes in Computer Sci.* 1261, Springer-Verlag, Berlin, 1997, 351–370.

- [212] Gh. Păun, *Marcus Contextual Grammars*, Kluwer Academic Publ., Boston, 1997.
- [213] Gh. Păun, Two-level distributed H systems, in S. Bozapalidis, ed., *Proc. of the Third Conf. on Developments in Language Theory*, Thessaloniki, 1997, Aristotle Univ. of Thessaloniki, 1997, 309–327.
- [214] Gh. Păun, (DNA) Computing by carving, *Research Report CTS-97-17*, Center for Theoretical Study of the Czech Academy of Sciences, Prague, 1997, and (in a revised form) *Soft Computing*, 3, 1 (1999), 30–36.
- [215] Gh. Păun, Distributed architectures in DNA computing based on splicing: Limiting the size of components, in [47], 323–335.
- [216] Gh. Păun, DNA computing based on splicing: universality results, *Proc. of Second Intern. Colloq. Universal Machines and Computations*, Metz, 1998, Vol. I, 67–91.
- [217] Gh. Păun, ed., *Computing with Bio-Molecules. Theory and Experiments*, Springer-Verlag, Singapore, 1998.
- [218] Gh. Păun, Computing with membranes, *J. Computer System Sciences*, 61 (2000), in press, and *TUCS Research Report* No. 208, November 1998, (<http://www.tucs.fi>).
- [219] Gh. Păun, Computing with membranes: An introduction, *Bulletin of the EATCS*, 68 (1999), 139–152.
- [220] Gh. Păun, Computing with membranes – A variant: P systems with polarized membranes, *Intern. J. of Foundations of Computer Science*, 11, 1 (2000), 167–182, and *CDMTCS Research Report* No. 098, 1999, (www.cs.auckland.ac.nz/CDMTCS).
- [221] Gh. Păun, P systems with active membranes: Attacking NP complete problems, *J. Automata, Languages, Combinatorics*, 5 (2000), and *CDMTCS Research Report* No. 102, 1999, (www.cs.auckland.ac.nz/CDMTCS).
- [222] Gh. Păun, G. Rozenberg, Sticker systems, *Theoretical Computer Sci.*, 204 (1998), 183–203.
- [223] Gh. Păun, G. Rozenberg, A. Salomaa, Computing by splicing, *Theoretical Computer Sci.*, 168, 2 (1996), 321–336.
- [224] Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms*. Springer-Verlag, Berlin, 1998.
- [225] Gh. Păun, G. Rozenberg, A. Salomaa, Complementarity versus universality: Keynotes on DNA computing, *Complexity*, 4, 1 (1998), 14–19.

- [226] Gh. Păun, G. Rozenberg, A. Salomaa, Membrane computing with external output, *Fundamenta Informaticae*, 41, 3 (2000), 259–266, and *TUCS Research Report* No. 218, December 1998, (<http://www.tucs.fi>).
- [227] Gh. Păun, Y. Sakakibara, T. Yokomori, P systems on graphs of restricted forms, submitted, 1999.
- [228] Gh. Păun, A. Salomaa, eds., *New Trends in Formal Languages. Control, Cooperation, Combinatorics, Lect. Notes in Computer Sci.*, 1218, Springer-Verlag, Berlin, 1997.
- [229] Gh. Păun, L. Sântean, Parallel communicating grammar systems: the regular case, *Ann. Univ. Buc., Series Matem.-Inform.*, 38 (1989), 55–63.
- [230] Gh. Păun, G. Thierrin, Multiset processing by means of systems of finite state transducers, in O. Boldt, H. Jürgensen, L. Robbins, eds., *Workshop on Implementing Automata WIA99*, Potsdam, August 1999, XV (1–18), and *CDMTCS Research Report* No. 101, 1999, (www.cs.auckland.ac.nz/CDMTCS).
- [231] Gh. Păun, T. Yokomori, Membrane computing based on splicing, in E. Winfree, D. Gifford, eds., *Fifth Intern. Workshop on DNA Based Computers*, MIT, 1999, 213–227.
- [232] Gh. Păun, T. Yokomori, Simulating H systems by P systems, *Journal of Universal Computer Science*, 6, 1 (2000), 178–193, (www.iicm.edu/jucs).
- [233] Gh. Păun, S. Yu, On synchronization in P systems, *Fundamenta Informaticae*, 38, 4 (1999), 397–410.
- [234] R. Penrose, *Shadows of the Mind*, Oxford University Press, Oxford, 1994.
- [235] A. Peres, Einstein, Gödel, Bohr, *Foundations of Physics*, 15 (1985), 201–205.
- [236] A. Peres, *Quantum Theory: Concepts and Methods*, Kluwer Academic Publishers, Dordrecht, 1993.
- [237] I. Petre, A normal form for P systems, *Bulletin of the EATCS*, 67 (1999), 165–172.
- [238] I. Petre, L. Petre, Mobile ambients and P systems, *Workshop on Formal Languages, FCT'99*, Iași, Romania, 1999.
- [239] N. Pisanti, DNA computing: a survey, *Bulletin of the EATCS*, 64 (February 1998), 188–216.

- [240] D. Pixton, Regular splicing systems, manuscript, 1995.
- [241] D. Pixton, Regularity of splicing languages, *Discrete Appl. Math.*, 69 (1996), 101–124.
- [242] D. Pixton, Splicing in abstract families of languages, *Technical Report of SUNY Univ. at Binghamton, New York*, 1997.
- [243] L. Priese, Y. Rogozhin, M. Margenstern, Finite H systems with 3 tubes are not predictable, in [7], 547–558.
- [244] I. Prigogine, *From Being to Becoming*, W. H. Freeman, San Francisco, 1980.
- [245] P. Pták, S. Pulmannová, *Orthomodular Structures as Quantum Logics*, Kluwer Academic Publishers, Dordrecht, 1991.
- [246] M. O. Rabin, Probabilistic algorithms, in [288], 21–39.
- [247] T. Rado, On non-computable functions, *Bell System Technical Journal* 41 (1962), 877–884.
- [248] H. Reichenbach, *The Direction of Time*, University of California Press, LA, 1956.
- [249] J. H. Reif, Paradigms for biomolecular computation, in [47], 72–93.
- [250] E. Rieffel, W. Polak, An introduction to quantum computing for non-physicists, *quant-ph/9809016*.
- [251] R. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public key cryptosystems, *Communications of ACM*, 21 (1978), 120–126.
- [252] Y. Rogozhin, Small universal Turing machines, *Theoretical Computer Sci.*, 168 (1996), 215–240.
- [253] Y. Rogozhin, A universal Turing machine with 22 states and 2 symbols, *Romanian J. of Information Science and Technology*, 1, 3 (1998), 259–265.
- [254] R. Rosen, Effective processes and natural law, in R. Herken, ed., *The Universal Turing Machine. A Half-Century Survey*, Kammerer & Unverzagt, Hamburg, 1988, p. 523.
- [255] B. Rovan, A framework for studying grammars, *Proc. MFCS 81, Lect. Notes in Computer Sci.* 118, Springer-Verlag, Berlin, 1981, 473–482.
- [256] G. Rozenberg, A. Salomaa, *The Mathematical Theory of L Systems*, Academic Press, New York, 1980.

- [257] G. Rozenberg, A. Salomaa, Watson-Crick complementarity, universal computations and genetic engineering, *Techn. Report 96-28*, Dept. of Computer Science, Leiden Univ., Oct. 1996.
- [258] G. Rozenberg, A. Salomaa, eds., *Handbook of Formal Languages*, 3 volumes, Springer-Verlag, Berlin, 1997.
- [259] A. Salomaa, *Formal Languages*, Academic Press, New York, 1973.
- [260] A. Salomaa, *Computation and Automata*, Cambridge Univ. Press, Cambridge, 1985.
- [261] A. Salomaa, Equality sets for homomorphisms of free monoids, *Acta Cybernetica*, 4 (1978), 127–139.
- [262] A. Salomaa, *Jewels of Formal Language Theory*, Computer Science Press, Rockville, Md., 1981.
- [263] A. Salomaa, *Public-Key Cryptography*, Springer-Verlag, Berlin, 1996.
- [264] A. Salomaa, Turing, Watson–Crick, and Lindenmayer. Aspects of DNA complementarity, in [47], 94–107.
- [265] B. Schumaker, Quantum coding, *Physical Review A*, 51, 4 (1995), 2738–2747.
- [266] D. B. Searls, The linguistics of DNA, *American Scientist*, 80 (1992), 579–591.
- [267] C. E. Shannon, A universal Turing machine with two internal states, *Automata Studies, Annals of Mathematical Studies*, 34, Princeton Univ. Press, 1956, 157–165.
- [268] C. E. Shannon, Computers and automata, *Proceedings of the I. R. E.* 41 (1953), 1235–1241.
- [269] P. W. Shor, Algorithms for quantum computation: discrete log and factoring, *Proceedings of the 35th IEEE Annual Symposium on Foundations of Computer Science*, 1994, 124–134.
- [270] D. R. Simon, On the power of quantum computation, *SIAM J. Comput.* 26, 5 (1997), 1474–1483; first published in *Proc. of the 35th IEEE Symposium on Foundations of Computer Science (FOCS)*, Santa Fe, New Mexico, IEEE Computer Society Press, Los Alamitos, CA, 1994, 116–123.
- [271] H. T. Siegelmann, Computation beyond the Turing limit, *Science*, 268 (April 1995), 545–548.

- [272] H. T. Siegelmann, *Neural Networks and Analog Computation: Beyond the Turing Limit*, Birkhauser, Boston, MA, 1999.
- [273] R. I. Soare, *Recursively Enumerable Sets and Degrees*, Springer-Verlag, Berlin, 1987.
- [274] W. M. Sofer, *Introduction to Genetic Engineering*, Butterworth-Heinemann, Boston, 1991.
- [275] E. Specker, Die Logik nicht gleichzeitig entscheidbarer Aussagen, *Dialectica*, 14 (1960), 175–182. Reprinted in [276], 175–182.
- [276] E. Specker, *Selecta*, Birkhäuser Verlag, Basel, 1990.
- [277] A. Steane, Quantum computing, *quant-ph/9708022*.
- [278] Y. Suzuki, H. Tanaka, On a LISP implementation of a class of P systems, *Romanian J. of Information Science and Technology*, 3, 2 (2000).
- [279] Y. Suzuki, H. Tanaka, Order parameter for a symbolic chemical system, *Proc. of Artificial Life Conf. VI*, MIT Press, 1998, 130–139.
- [280] Y. Suzuki, H. Tanaka, Symbolic chemical systems based on an abstract rewriting system and its behavior pattern, *Journal of Artificial Life and Robotics*, 1 (1997), 211–219.
- [281] Y. Suzuki, S. Tsumoto, H. Tanaka, Analysis of cycles in symbolic chemical systems based on abstract rewriting systems on multisets, *Prof. of Artificial Life Conf. V*, MIT Press, 1996, 522–528.
- [282] K. Svozil, *Randomness & Undecidability in Physics*, World Scientific, Singapore, 1993.
- [283] K. Svozil, *Quantum Logic*, Springer-Verlag, Singapore, 1998.
- [284] K. Svozil, The Church-Turing Thesis as a guiding principle for physics, in [47], 371–385.
- [285] K. Svozil, One-to-one, *Complexity*, 4 (1) (1998), 25–29.
- [286] L. Szilard, On the decrease of entropy in a thermodynamic system by the intervention of intelligent beings, *Zeitschrift für Physics*, 53 (1929), 840–856.
- [287] T. Toffoli, N. Margolus, *Cellular Automata Machines*, MIT Press, Cambridge, MA, 1987.
- [288] J. Traub, *Algorithms and Complexity: New Directions and Results*, Academic Press, London, 19976.

- [289] A. M. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Math. Soc.*, Ser. 2, 42 (1936), 230–265; a correction, 43 (1936), 544–546.
- [290] H. Weyl, *Philosophy of Mathematics and Natural Science*, Princeton University Press, Princeton, 1949.
- [291] J. A. Wheeler, W. H. Zurek, *Quantum Theory and Measurement*, Princeton University Press, Princeton, 1983.
- [292] U. Vazirani, Quantum computing, 1997, (<http://www.cs.berkeley.edu/~vazirani.>)
- [293] E. Winfree, Complexity of restricted and unrestricted models of molecular computability, in [170], 187–198.
- [294] E. Winfree, On the computational power of DNA annealing and ligation, in [170], 199–210.
- [295] E. Winfree, *Algorithmic Self-Assembly of DNA*, PhD Thesis, California Institute of Technology, Pasadena, CA, 1998.
- [296] C. P. Williams, S. H. Clearwater, *Explorations in Quantum Computing*, Springer-Verlag, New York, 1997.
- [297] C. P. Williams, S. H. Clearwater, *Ultimate Zero and One: Computing at the Quantum Frontier*, Springer-Verlag, Heidelberg, 2000.
- [298] E. Winfree, X. Yang, N. Seeman, Universal computation via self-assembly of DNA; some theory and experiments, in [18], 191–213.
- [299] D. Wood, Iterated NGSM maps and Γ -systems, *Inform. Control*, 32 (1976), 1–26.
- [300] D. Wood, L. Kari, R. Lipton, J. Reif, N. Seeman, E. Winfree, eds., *Preliminary Proceedings of the 3rd DIMACS Workshop on DNA Based Computers*, Pennsylvania Univ., Philadelphia, June 1997.
- [301] W. K. Wootters, W. H. Zurek, A single quantum cannot be cloned, *Nature*, 299 (1992), 802–803.
- [302] A. Yao, Quantum circuit complexity, *Proceedings of the 34th IEEE Symposium on Foundations of Computer Science (FOCS)*, IEEE Computer Science Press, Los Alamitos, 1993, 352–260.
- [303] T. Yokomori, YAC: Yet another computation model of self-assembly, in E. Winfree, D. Gifford, eds., *Preliminary Proc. 5th Intern. Workshop on DNA Based Computers*, MIT, 1999, 153–168.

- [304] T. Yokomori, Computation = self-assembly + conformational change: Toward new computing paradigm, *Preproceedings of DLT 99* (Invited Lecture), Aachen, 1999, 21–30.
- [305] C. Zalka, Grover’s quantum searching algorithm is optimal, *quant-ph/9711070*.
- [306] C. Zandron, C. Ferretti, G. Mauri, A reduced distributed splicing system for RE languages, in [228], 346–366.
- [307] N. Zierler, M. Schlessinger, Boolean embeddings of orthomodular sets and quantum logic, *Duke Mathematical Journal*, 32 (1965), 251–262.
- [308] W. H. Zurek, Algorithmic randomness, physical entropy, measurements, and the Demon of choice, in [134], 393–410.

Index

- adenine (A), 24
- alphabet, 1
- amplification, 34, 90
- amplify, 40
- annealing, 30, 65
- asymmetric graph, 154

- Basic Universality Lemma, 83
- Bell's Theorem, 216
- bounded language, 127
 - strictly, 127
- BPP, 265
- BQP, 272
- Bridge Theorem, 122

- catalyst, 112
- cellular automata, 51
- chemical abstract machine, 110
- Chomsky grammar, 8
 - context-free, 9
 - context-sensitive, 9
 - linear, 9
 - monotonous, 9
 - regular, 9
- Chomsky hierarchy, 9
- Church-Turing Thesis, 15, 57
- coding, 2
- complementation, 56
- complete data pool, 50
- computably enumerable language, 9, 56, 72, 83, 123
- computably enumerable languages, 146
- computational completeness, 16
- computing by carving, 50, 55
- copy, 47

- concatenation, 2
- controlled- U gate, 238
- controlled-controlled-NOT, 202
- controlled-NOT gate, 201
- cytosine (C), 24

- DAE construct, 51
- deletion, 34
- denaturation, 30
- deoxyribonucleic acid, 23
- detect, 40
- Discrete Fourier Transform, 250
- DNA, see deoxyribonucleic acid, 23
- double helix, 24

- endonuclease, 31, 32
- equality set, 15
- equivalent grammars, 8
- error probability, 264
- Euclid's algorithm, 254
- evolutionary computing, 41
- exonuclease, 31

- Fast Fourier Transform, 250
- filtering, 35
- finite automaton, 11, 81
 - deterministic, 11
 - Watson-Crick, 75
 - reverse, 76

- gel electrophoresis, 36
- gel electrophoresis, 39
- generalized sequential machine (gsm), 12, 58
- grammar scheme, 18
- grammar system, 9, 95

- cooperating distributed, 9
- parallel communicating, 10, 95
- splicing, 95
- Grover's search algorithm, 261
- gsm
 - deterministic, 12
- guanine (G), 24
- H scheme, 80
- H system, 77
 - communicating distributed, 97
 - distributed, 95
 - extended, 85
 - ordered, 89
 - sequential distributed, 102
 - time-varying, 98
 - with forbidding contexts, 88
 - with global target, 89
 - with local targets, 89
 - with multisets, 91
 - with permitting contexts, 87
- halting problem, 206
- Hamiltonian path problem, 37, 47
- insertion, 34
- Kleene closure, 2
- Landauer's principle, 182
- language, 1
- length-separate**, 41
- ligase, 34, 77
- ligation, 34
- Lindenmayer system, 10
 - deterministic, 10
 - extended, 11
 - propagating, 10
 - tabled, 11
- matrix grammar, 122
 - in binary normal form, 123
 - with appearance checking, 122
- matrix languages, 146
- maximal ideal, 229
- maximum clique problem, 47, 49
- Maxwell demon, 187
- membrane, 111
 - elementary, 111
- membrane structure, 110
 - index of, 111
- merge**, 40
- menmology, 43
- metabolic system, 110
- mirror image, 2
- morphism, 2
 - λ -free, 2
- multiset, 2, 40, 90, 112
- M*₀₂, 226
- normal form, 10
 - Geffert, 10
 - Kuroda, 10
- nuclease, 31
- nucleotide, 23
- oligonucleotide (oligo), 31
- orthocomplemented lattice, 225
- orthoposet, 228
- P system, 112
 - Computational Completeness Lemma, 123, 133, 139, 165
 - configuration of, 114
 - rewriting, 132
 - splicing, 168
 - with polarized membranes, 136
- P' system, 154
- Parikh set, 1
- Parikh vector, 1
- Peres, 213, 227, 234, 235
- polymerase (DNA), 31
- Polymerase Chain Reaction (PCR), 34
- polynomial time bounded probabilistic machine, 265
- position-separate**, 41

- PP, 265
primer, 31
Principle of Superposition, 192
probabilistic Turing machine, 264
program-size complexity, 214
projection, 2
protein channel, 157
pumping lemma, 63
pure grammar, 8
purine, 24
pyrimidine, 24
- quanta, 189
Quantum Fourier Transform, 250
quantum parallelism, 244
quantum Turing machine, 271
- radius, 80, 113
region, 111
regular approximation, 62
regular language, 69
regular sequence of languages, 58
Regularity Preserving Lemma, 81
regulated rewriting, 87
Reichenbach, 222
remove, 47
restriction enzyme, 32, 77
rewriting system, 8
- SAT problem, 44, 157, 160
satisfiability problem, 20
select, 47
sentential form, 8
separate, 40
sequential transducer, 12
 Watson–Crick, 77
Shor’s algorithm, 252
shuffle, 2
Simon’s problem, 266
space complexity, 19
splicing, 77, 79
 iterated, 81
 rule, 80
splicing rule, 80
sticker operation, 67
- sticker system, 68
 one-sided, 69
 regular, 69
 simple, 69
sticky end, 25
subgraph isomorphism problem, 47
super-cell, 112
- template, 31
test tube programming language, 40
- three-vertex-colourability, 47
thymine (T), 24
time complexity, 19
Turing machine, 13, 15
 deterministic, 13
 universal, 16
twin-shuffle language, 23, 26
 reverse, 29
- type-0 grammar, 9
 universal, 18, 87
- uncertainty principle, 189
union, 47
universal type-0 grammar, 18
universality, 16
 U_f , 241
- Walsh–Hadamard transformation, 198
Watson–Crick complementarity, 24, 39, 65, 78
Watson–Crick domain, 65

'Goes right to the frontier - and across it - in giving an accessible account of today's fast-developing ways of harnessing the processes of nature for computation. If you ever wondered how computer scientists and engineers are using cellular processes and the strange nature of the quantum realm to develop tomorrow's computers, this is the place to find out. A first-rate treatment of what is sure to become the computing technology of the 21st century.'

John L. Casti, Santa Fe Institute, Santa Fe, NM, USA and the Technical University of Vienna, Vienna, Austria.

'Natural computing, i.e., computing inspired by or gleaned from nature, is an exciting and fast growing research area at the crossroads of computer science, biology, biochemistry and physics. This book provides a good insight into some of the basic models of natural computing. The reader of this well written and engaging book will develop a vision of what computing is going to be like in the not too distant future.'

G. Rozenberg, Director of the Leiden Centre for Natural Computing, Leiden University, The Netherlands and the Dept. of Computer Science, University of Colorado at Boulder, CO, USA.

The theory and technology of computation has rested for more than 50 years on the Turing-machine model, which leads to many intractable problems. Are there any alternatives? Two main directions of research, both based on quite unconventional ideas, are most promising: quantum computing and molecular computing, especially using DNA.

The book presents the main practical results reported so far and the main theoretical developments. In the DNA computing coverage, the authors discuss Adleman's famous experiment, with subsequent variants as well as many theoretical models: sticker systems and Watson-Crick automata, insertion-deletion systems, splicing systems and the idea of "computing by carving". A special feature is the chapter about P systems, computing models based on membrane structures whose theory has only recently emerged. In the quantum realm, the authors present the elementary theory, the logic of quantum computation as well as some important applications to cryptography, teleportation, error correction and randomness.

The book is self-contained, including all the necessary facts from mathematics, computer science, biology and quantum mechanics. It provides a lucid and critical introduction for graduates and advanced undergraduates.

Cristian S. Calude is Professor and Director of the Centre for Discrete Mathematics and Theoretical Computer Science, Auckland, New Zealand. **Gheorghe Paun** is Senior Researcher (Mathematics) and Corresponding Member of the Romanian Academy, Bucharest, Romania.

11 New Fetter Lane, London EC4P 4EE
29 West 35th Street, New York NY 10001
Printed in Great Britain www.tandf.co.uk

ISBN 0-7484-0899-1



9 780748 408993

