Studies in Computational Intelligence 113

Gemma Bel-Enguix
M. Dolores Jiménez-López
Carlos Martín-Vide (Eds.)

New Developments in Formal Languages and Applications

Springer

Gemma Bel-Enguix, M. Dolores Jiménez-López and Carlos Martín-Vide (Eds.)

New Developments in Formal Languages and Applications

Gemma Bel-Enguix
M. Dolores Jiménez-López
Carlos Martín-Vide
(Eds.)

New Developments in Formal Languages and Applications

Springer

Gemma Bel-Enguix
M. Dolores Jiménez-López
Carlos Martín-Vide

Rovira i Virgili University
Research Group on Mathematical Linguistics
Plaza Imperial Tàrraco, 1
43005 Tarragona
Spain

ISBN 978-3-540-78290-2 e-ISBN 978-3-540-78291-9

Studies in Computational Intelligence ISSN 1860-949X

Library of Congress Control Number: 2008921711

Cover design: Deblik, Berlin, Germany

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

Preface

The theory of formal languages is widely accepted as the backbone of theoretical computer science. It mainly originated from mathematics (combinatorics, algebra, mathematical logic) and generative linguistics. Later, new specializations emerged from areas of either computer science (concurrent and distributed systems, computer graphics, artificial life), biology (plant development, molecular genetics), linguistics (parsing, text searching), or mathematics (cryptography). All human problem solving capabilities can be considered, in a certain sense, as a manipulation of symbols and structures composed by symbols, which is actually the stem of formal language theory. Language – in its two basic forms, natural and artificial – is a particular case of a symbol system.

This wide range of motivations and inspirations explains the diverse applicability of formal language theory Ũ and all these together explain the very large number of monographs and collective volumes dealing with formal language theory.

In 2004 Springer-Verlag published the volume Formal Languages and Applications, edited by C. Martín-Vide, V. Mitrana and G. Păun in the series Studies in Fuzziness and Soft Computing 148, which was aimed at serving as an overall course-aid and self-study material especially for PhD students in formal language theory and applications. Actually, the volume emerged in such a context: it contains the core information from many of the lectures delivered to the students of the International PhD School in Formal Languages and Applications organized since 2002 by the Research Group on Mathematical Linguistics from Rovira i Virgili University, Tarragona, Spain.

During the editing process of the aforementioned volume, two situations appeared:

Some important aspects, mostly extensions and applications of classical formal language theory to different scientific areas, could not be covered, by different reasons. New courses were promoted in the next editions of the PhD School mentioned above.

To intend to fill up this gap, the volume Recent Advances in Formal Languages and Applications, edited by Z. Ésik, C. Martín-Vide and V. Mitrana, was published in 2006 by Springer-Verlag in the series Studies in Computational Intelligence 25.

The present volume is a continuation of this comprehensive publication effort. We believe that, besides accomplishing its main goal of complementing the previous volumes in representing a gate to formal language theory and its applications, it will be also useful as a general source of information in computation theory, both at the undergraduate and research level.

For the sake of uniformity, the introductory chapter of the first volume that presents the mathematical prerequisites as well as most common concepts and notations used throughout all chapters appears in the present volume as well. However, it may happen that terms other than those in the introductory chapter have different meanings in different chapters or different terms have the same meaning. In each chapter, the subject is treated relatively independent of the other chapters, even if several chapters are related. This way, the reader gets in touch with diverse points of view on an aspect common to two or more chapters. We are convinced of the usefulness of such an opportunity to a young researcher.

Acknowledgements

Our deep gratitude is due to all the contributors, for their professional and friendly cooperation, as well as to Springer-Verlag, for the efficient and pleasant collaboration.

Tarragona,
October 2007

Gemma Bel-Enguix
M. Dolores Jiménez-López
Carlos Martín-Vide

Contents

1

Basic Notation and Terminology

This chapter presents the basic mathematical and formal language theory notations and terminology used throughout the book.

1.1 General Mathematical Notations

The notations are those provided by standard Latex and customary in mathematics.

Set theory: $\in$ denotes the membership (of an element to a set), $\subseteq$ denotes the inclusion (not necessarily proper) and $\subset$ denotes the strict inclusion; the union, intersection, and difference of two sets are denoted by $\cup, \cap, -$, respectively. (We do not use $\backslash$ for the difference, because $\backslash$ denotes the left quotient of languages.) The empty set is denoted by $\emptyset$, the power set of a set X is denoted by 2^X, while the cardinality of a set X is denoted by $\mathrm{card}(X)$. A singleton set is often identified with its single element, and hence we also write a for $\{a\}$. Two sets X and Y are said to be *incomparable* if both $X - Y$ and $Y - X$ are non-empty.

Sets of numbers: the set of natural numbers (zero included) is denoted by $\mathbf{N}$, while the sets of integer, rational, and real numbers are denoted by $\mathbf{Z}$, $\mathbf{Q}$, $\mathbf{R}$, respectively. The subsets of these sets consisting of strictly positive numbers are denoted by $\mathbf{N}_+, \mathbf{Z}_+, \mathbf{Q}_+, \mathbf{R}_+$, respectively.

1.2 Basic String Notions and Notation

An *alphabet* is a finite nonempty set of abstract symbols. For an alphabet V we denote by V^* the set of all *strings* (we also say *words*) of symbols from 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 being a subset of V^+) is said to be λ-*free.*

If $x = x_1x_2$, for some $x_1, x_2 \in V^*$, then x_1 is called a *prefix* of x and x_2 is called a *suffix* of x; if $x = x_1x_2x_3$ for some $x_1, x_2, x_3 \in V^*$, then x_2 is called a *substring* of x. The sets of all prefixes, suffixes, and substrings of a string x are denoted by $\mathrm{Pref}(x)$, $\mathrm{Suf}(x)$, and $\mathrm{Sub}(x)$, respectively. The sets of proper (that is, different from λ and from the string itself) prefixes, suffixes, and subwords of x are denoted by $\mathrm{PPref}(x)$, $\mathrm{PSuf}(x)$, and $\mathrm{PSub}(x)$, respectively.

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

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

For a language $L \subseteq V^*$, the set $length(L) = \{|x| \mid x \in L\}$ is called the *length set* of L.

The set of symbols occurring in a string x is denoted by $alph(x)$. For a language $L \subseteq V^*$, we denote $alph(L) = \bigcup_{x \in L} alph(x)$.

The *Parikh vector* associated with a string $x \in V^*$ with respect to the alphabet $V = \{a_1, \ldots, a_n\}$ is $\Psi_V(x) = (|x|_{a_1}, |x|_{a_2}, \ldots, |x|_{a_n})$ (note that the ordering of the symbols from V is relevant). For $L \subseteq V^*$ we define $\Psi_V(L) = \{\Psi_V(x) \mid x \in L\}$; the mapping $\Psi_V : V^* \longrightarrow \mathbf{N}^n$ is called the *Parikh mapping* associated with V.

A set M of vectors in $\mathbf{N}^n$, for some $n \geq 1$, is said to be *linear* if there are $m \geq 0$ and the vectors $v_i \in \mathbf{N}^n$, $0 \leq i \leq m$, such that

$$M = \{v_0 + \sum_{i=1}^{m} \alpha_i v_i \mid \alpha_1, \ldots, \alpha_m \in \mathbf{N}\}.$$

A finite union of linear sets is said to be *semilinear.*

A language $L \subseteq V^*$ is *semilinear* if $\Psi_V(L)$ is a semilinear set. The family of semilinear languages is denoted by $SLIN$.

1.3 Operations with Strings and Languages

The operations of union, intersection, difference, and complement are defined for languages in the standard set-theoretical way.

The *concatenation* of two languages L_1, L_2 is $L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}$.

We define further:

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

A mapping $s : V \longrightarrow 2^{U^*}$, extended to $s : V^* \longrightarrow 2^{U^*}$ by $s(\lambda) = \{\lambda\}$ and $s(x_1x_2) = s(x_1)s(x_2)$, for $x_1, x_2 \in V^*$, is called a *substitution.* If for all $a \in V$ we have $\lambda \notin s(a)$, then h is a λ-*free* substitution. If $\mathrm{card}(s(a)) = 1$ for all $a \in V$, then s is called a *morphism* (we also say homomorphism).

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 a morphism $h : V^* \longrightarrow U^*$, we define a mapping $h^{-1} : U^* \longrightarrow 2^{V^*}$ (and we call it an *inverse morphism*) by $h^{-1}(w) = \{x \in V^* \mid h(x) = w\}$, $w \in U^*$.

The substitutions (hence also the morphisms and inverse morphisms) are extended to languages in the natural way.

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

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

The *left quotient* of a language $L_1 \subseteq V^*$ with respect to $L_2 \subseteq V^*$ is

$$L_2 \backslash L_1 = \{w \in V^* \mid \text{there is } x \in L_2 \text{ such that } xw \in L_1\}.$$

The *left derivative* of a language $L \subseteq V^*$ with respect to a string $x \in V^*$ is

$$\partial_x^l(L) = \{w \in V^* \mid xw \in L\}.$$

The *right quotient* and the *right derivative* are defined in a symmetric manner:

$$\begin{aligned} &L_1/L_2 = \{w \in V^* \mid \text{there is } x \in L_2 \text{ such that } wx \in L_1\},\\ &\partial_x^r(L) = \{w \in V^* \mid wx \in L\}. \end{aligned}$$

Let $\mathcal{F}$ be a family of languages and $\circ$ be an n-ary operation with languages from $\mathcal{F}$. The family $\mathcal{F}$ is *closed under* $\circ$ if $\circ(L_1, L_2, \ldots, L_n) \in \mathcal{F}$ for any choice of the languages $L_i \in \mathcal{F}$, $1 \leq i \leq n$. The family $\mathcal{F}$ is *closed under substitution with languages from* the family $\mathcal{C}$ if for any language $L \subseteq V^*$, $L \in \mathcal{F}$, and any substitution $s : V^* \longrightarrow 2^{U^*}$ such that $s(a) \in \mathcal{C}$ for all $a \in V$, the language $s(L) = \bigcup_{x \in L} s(x)$ still lies in $\mathcal{F}$. If $\mathcal{C} = \mathcal{F}$, we simply say that $\mathcal{F}$ is closed under substitution.

A family of languages closed under (arbitrary) λ-free morphisms, inverse morphisms and intersection with regular languages is called *(full) trio* - known also as *(cone) faithful cone.* If a (full) trio is further closed under union, then it is called *(full) semi-AFL.* The abbreviation AFL comes from *Abstract Family of Languages.* A (full) semi-AFL closed under concatenation and Kleene (*-) $+$-closure is called a *(full) AFL.*

1.4 Chomsky Grammars

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* (start symbol), 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 $u \to v$. Note that $|u|_N \geq 1$. Sometimes, one uses to denote by V_G the total alphabet of G, that is, $V_G = N \cup T$.

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

$$x \Longrightarrow_G y \text{ iff } x = x_1ux_2, y = x_1vx_2,$$
$$\text{for some } x_1, x_2 \in (N \cup T)^* \text{ and } u \to v \in P.$$

One says that x *directly derives* y (with respect to G). When G is understood we write $\Longrightarrow$ instead of $\Longrightarrow_G$. The reflexive closure of the relation $\Longrightarrow$ is denoted by $\Longrightarrow^+$, and the reflexive and transitive closure by $\Longrightarrow^*$. Each string $w \in (N \cup T)^*$ such that $S \Longrightarrow_G^* w$ is called a *sentential form.*

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

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

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).

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

- *length-increasing* (one also says *monotonous*), if for all $u \to v \in P$ we have $|u| \leq |v|$.
- *context-sensitive*, if each $u \to v \in P$ has $u = u_1Au_2, v = u_1xu_2$, for $u_1, u_2 \in (N \cup T)^*, A \in N$, and $x \in (N \cup T)^+$. (In length-increasing and context-sensitive grammars the production $S \to \lambda$ is allowed, provided that S does not appear in the right-hand members of rules in P.)
- *context-free*, if each production $u \to v \in P$ has $u \in N$.
- *linear*, if each rule $u \to v \in P$ has $u \in N$ and $v \in T^* \cup T^*NT^*$.
- *right-linear*, if each rule $u \to v \in P$ has $u \in N$ and $v \in T^* \cup T^*N$.
- *left-linear*, if each rule $u \to v \in P$ has $u \in N$ and $v \in T^* \cup NT^*$.
- *regular*, if each rule $u \to v \in P$ has $u \in N$ and $v \in T \cup TN \cup \{\lambda\}$.

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

We denote by *RE, LI, CS, CF, LIN, RLIN, LLIN*, and *REG* the families of languages generated by arbitrary, length-increasing, context-sensitive, context-free, linear, right-linear, left-linear, and regular grammars, respectively (RE stands for *recursively enumerable*). By FIN we denote the family of finite languages, and by ARB the family of arbitrary languages.

The following equalities and strict inclusions hold:

$$FIN \subset REG = RLIN = LLIN \subset LIN \subset CF \subset CS = LI \subset RE \subset ARB.$$

We call this the *Chomsky hierarchy.*

1.5 Decision Problems

The goal of this section is to give an informal description of a decision problem and to mention the most common decision problems in formal language theory.

Roughly speaking, a *decision problem* requires an output YES/NO to any of its instances. For example, "Is the natural number n prime?" is a decision problem; further, "Is 3 prime?" is an instance of the problem which is true while "Is 4 prime?" is a false instance of the same problem. A decision problem is *(algorithmically/recursively) decidable* if there exists an algorithm, which for any instance of the problem given as input, outputs YES or NO, provided that the input is true or not, respectively.

The most common decision problems in formal language theory are:

- *Emptiness*: Is a given language empty?
- *Finiteness*: Is a given language a finite set?
- *Membership*: Does $w \in L$ hold for a given word w and a language L?
- *Inclusion*: Does $L_1 \subseteq L_2$ hold for two given languages L_1 and L_2?
- *Equivalence*: Does $L_1 = L_2$ hold for two given languages L_1 and L_2?

Clearly, a decision problem is proved to be decidable if one provides an algorithm as above. Generally, a decision problem is proved to be undecidable by reducing it to a problem known to be undecidable. The following combinatorial problem, known as the *Post Correspondence Problem* (PCP), is undecidable. An instance of the PCP consists of an alphabet V with at least two letters and two lists of words over V

$$u = (u_1, u_2, \ldots, u_n) \quad \text{and} \quad v = (v_1, v_2, \ldots, v_n).$$

The problem asks whether or not a sequence $i_1, i_2, \ldots, i_k$ of positive integers exists, each between 1 and n, such that $u_{i_1} u_{i_2} \ldots u_{i_k} = v_{i_1} v_{i_2} \ldots v_{i_k}$.

We do not give here further elements of formal language theory. They will be elaborated in the subsequent chapters.

For the reader's convenience, we end this section with a list of monographs and collective volumes directly or partially related to formal language theory.

1.6 Books on Formal Language Theory

1. 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.
2. A.V. Aho, J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977.
3. I. Alexander, F.K. Hanna, *Automata Theory: An Engineering Approach*, Crane Russak, 1975.
4. J. Berstel, *Transductions and Context-Free Languages*, Teubner, Stuttgart, 1979.
5. R.V. Book, ed., *Formal Language Theory. Perspectives and Open Problems*, Academic Press, New York, 1980.
6. W. Brauer, *Automatentheorie*, B.G. Teubner, Stuttgart, 1984.
7. C. Choffrut, ed., *Automata Networks, Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1988.
8. D.I.A. Cohen, *Computer Theory*, 2nd edition, John Wiley, 1997.
9. E. Csuhaj-Varjú, J. Dassow, J. Kelemen, Gh. Păun, *Grammar Systems. A Grammatical Approach to Distribution and Cooperation*, Gordon and Breach, London, 1994.
10. J. Dassow, Gh. Păun, *Regulated Rewriting in Formal Language Theory*, Springer-Verlag, Berlin, Heidelberg, 1989.
11. J. Dassow, G. Rozenberg, A. Salomaa, eds., *Developments in Language Theory*, World Scientific, Singapore, 1995.
12. M.D. Davis, E.J. Weyuker, *Computability, Complexity, and Languages*, Academic Press, New York, 1983.
13. P.J. Denning, J.B. Dennis, J.E. Qualitz, *Machines, Languages, and Computation*, Prentice-Hall, Englewood Cliffs, N.J., 1978.
14. D.-Z. Du, K.-I Ko, *Problem Solving in Automata, Languages and Complexity*, John Wiley, 2001.
15. H. Ehrig, G. Engels, H-J. Kreowski, G. Rozenberg, eds., *Handbook of Graph Grammars and Computing by Graph Transformation*, World Scientific, Singapore, 1999.
16. S. Eilenberg, *Automata, Languages, and Machines*, Academic Press, New York, vol. A: 1974, vol. B: 1976.
17. E. Engeler, *Formal Languages*, Markham, Chicago, 1968.
18. Z. Ésik, C. Martín-Vide, V. Mitrana, eds., *Recent Advances in Formal Languages and Applications*, Springer-Verlag, Berlin, 2006.
19. K.S. Fu, *Syntactic Pettern Recognition. Applications*, Springer-Verlag, Heidelberg, 1977.
20. M.R. Garey, D.S. Johnson, *Computers and Intractability. A Guide to the Theory of NP-completeness*, W.H. Freeman, San Francisco, 1979.
21. F. Gécseg, *Products of Automata*, Springer-Verlag, Berlin, 1986.
22. F. Gécseg, I. Peak, *Algebraic Theory of Automata*, Akademiai Kiado, Budapest, 1972.
23. F. Gécseg, M. Steinby, *Tree Automata*, Akademiai Kiado, Budapest, 1984.

24. S. Ginsburg, *The Mathematical Theory of Context-Free Languages*, McGraw-Hill Book Comp., New York, 1966.
25. S. Ginsburg, *Algebraic and Automata-Theoretic Properties of Formal Languages*, North-Holland, Amsterdam, 1975.
26. A. Ginzburg, *Algebraic Theory of Automata*, Academic Press, New York, 1968.
27. M. Gross, A. Lentin, *Notions sur les grammaires formelles*, Gauthier-Villars, Paris, 1967.
28. M. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, Mass., 1978.
29. G.T. Herman, G. Rozenberg, *Developmental Systems and Languages*, North-Holland, Amsterdam, 1975.
30. J.E. Hopcroft, J.D. Ullman, *Formal Languages and Their Relations to Automata*, Addison-Wesley, Reading, Mass., 1969.
31. J.E. Hopcroft, J.D. Ullman, *Introduction to Automata Theory, Languages, and Computing*, Addison-Wesley, Reading, Mass., 1979.
32. J.E. Hopcroft, R. Motwani, J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Boston, 2001.
33. M. Ito, ed., *Words, Languages, and Combinatorics*, World Scientific, Singapore, 1992.
34. M. Ito, Gh. Păun, S. Yu, eds., *Words, Semigroups, and Transductions*, World Scientific, Singapore, 2001.
35. J. Karhumäki, H.A. Maurer, Gh. Păun, G. Rozenberg, eds., *Jewels are Forever*, Springer-Verlag, Berlin, 1999.
36. D. Kelley, *Automata and Formal Languages. An Introduction.* Prentice-Hall, New Jersey, 1995.
37. Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill Book Comp., New York, 1978.
38. D.C. Kozen, *Automata and Computability*, Springer-Verlag, New York, 1997.
39. W. Kuich, A. Salomaa, *Semirings, Automata, Languages*, Springer-Verlag, Berlin, Heidelberg, New York, 1986.
40. P. Linz, *An Introduction to Formal Languages and Automata*, D.C. Heath and Co., Lexington, Mass., 1990.
41. M. Lothaire, *Combinatorics on Words*, Addison-Wesley, Reading, Mass., 1983.
42. M. Lothaire, *Algebraic Combinatorics on Words*, Cambridge University Press, 1997.
43. C. Martín-Vide, V. Mitrana, eds., *Where Mathematics, Computer Science, Linguistics, and Biology Meet*, Kluwer, Dordrecht, 2000.
44. C. Martín-Vide, V. Mitrana, eds., *Grammars and Automata for String Processing: From Mathematics and Computer Science to Biology, and Back*, Taylor and Francis, London, 2002.
45. C. Martín-Vide, Gh. Păun, eds., *Recent Topics in Mathematical and Computational Linguistics*, Ed. Academiei, Bucureşti, 2000.

46. C. Martín-Vide, V. Mitrana, Gh. Păun, eds., *Formal Languages and Applications*, Springer-Verlag, Berlin, 2004.
47. H. Maurer, *Theoretische Grundlagen der Programmiersprachen*, Hochschultaschenbücher 404, Bibliographisches Inst., 1969.
48. A. Meduna, *Automata and Languages*, Springer-Verlag, London, 2000.
49. M. Minsky, *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, NJ, 1967.
50. J.N. Mordenson, D.S. Malik, *Fuzzy Automata and Languages*, Chapman & Hall/CRC, London, 2002.
51. A. Paz, *Introduction to Probabilistic Automata*, Academic Press, New York, 1971.
52. Gh. Păun, *Recent Results and Problems in Formal Language Theory*, The Scientific and Encyclopaedic Publ. House, Bucharest, 1984 (in Romanian).
53. Gh. Păun, ed., *Mathematical Aspects of Natural and Formal Languages*, World Scientific, Singapore, 1994.
54. Gh. Păun, ed., *Mathematical Linguistics and Related Topics*, The Publ. House of the Romanian Academy, Bucharest, 1995.
55. Gh. Păun, *Marcus Contextual Grammars*, Kluwer, Dordrecht, 1997.
56. Gh. Păun, *Membrane Computing. An Introduction*, Springer-Verlag, Berlin, 2002.
57. Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms*, Springer-Verlag, Berlin, 1998.
58. Gh. Păun, G. Rozenberg, A. Salomaa, eds., *Current Trends in Theoretical Computer Science. Entering the 21st Century*, World Scientific, Singapore, 2001.
59. Gh. Păun, A. Salomaa, eds., *New Trends in Formal Languages: Control, Cooperation, Combinatorics*, *Lecture Notes in Computer Science* 1218, Springer-Verlag, Berlin, 1997.
60. Gh. Păun, A. Salomaa, eds., *Grammatical Models of Multi-Agent Systems*, Gordon and Breach, London, 1999.
61. J.E. Pin, *Varieties of Formal Languages*, Plenum Press, Oxford, 1986.
62. G.E. Revesz, *Introduction to Formal Languages*, McGraw-Hill Book Comp., New York, 1983.
63. G. Rozenberg, A. Salomaa, *The Mathematical Theory of L Systems*, Academic Press, New York, 1980.
64. G. Rozenberg, A. Salomaa, *Cornerstones of Undecidability*, Prentice Hall, New York, 1994.
65. G. Rozenberg, A. Salomaa, eds., *Developments in Language Theory*, World Scientific, Singapore, 1994.
66. G. Rozenberg, A. Salomaa, eds., *Handbook of Formal Languages*, Springer-Verlag, Berlin, 3 volumes, 1997.
67. A. Salomaa, *Theory of Automata*, Pergamon, Oxford, 1969.
68. A. Salomaa, *Formal Languages*, Academic Press, New York, London, 1973.
69. A. Salomaa, *Jewels of Formal Language Theory*, Computer Science Press, Rockville, 1981.

70. A. Salomaa, *Computation and Automata*, Cambridge Univ. Press, Cambridge, 1985.
71. A. Salomaa, M. Soittola, *Automata-Theoretic Aspects of Formal Power Series*, Springer-Verlag, Berlin, New York, 1978.
72. A. Salomaa, D. Wood, S. Yu, eds., *A Half-Century of Automata Theory*, World Scientific, Singapore, 2001.
73. D. Simovici, R.L. Tenney, *Theory of Formal Languages With Applications*, World Scientific, Singapore, 1999.
74. S. Sippu, E. Soisalon-Soininen, *Parsing Theory. Vol. I: Languages and Parsing*, Springer-Verlag, Berlin, Heidelberg, 1988.
75. M. Sipser, *Introduction to the Theory of Computation*, PWS Publishing Company, Boston, 1997.
76. H.J. Shyr, *Free Monoids and Languages*, Hon Min Book Comp., Taichung, 1991.
77. R.G. Taylor, *Models of Computation and Formal Languages*, Oxford University Press, 1998.
78. D. Wood, *Grammar and L Forms. An Introduction*, Springer-Verlag, Berlin, 1980 (*Lecture Notes in Computer Science*, 91).
79. D. Wood, *Theory of Computation*, Harper and Row, New York, 1987.

2

Open Problems on Partial Words*

Francine Blanchet-Sadri

Department of Computer Science, University of North Carolina
P.O. Box 26170, Greensboro, NC 27402–6170, USA
blanchet@uncg.edu

2.1 Introduction

Combinatorics on *words*, or sequences or strings of symbols over a finite alphabet, is a rather new field although the first papers were published at the beginning of the 20th century [120, 121]. The interest in the study of combinatorics on words has been increasing since it finds applications in various research areas of mathematics, computer science, and biology where the data can be easily represented as words over some alphabet. Such areas may be concerned with algorithms on strings [38, 48, 50, 51, 52, 69, 72, 84, 102, 118], semigroups, automata and languages [2, 45, 55, 75, 82, 92, 93], molecular genetics [78], or codes [5, 73, 79].

Motivated by molecular biology of nucleic acids, Berstel and Boasson introduced in 1999 the notion of *partial words* which are sequences that may contain a number of "do not know" symbols or "holes" [4]. *DNA molecules* are the carriers of the genetic information in almost all organisms. Let us look into the structure of such a molecule. A *single stranded DNA molecule* or a *DNA strand* may be viewed as a sequence over the alphabet consisting of the four *nucleotides*: a (adenine), c (cytosine), g (guanine), and t (thymine). Each strand has two different ends: the $3'$ end, and the $5'$ end. The familiar double helix of DNA, which was discovered by Watson and Crick, arises by the bonding of a strand in the $5' - 3'$ direction with another strand in the $3' - 5'$ direction with the restriction that adenine bonds with thymine, and cytosine bonds with guanine. Such a bonding gives rise to a *double stranded DNA molecule* as in the figure

$$\begin{array}{c} 5'\text{ - } c\ c\ a\ c\ c\ t\ c\ g\ a\ c\ c\ c\ t\ c \text{ - } 3' \\ 3'\text{ - } g\ g\ t\ g\ g\ a\ g\ c\ t\ g\ g\ g\ a\ g \text{ - } 5' \end{array}$$

Because of Watson-Crick's complementarity (a bonds to only t, and c bonds only to g), we can view double stranded DNA sequences as single stranded

*This material is based upon work supported by the National Science Foundation under Grant Nos. CCF–0207673 and DMS–0452020.

strings by keeping the strand in the $5' - 3'$ direction. The molecule in the example above can be viewed as

$$5' \text{ - } ccacctcgaccctc \text{ - } 3'$$

or simply as *ccacctcgaccctc*, a word over the alphabet $\{a, c, g, t\}$. However bonding is not always perfect in nature as in the figure

$$\begin{array}{l} 5' \text{ - } c\ c\ a\ c\ c\ t\ c\ g\ a\ c\ c\ c\ t\ c \text{ - } 3' \\ 3' \text{ - } g\ g\ t\ t\ g\ a\ g\ c\ c\ g\ g\ g\ a\ g \text{ - } 5' \end{array}$$

where there is an occurrence of c paired with t, and an occurrence of a paired with c. In such a case, we can view the molecule as $5' - cca{\diamond}ctcg{\diamond}ccctc - 3'$ or as $cca{\diamond}ctcg{\diamond}ccctc$, where the $\diamond$'s stand for "do not know" symbols also called "holes". Thus, the latter example gives rise to a partial word with two holes over the alphabet $\{a, c, g, t\}$. Processes in molecular biology can be seen as operations on DNA sequences [72, 112]. If a set of DNA molecules fulfilling a certain property has changed a little bit after some time or under some influence, it is important to know whether the desired property still holds [91].

Several interesting combinatorial properties of partial words have been investigated and connections have been made with problems in graph theory and number theory, in particular, with problems concerning primitive sets of integers [23, 24], lattices [23, 24], partitions of integers and their generalizations [14], chromatic polynomials [23], Sudoku games [107], vertex connectivity in graphs [12, 29], etc. Partial words are useful in a new generation of pattern matching algorithms that search for local similarities between sequences. In this area, they are called "spaced seeds" and a lot of work has been dedicated to their influence on the algorithms' performance [40, 66, 83, 97, 103, 104]. Partial words have the potential for impacts in bio-inspired computing where they have been considered, in particular, for finding good encodings for DNA computations [90].

We provide here a few bibliographic remarks. Lothaire's first book *Combinatorics on Words* appeared in 1983 [92], while recent developments culminated in a second book *Algebraic Combinatorics on Words* which appeared in 2002 [93] and in a third book which appeared in 2005 [94]. Several books have appeared quite recently that emphasize connections of combinatorics on words to several research areas. We mention the book of Allouche and Shallit where the emphasis is on automata theory [2], the book of Crochemore and Rytter where the emphasis is on string algorithms [52], the book of Gusfield where the emphasis is on algorithms related to biology [72], the book of de Luca and Varrichio where the emphasis is on algebra [55], and finally the book of Blanchet-Sadri where the emphasis is on partial words [10].

Research in combinatorics on partial words is underway where there are several open problems that lay unexplored. After reviewing basic concepts on words and partial words in Section 2.2, we will discuss some of these open problems which we have divided into sections: 2.3–2.5 study extensions to partial words of three basic classical results on periodicity of words: The theorem of Fine and Wilf which considers the simultaneous occurrence of different periods in one word [67], the critical factorization theorem which relates local and global periodicity of words [43], and a theorem of Guibas and Odlyzko which gives the structure of the set of periods of a word [71]. Section 2.6 deals with the two word properties of primitiveness and borderedness and is concerned, in particular, with the counting of primitive and unbordered partial words. Section 2.7 solves some equations on partial words. Here the notion of "equality" is replaced by that of "*compatibility*". Section 2.8 studies the concept of unavoidable set of partial words, while Section 2.9 develops square- and overlap-freeness of partial words. Finally, Section 2.10 discusses some other open problems related to codes of partial words, punctured languages, and tiling periodicity.

2.2 Preliminaries

This section is devoted to reviewing basic concepts on words and partial words.

2.2.1 Words

Let A be a nonempty finite set of symbols called an *alphabet*. Symbols in A are called *letters* and any finite sequence of letters is called a *word* over A. The *empty word*, that is, the word containing no letter, is denoted by ε. For any word u over A, $|u|$ denotes the number of letters occurring in u and is called the *length* of u. In particular, $|\varepsilon| = 0$. The set of all words over A is denoted by A^*. If we define the operation of two words u and v of A^* by juxtaposition (or concatenation), then A^* is a monoid with identity ε. We call $A^+ = A^* \setminus \{\varepsilon\}$ the *free semigroup generated by* A and A^* *the free monoid generated by* A. The set A^* can also be viewed as $\bigcup_{n\geq 0} A^n$ where $A^0 = \{\varepsilon\}$ and A^n is the set of all words of length n over A.

A word of length n over A can be defined by a total function $u : \{0, \ldots, n-1\} \to A$ and is usually represented as $u = a_0 a_1 \ldots a_{n-1}$ with $a_i \in A$. A *period* of u is a positive integer p such that $a_i = a_{i+p}$ for $0 \leq i < n - p$. For a word u, the powers of u are defined inductively by $u^0 = \varepsilon$ and, for any $i \geq 1$, $u^i = uu^{i-1}$. The set of symbols occurring in a word u is denoted by $\alpha(u)$. The *reversal* of u, denoted by $\text{rev}(u)$, is defined as follows: If $u = \varepsilon$, then $\text{rev}(\varepsilon) = \varepsilon$, and if $u = a_0 a_1 \ldots a_{n-1}$, then $\text{rev}(u) = a_{n-1} \ldots a_1 a_0$. A word u is a *factor* of the word v if there exist words x, y such that $v = xuy$. The factor u is called *proper* if $u \neq \varepsilon$ and $u \neq v$. The word u is a *prefix* (respectively, *suffix*) of v if $x = \varepsilon$ (respectively, $y = \varepsilon$).

A nonempty word u is *primitive* if there exists no word v such that $u = v^i$ with $i \geq 2$. Note the fact that the empty word is not primitive. If u is a nonempty word, then there exist a unique primitive word v and a unique positive integer i such that $u = v^i$.

2.2.2 Partial Words

A *partial word* u of length n over A is a partial function $u : \{0, \ldots, n-1\} \to A$. For $0 \leq i < n$, if $u(i)$ is defined, then i belongs to the *domain* of u, denoted by $i \in D(u)$, otherwise i belongs to the *set of holes* of u, denoted by $i \in H(u)$. A word over A is a partial word over A with an empty set of holes (we sometimes refer to words as *full* words). The length of u or n is denoted by $|u|$.

If u is a partial word of length n over A, then the *companion* of u, denoted by $u_\diamond$, is the total function $u_\diamond : \{0, \ldots, n-1\} \to A \cup \{\diamond\}$ defined by

$$u_\diamond(i) = \begin{cases} u(i) & \text{if } i \in D(u) \\ \diamond & \text{otherwise} \end{cases}$$

The bijectivity of the map $u \mapsto u_\diamond$ allows us to define for partial words concepts such as concatenation, powers, etc in a trivial way. The word $u_\diamond = abb\diamond bbcb$ is the companion of the partial word u of length $|u| = 8$ where $D(u) = \{0, 1, 2, 4, 5, 6, 7\}$ and $H(u) = \{3\}$. For convenience, we will refer to a partial word over A as a word over the enlarged alphabet $A_\diamond = A \cup \{\diamond\}$, where the additional symbol $\diamond$ plays the special role of a "do not know" symbol or "hole". This allows us to say for example "the partial word $aba\diamond aa\diamond$" instead of "the partial word with companion $aba\diamond aa\diamond$". The set of all partial words over A with an arbitrary number of holes is denoted by $A_\diamond^*$ which is a monoid under the operation of concatenation where ε serves as the identity.

A *(strong) period* of a partial word u over A is a positive integer p such that $u(i) = u(j)$ whenever $i, j \in D(u)$ and $i \equiv j \bmod p$. In such a case, we call u *(strongly) p-periodic*. Similarly, a *weak period* of u is a positive integer p such that $u(i) = u(i + p)$ whenever $i, i + p \in D(u)$. In such a case, we call u *weakly p-periodic*. The partial word $abb\diamond bbcbb$ is weakly 3-periodic but is not strongly 3-periodic. The latter shows a difference between partial words and full words since every weakly p-periodic full word is strongly p-periodic. Another difference worth noting is the fact that even if the length of a partial word u is a multiple of a weak period of u, then u is not necessarily a power of a shorter partial word. The minimum period of u is denoted by $p(u)$, and the minimum weak period by $p'(u)$. The set of all periods (respectively, weak periods) of u is denoted by $\mathcal{P}(u)$ (respectively, $\mathcal{P}'(u)$).

For a partial word u, positive integer p and integer $0 \leq i < p$, define

$$u_{i,p} = u(i)u(i + p)u(i + 2p) \ldots u(i + jp)$$

where j is the largest nonnegative integer such that $i + jp < |u|$. Then u is (strongly) p-periodic if and only if $u_{i,p}$ is (strongly) 1-periodic for all $0 \leq i < p$, and u is weakly p-periodic if and only if $u_{i,p}$ is weakly 1-periodic for all $0 \leq i < p$. Strongly 1-periodic partial words as well as the full factors of weakly 1-periodic partial words are over a singleton alphabet.

If u and v are two partial words of equal length, then u is said to be contained in v, denoted by $u \subset v$, if all elements in $D(u)$ are in $D(v)$ and

$u(i) = v(i)$ for all $i \in D(u)$. The order $u \subset v$ on partial words is obtained when we let $\diamond < a$ and $a \leq a$ for all $a \in A$. For example, $a\diamond b\diamond \not\subset a\diamond\diamond b$ and $a\diamond b\diamond \not\subset a\diamond ab$, while $a\diamond b\diamond \subset a\diamond bb$.

A partial word u is *primitive* if there exists no word v such that $u \subset v^i$ with $i \geq 2$. Note that if v is primitive and $v \subset u$, then u is primitive as well. It was shown in [9] that if u is a nonempty partial word, then there exist a primitive word v and a positive integer i such that $u \subset v^i$. However uniqueness does not hold as seen with the partial word $u = \diamond a$ where $u \subset a^2$ and $u \subset ba$ for distinct letters a, b.

Partial words u and v are *compatible*, denoted by $u \uparrow v$, if there exists a partial word w such that $u \subset w$ and $v \subset w$. In other words, $u(i) = v(i)$ for every $i \in D(u) \cap D(v)$. Note that for full words, the notion of compatibility is simply that of equality. For example, $a\diamond b\diamond a\diamond \not\uparrow a\diamond\diamond cbb$ but $a\diamond bbc\diamond \uparrow \diamond bb\diamond c\diamond$.

In the rest of this section, we discuss commutativity and conjugacy in the context of partial words.

Let us start with commutativity. The case of full words is well known and is stated in the following theorem.

Theorem 1. *Let x and y be nonempty words. Then $xy = yx$ if and only if there exists a word z such that $x = z^m$ and $y = z^n$ for some integers m, n.*

For nonempty partial words x and y, if there exist a word z and integers m, n such that $x \subset z^m$ and $y \subset z^n$, then $xy \subset z^{m+n}$, $yx \subset z^{m+n}$, and $xy \uparrow yx$. The converse is not true in general: if $x = \diamond bb$ and $y = abb\diamond$, then

$$xy = \diamond bbabb\diamond \uparrow abb\diamond\diamond bb = yx$$

but no desired z exists.

Let us first examine the case of one hole.

Theorem 2. *[4] Let x, y be nonempty partial words such that xy has at most one hole. If $xy \uparrow yx$, then there exists a word z such that $x \subset z^m$ and $y \subset z^n$ for some integers m, n.*

Now, for the case of an arbitrary number of holes, let k, l be positive integers satisfying $k \leq l$. For $0 \leq i < k + l$, define

$$\mathrm{seq}_{k,l}(i) = (i_0, i_1, i_2, \ldots, i_n, i_{n+1})$$

where $i_0 = i = i_{n+1}$; for $1 \leq j \leq n$, $i_j \neq i$; and for $1 \leq j \leq n + 1$,

$$i_j = \begin{cases} i_{j-1} + k \text{ if } i_{j-1} < l \\ i_{j-1} - l \text{ otherwise} \end{cases}$$

For example, $\mathrm{seq}_{6,8}(0) = (0, 6, 12, 4, 10, 2, 8, 0)$.

Definition 1. *[11] Let k, l be positive integers satisfying $k \leq l$ and let z be a partial word of length $k + l$. We say that z is (k, l)-special if there exists $0 \leq i < k$ such that $seq_{k,l}(i) = (i_0, i_1, i_2, \ldots, i_n, i_{n+1})$ contains (at least) two positions that are holes of z while $z(i_0)z(i_1)z(i_2)\ldots z(i_{n+1})$ is not 1-periodic.*

Example 1. Let $z = cbca\diamond\diamond cbc\diamond caca$, and let $k = 6$ and $l = 8$ so $|z| = k+l$. We wish to determine if z is $(6, 8)$-special. We already calculated $\mathrm{seq}_{6,8}(0)$ and

$$\begin{array}{cccccccc} z(0) & z(6) & z(12) & z(4) & z(10) & z(2) & z(8) & z(0) \\ c & c & c & \diamond & c & c & c & c \end{array}$$

This sequence does not satisfy the definition, and so we must continue with calculating $\mathrm{seq}_{6,8}(1) = (1, 7, 13, 5, 11, 3, 9, 1)$. The corresponding letter sequence is

$$\begin{array}{cccccccc} z(1) & z(7) & z(13) & z(5) & z(11) & z(3) & z(9) & z(1) \\ b & b & a & \diamond & a & a & \diamond & b \end{array}$$

Here we have two positions in the sequence which are holes, and the sequence is not 1-periodic. Hence, z is $(6, 8)$-special.

Under the extra condition that xy is not $(|x|, |y|)$-special, an extension of Theorem 2 holds when xy has an arbitrary number of holes.

Theorem 3. *[11] Let x, y be nonempty partial words such that $|x| \leq |y|$. If $xy \uparrow yx$ and xy is not $(|x|, |y|)$-special, then there exists a word z such that $x \subset z^m$ and $y \subset z^n$ for some integers m, n.*

Now, let us discuss conjugacy. Again, the case of full words is well known.

Theorem 4. *Let x, y, z ($x \neq \varepsilon$ and $y \neq \varepsilon$) be words such that $xz = zy$. Then $x = uv$, $y = vu$, and $z = (uv)^n u$ for some words u, v and integer $n \geq 0$.*

For example, if $x = abcda$, $y = daabc$, and $z = abc$, then $xz = zy$ because $(abcda)(abc) = (abc)(daabc)$. Here $u = abc$, $v = da$, and $n = 0$.

The case of partial words is more subtle.

Theorem 5. *[28] Let x, y, z be partial words with x, y nonempty. If $xz \uparrow zy$ and $xz \vee zy$ is $|x|$-periodic, then there exist words u, v such that $x \subset uv$, $y \subset vu$, and $z \subset (uv)^n u$ for some integer $n \geq 0$.*

To illustrate Theorem 5, let $x = \diamond ba$, $y = \diamond b\diamond$, and $z = b\diamond ab\diamond\diamond\diamond\diamond$. Then we have

$$\begin{aligned} xz &= \diamond\ b\ a\ b\ \diamond\ a\ b\ \diamond\ \diamond\ \diamond\ \diamond \\ zy &= b\ \diamond\ a\ b\ \diamond\ \diamond\ \diamond\ \diamond\ \diamond\ b\ \diamond \\ xz \vee zy &= b\ b\ a\ b\ \diamond\ a\ b\ \diamond\ \diamond\ b\ \diamond \end{aligned}$$

It is clear that $xz \uparrow zy$ and $xz \vee zy$ is $|x|$-periodic. Putting $u = bb$ and $v = a$, we can verify that the conclusion does indeed hold.

Corollary 1. *[28] Let x, y be nonempty partial words, and let z be a full word. If $xz \uparrow zy$, then there exist words u, v such that $x \subset uv$, $y \subset vu$, and $z \subset (uv)^n u$ for some integer $n \geq 0$.*

Note that the above Corollary does not necessarily hold if z is not full even if x, y are full. The partial words $x = a, y = b$, and $z = \diamond bb$ provide a counterexample.

Two conjugacy theorems follow without any restriction on z.

Theorem 6. *[13] Let x, y and z be partial words such that $|x| = |y| > 0$. Then $xz \uparrow zy$ if and only if xzy is weakly $|x|$-periodic.*

Theorem 7. *[13]*

Let x, y and z be partial words such that $|x| = |y| > 0$. Then the following hold:

1. *If $xz \uparrow zy$, then xz and zy are weakly $|x|$-periodic.*
2. *If xz and zy are weakly $|x|$-periodic and $\lfloor \frac{|z|}{|x|} \rfloor > 0$, then $xz \uparrow zy$.*

The assumption $\lfloor \frac{|z|}{|x|} \rfloor > 0$ is necessary. To see this, consider $x = aa$, $y = ba$ and $z = a$. Here, xz and zy are weakly $|x|$-periodic, but $xz \not\uparrow zy$.

2.3 Periods in Partial Words

Notions and techniques related to periodic structures in words find important applications in virtually every area of theoretical and applied computer science, notably in text processing [51, 52], data compression [49, 119, 123], coding [5], computational biology [39, 72, 100, 112], string searching and pattern matching algorithms [38, 50, 51, 52, 69, 72, 84, 102]. Repeated patterns and related phenomena in words have played over the years a central role in the development of combinatorics on words, and have been highly valuable tools for the design and analysis of algorithms [45, 92, 93, 94]. In many practical applications, such as DNA sequence analysis, repetitions admit a certain variation between copies of the repeated pattern because of errors due to mutation, experiments, etc. Approximate repeated patterns, or repetitions where errors are allowed, are playing a central role in different variants of string searching and pattern matching problems [85, 86, 87, 88, 111]. Partial words have acquired great importance in this context.

The notion of *period* is central in combinatorics on words and there are many fundamental results on periods of words. Among them is the well known periodicity result of Fine and Wilf [67] which intuitively determines how far two periodic events have to match in order to guarantee a common period. More precisely, any word u having two periods p, q and length at least $p + q - \gcd(p, q)$ has also the greatest common divisor of p and q, $\gcd(p, q)$, as a period. The bound $p + q - \gcd(p, q)$ is optimal since counterexamples can be provided for shorter lengths, that is, there exists an *optimal* word of length $p + q - \gcd(p, q) - 1$ having p and q as periods but not having $\gcd(p, q)$ as period [45]. Extensions of Fine and Wilf's result to more than two periods are given in [42, 47, 80, 122]. For instance, in [47], Constantinescu and Ilie give an extension for an arbitrary number of periods and prove that their bounds are optimal.

Fine and Wilf's result has been generalized to partial words in two ways:

- First, any partial word u with h holes and having two weak periods p, q and length at least the so-denoted $l(h, p, q)$ has also strong period $\gcd(p, q)$ provided u satisfies the condition of not being (h, p, q)-*special* (this concept will be defined below). This extension was done for one hole by Berstel and Boasson where the class of $(1, p, q)$-special partial words is empty [4]; for two or three holes by Blanchet-Sadri and Hegstrom [25]; and for an arbitrary number of holes by Blanchet-Sadri [8]. Elegant closed formulas for the bounds $l(h, p, q)$ were given and shown to be optimal.
- Second, any partial word u with h holes and having two strong periods p, q and length at least the so-denoted $L(h, p, q)$ has also strong period $\gcd(p, q)$. The study of the bounds $L(h, p, q)$ was initiated by Shur and Gamzova [114]. In particular, they gave a closed formula for $L(h, p, q)$ in the case where $h = 2$ (the cases where $h = 0$ or $h = 1$ are implied by the above mentioned results). In [12], Blanchet-Sadri, Bal and Sisodia gave closed formulas for the optimal bounds $L(h, p, q)$ in the case where $p = 2$ and also in the case where q is "large". In addition, they gave upper bounds when q is "small" and $h = 3, 4, 5$ or 6. Their proofs are based on connectivity in a graph $G_{(p,q)}(u)$ associated with a given p- and q-periodic partial word u. More recently, in [29], Blanchet-Sadri, Mandel and Sisodia pursue by studying two types of vertex connectivity on $G_{(p,q)}(u)$: the so-called modified degree connectivity and r-set connectivity where $r = q \bmod p$. As a result, they give an efficient algorithm for computing $L(h, p, q)$, and manage to give closed formulas in several cases including the $h = 3$ and $h = 4$ cases.

In this section, we discuss in details the two ways Fine and Wilf's periodicity result has been extended to partial words: Section 2.3.1 discusses the weak periodicity generalizations and Section 2.3.2 the strong periodicity generalizations. For easy reference, we recall Fine and Wilf's result.

Theorem 8. *[67]*
Let p and q be positive integers. If a full word u is p-periodic and q-periodic and $|u| \geq p + q - \gcd(p, q)$, then u is $\gcd(p, q)$-periodic.

2.3.1 Weak Periodicity

In this section, we review the generalizations related to weak periodicity.

We first recall Berstel and Boasson's result for partial words with exactly one hole where the bound $p + q$ is optimal.

Theorem 9. *[4]*

Let p and q be positive integers satisfying $p < q$. Let u be a partial word with one hole. If u is weakly p-periodic and weakly q-periodic and $|u| \geq l(1, p, q) = p + q$, then u is strongly $\gcd(p, q)$-periodic.

When we discuss partial words with $h \geq 2$ holes, we need the extra assumption of u not being (h, p, q)-*special* for a similar result to hold true. Indeed, if p and q are positive integers satisfying $p < q$ and $\gcd(p, q) = 1$, then the infinite sequence $(ab^{p-1}\diamond b^{q-p-1}\diamond b^n)_{n>0}$ consists of $(2, p, q)$-special partial words with two holes that are weakly p-periodic and weakly q-periodic but not $\gcd(p, q)$-periodic.

In order to define the concept of (h, p, q)-speciality, note that a partial word u that is weakly p-periodic and weakly q-periodic can be represented as a 2-dimensional structure. Consider for example the partial word

$$w = ababa\diamond\diamond\diamond bab\diamond bb\diamond bbbbbbbb$$

where $p = 2$ and $q = 5$. The array looks like:

$$\begin{array}{cccccc}
 & u(0) & u(5) & u(10) & u(15) & u(20) \\
 & u(2) & u(7) & u(12) & u(17) & u(22) \\
 & u(4) & u(9) & u(14) & u(19) & \\
u(1) & u(6) & u(11) & u(16) & u(21) & \\
u(3) & u(8) & u(13) & u(18) & u(23) &
\end{array}$$

and its corresponding array of symbols looks like:

$$\begin{array}{cccccc}
 & a & \diamond & b & b & b \\
 & a & \diamond & b & b & b \\
 & a & a & \diamond & b & \\
b & \diamond & \diamond & b & b & \\
b & b & b & b & b &
\end{array}$$

In general, if $\gcd(p, q) = d$, we get d arrays. Each of these arrays is associated with a subgraph $G = (V, E)$ of $G_{(p,q)}(u)$ as follows: V is the subset of $D(u)$ comprising the defined positions of u within the array, and $E = E_p \cup E_q$ where $E_p = \{\{i, i-p\} \mid i, i-p \in V\}$ and $E_q = \{\{i, i-q\} \mid i, i-q \in V\}$. For $0 \leq j < \gcd(p, q)$, the subgraph of $G_{(p,q)}(u)$ corresponding to

$$D(u) \cap \{i \mid i \geq 0 \text{ and } i \equiv j \bmod \gcd(p, q)\}$$

will be denoted by $G^j_{(p,q)}(u)$. Whenever $\gcd(p, q) = 1$, $G^0_{(p,q)}(u)$ is just $G_{(p,q)}(u)$. Referring to the partial word w above, the graph $G_{(2,5)}(w)$ is disconnected (w is $(5, 2, 5)$-special). Here, the $\diamond$'s isolate the a's from the b's.

We now define the concept of speciality.

Definition 2. *[8]*

Let p and q be positive integers satisfying $p < q$, and let h be a nonnegative integer. Let

$$l(h,p,q) = \begin{cases} (\frac{h}{2}+1)(p+q) - \gcd(p,q) & \textit{if } h \textit{ is even} \\ (\lfloor \frac{h}{2} \rfloor + 1)(p+q) & \textit{otherwise} \end{cases}$$

Let u be a partial word with h holes of length at least $l(h,p,q)$. Then u is (h,p,q)-special if $G^{j}_{(p,q)}(u)$ is disconnected for some $0 \leq j < \gcd(p,q)$.

It turns out that the bound $l(h,p,q)$ is optimal for a number of holes h.

Theorem 10. *[8]*

Let p and q be positive integers satisfying $p < q$, and let u be a non (h,p,q)-special partial word with h holes. If u is weakly p-periodic and weakly q-periodic and $|u| \geq l(h,p,q)$, then u is strongly gcd(p,q)-periodic.

In [33], progress was made towards allowing an arbitrary number of holes and an arbitrary number of weak periods. There, the authors proved that any partial word u with h holes and having weak periods $p_1, \ldots, p_n$ and length at least the so-denoted $l(h, p_1, \ldots, p_n)$ has also strong period $\gcd(p_1, \ldots, p_n)$ provided u satisfies some criteria. In addition to speciality, they discovered that the concepts of *intractable* period sets and *interference* between periods play a role.

Open problem 1 *Give an algorithm which given a number of holes h and weak periods $p_1, \ldots, p_n$, computes the optimal bound $l(h, p_1, \ldots, p_n)$ and an optimal partial word for that bound (a partial word u with h holes of length $l(h, p_1, \ldots, p_n) - 1$ is optimal for the bound $l(h, p_1, \ldots, p_n)$ if $p_1, \ldots, p_n$ are weak periods of u but $\gcd(p_1, \ldots, p_n)$ is not a strong period of u).*

Open problem 2 *Give closed formulas for the bounds $l(h, p_1, \ldots, p_n)$.*

The optimality proof will probably be based on results of graphs associated with bounds and tuples of weak periods.

2.3.2 Strong Periodicity

In this section, we review the generalizations related to strong periodicity. There exists an integer L such that if a partial word u with h holes has strong periods p, q satisfying $p < q$ and $|u| \geq L$, then u has strong period $\gcd(p,q)$ ($L(h,p,q)$ is the smallest such integer L) [115]. Recall that $L(0,p,q) = p + q - \gcd(p,q)$.

The following result is a direct consequence of Berstel and Boasson's result.

Theorem 11. *[4] The equality $L(1,p,q) = p + q$ holds.*

For $h = 2, 3$ or 4, we have the following results.

Theorem 12. *[114, 115] The equality* $L(2,p,q) = 2p + q - \gcd(p,q)$ *holds.*

Theorem 13. *[29] The following equality holds:*

$$L(3,p,q) = \begin{cases} 2q + p & \text{if } q - p < \frac{p}{2} \\ 4p & \text{if } \frac{p}{2} < q - p < p \\ 2p + q & \text{if } p < q - p \end{cases}$$

Theorem 14. *[29] The following equality holds:*

$$L(4,p,q) = \begin{cases} q + 3p - \gcd(p,q) & \text{if } q - p < \frac{p}{2} \\ q + 3p & \text{if } \frac{p}{2} < q - p < p \\ q + 3p - \gcd(p,q) & \text{if } p < q - p \end{cases}$$

Other results follow.

Theorem 15. *[12, 113, 114, 115] The equality* $L(h,2,q) = (2\lfloor\frac{h}{q}\rfloor + 1)q + h \bmod q + 1$ *holds.*

Setting $h = nq+r$ where $0 \leq r < q$, $L(h,2,q) = (2n+1)q+r+1$. Consider the word $u = \diamond^r w(\diamond^q w)^n$ where w is the unique full word of length q having periods 2 and q but not having period 1. Note that u is an optimal word for the bound $L(h,2,q)$. Indeed, $|u| = (2n+1)q + r$, u has h holes, and since w is not 1-periodic, we also have that u is not strongly 1-periodic. It is easy to show that u is strongly 2- and q-periodic.

In [114], the authors proved that if $q > p \geq 3$ and $\gcd(p,q) = 1$ and h is large enough, then

$$\tfrac{pq}{p+q-2}(h+1) \leq L(h,p,q) < \tfrac{pqh}{p+q-2} + 4(q-1)$$

Open problem 3 *Give closed formulas for the bounds* $L(h,p,q)$ *where* $h > 4$.

Any partial word with h holes and having n strong periods $p_1, \ldots, p_n$ and length at least the so-denoted $L(h, p_1, \ldots, p_n)$ has also $\gcd(p_1, \ldots, p_n)$ as a strong period.

Open problem 4 *Give an algorithm which given a number of holes h and strong periods $p_1, \ldots, p_n$, computes the optimal bound $L(h, p_1, \ldots, p_n)$ and an optimal partial word for that bound (a partial word u with h holes of length $L(h, p_1, \ldots, p_n) - 1$ is optimal for the bound $L(h, p_1, \ldots, p_n)$ if $p_1, \ldots, p_n$ are strong periods of u but $\gcd(p_1, \ldots, p_n)$ is not a strong period of u).*

Open problem 5 *Give closed formulas for the bounds* $L(h, p_1, \ldots, p_n)$.

2.4 Critical Factorizations of Partial words

Results concerning periodicity include the well known and fundamental critical factorization theorem, of which several versions exist [43, 45, 60, 61, 59, 92, 93]. It intuitively states that the minimal period (or global period) of a word of length at least two is always locally detectable in at least one position of the word resulting in a corresponding *critical factorization*. More specifically, given a word w and nonempty words u, v satisfying $w = uv$, the *minimal local period* associated to the factorization (u, v) is the length of the shortest square at position $|u| - 1$. It is easy to see that no minimal local period is longer than the global period of the word. The critical factorization theorem shows that critical factorizations are unavoidable. Indeed, for any word, there is always a factorization whose minimal local period is equal to the global period of the word.

More precisely, we consider a word $a_0 a_1 \dots a_{n-1}$ and, for any integer i $(0 \leq i < n-1)$, we look at the shortest repetition (a *square*) *centered* in this position, that is, we look at the shortest (virtual) suffix of $a_0 a_1 \dots a_i$ which is also a (virtual) prefix of $a_{i+1} a_{i+2} \dots a_{n-1}$. The minimal local period at position i is defined as the length of this shortest square. The critical factorization theorem states, roughly speaking, that the global period of $a_0 a_1 \dots a_{n-1}$ is simply the maximum among all minimal local periods. As an example, consider the word $w = babbaab$ with global period 6. The minimal local periods of w are 2, 3, 1, 6, 1 and 3 which means that the factorization $(babb, aab)$ is critical.

Crochemore and Perrin showed that a critical factorization can be found very efficiently from the computation of the maximal suffixes of the word with respect to two total orderings on words: the lexicographic ordering related to a fixed total ordering on the alphabet $\preceq_l$, and the lexicographic ordering obtained by reversing the order of letters in the alphabet $\preceq_r$ [50]. If v denotes the maximal suffix of w with respect to $\preceq_l$ and v' the maximal suffix of w with respect to $\preceq_r$, then let u, u' be such that $w = uv = u'v'$. The factorization (u, v) turns out to be critical when $|v| \leq |v'|$, and the factorization (u', v') is critical when $|v| > |v'|$. There exist linear time (in the length of w) algorithms for such computations [50, 51, 101] (the latter two use the suffix tree construction). Returning to the example above, order the letters of the alphabet by $a \prec b$. Then the maximal suffix with respect to $\preceq_l$ is $v = bbaab$ and with respect to $\preceq_r$ is $v' = aab$. Since $|v| > |v'|$, the factorization $(u', v') = (babb, aab)$ of w is critical.

In [22], Blanchet-Sadri and Duncan extended the critical factorization theorem to partial words with one hole. In this case, the concept of *local period*, which characterizes a local periodic structure at each position of the word, is defined as follows.

Definition 3. *[22] Let w be a nonempty partial word. A positive integer p is called* a local period *of w at position i if there exist partial words u, v, x, y such that $u, v \neq \varepsilon$, $w = uv$, $|u| = i + 1$, $|x| = p$, $x \uparrow y$, and such that one of the following conditions holds for some partial words r, s:*

1. $u = rx$ *and* $v = ys$ *(internal square),*
2. $x = ru$ *and* $v = ys$ *(left-external square if* $r \neq \varepsilon$*),*
3. $u = rx$ *and* $y = vs$ *(right-external square if* $s \neq \varepsilon$*),*
4. $x = ru$ *and* $y = vs$ *(left- and right-external square if* $r, s \neq \varepsilon$*).*

In this context, a factorization is called *critical* if its minimal local period is equal to the minimal weak period of the partial word. As an example, consider the partial word with one hole $w = ba\diamond baab$ with minimal weak period 3. The minimal local periods of w are 2 (left-external square), 1 (internal square), 1 (internal square), 3 (internal square), 1 (internal square) and 3 (right-external square), and both $(ba\diamond b, aab)$ and $(ba\diamond baa, b)$ are critical.

It turns out that for partial words, critical factorizations may be avoidable. Indeed, the partial word *babdaab* has no critical factorization. The class of the so-called *special* partial words with one hole has been described that possibly avoid critical factorizations. Refining the method based on the maximal suffixes with respect to the lexicographic/ reverse lexicographic orderings leads to a version of the critical factorization theorem for the *nonspecial* partial words with one hole whose proof provides an efficient algorithm which, given a partial word with one hole, outputs a critical factorization when one exists or outputs "no such factorization exists".

In [35], Blanchet-Sadri and Wetzler further investigated the relationship between local and global periodicity of partial words: (1) They extended the critical factorization theorem to partial words with an arbitrary number of holes; (2) They characterized precisely the class of partial words that do not admit critical factorizations; and (3) They developed an efficient algorithm which computes a critical factorization when one exists.

Some open problems related to the critical factorization theorem follow.

Open problem 6 *Discover some good criterion for the existence of a critical factorization of an unbordered partial word defined as follows: A nonempty partial word* u *is unbordered if no nonempty partial words* x, v, w *exist such that* $u \subset xv$ *and* $u \subset wx$*.*

Open problem 7 *In the framework of partial words, study the periodicity theorem on words, which has strong analogies with the critical factorization theorem, that was derived in [102].*

In [62], the authors present an $O(n)$ time algorithm for computing *all* local periods of a given word of length n, assuming a constant-size alphabet. This subsumes (but is substantially more powerful than) the computation of the global period of the word and the computation of a critical factorization. Their method consists of two parts: (1) They show how to compute all *internal* minimal squares; and (2) They show how to compute *left-* and *right-external* minimal squares, in particular for those positions for which no internal square has been found.

Open problem 8 *Find the time complexity for the computation of* all *the local periods of a given partial word.*

Now, consider the language

$$CF = \{w \mid w \text{ is a partial word over } \{a, b\} \text{ that has a critical factorization}\}$$

What is the position of CF in the Chomsky hierarchy? It has been proved that CF is a context sensitive language that is not regular. The question whether or not CF is context-free remains open.

Theorem 16. *[36] The language CF is not regular.*

Theorem 17. *[21] The language CF is context sensitive.*

Open problem 9 *Is the language CF context-free?*

2.5 Correlations of Partial Words

In [71], Guibas and Odlyzko consider the period sets of words of length n over a finite alphabet, and specific representations of them, *(auto)correlations*, which are binary vectors of length n indicating the periods. Among the possible 2^n bit vectors, only a small subset are valid correlations. There, they provide characterizations of correlations, asymptotic bounds on their number, and a recurrence for the *population size* of a correlation, that is, the number of words sharing a given correlation. In [108], Rivals and Rahmann show that there is redundancy in period sets and introduce the notion of an *irreducible* period set. They prove that Γ_n, the set of all correlations of length n, is a lattice under set inclusion and does not satisfy the Jordan-Dedekind condition. They propose the first efficient enumeration algorithm for Γ_n and improve upon the previously known asymptotic lower bounds on the cardinality of Γ_n. Finally, they provide a new recurrence to compute the number of words sharing a given period set, and exhibit an algorithm to sample uniformly period sets through irreducible period sets.

In [24], the combinatorics of possible sets of periods and weak periods of partial words were studied in a similar way. There, the notions of binary and ternary correlations were introduced, which are binary and ternary vectors indicating the periods and weak periods of partial words. Extending the result of Guibas and Odlyzko, Blanchet-Sadri, Gafni and Wilson characterized precisely which binary and ternary vectors represent the period and weak period sets of partial words and proved that all valid correlations may be taken over the binary alphabet (the one-hole case was proved earlier in [16]). They showed that the sets of all such vectors of a given length form distributive lattices under suitably defined partial orderings extending results of Rivals and Rahmann. They also showed that there is a well defined minimal set of generators for any binary correlation of length n, called an *irreducible* period set, and demonstrated that these generating sets are the primitive subsets of $\{1, 2, \ldots, n-1\}$. These primitive sets of integers have been extensively studied by many researchers including Erdös [65]. Finally, they investigated the number of partial word correlations of length n. More recently, recurrences for computing the size of populations of partial word correlations were obtained as well as random sampling of period and weak period sets [23].

We first define the *greatest lower bound* of two given partial words u and v of equal length as the partial word $u \wedge v$, where $(u \wedge v) \subset u$ and $(u \wedge v) \subset v$, and if $w \subset u$ and $w \subset v$, then $w \subset (u \wedge v)$. The following example illustrates this new concept which plays a role in this section:

$$\begin{array}{l}
u \quad\;\;\, = a\ b \diamond c\ a\ a\ b \diamond \diamond\ a\ a \\
v \quad\;\;\, = a\ c\ b\ c\ a\ a\ b \diamond b\ b\ a \\
\hline
u \wedge v = a \diamond \diamond\ c\ a\ a\ b \diamond \diamond \diamond\ a
\end{array}$$

The contents of Section 2.5 is as follows: In Section 2.5.1, we give characterizations of correlations. In Section 2.5.2, we provide structural properties of correlations. And in Section 2.5.3, we consider the problem of counting correlations.

2.5.1 Characterizations of Correlations

Full word correlations are vectors representing sets of periods as stated in the following definition.

Definition 4. *Let u be a (full) word. Let v be the binary vector of length $|u|$ for which $v_0 = 1$ and*

$$v_i = \begin{cases} 1 & \text{if } i \in \mathcal{P}(u) \\ 0 & \text{otherwise} \end{cases}$$

We call v the correlation of u.

For instance, the word *abbababbab* has periods 5 and 8 (and 10) and thus has correlation 1000010010.

Binary vectors may satisfy some propagation rules.

Definition 5.
1. *A binary vector v of length n is said to satisfy the forward propagation rule if for all $0 \le p < q < n$ such that $v_p = v_q = 1$ we have that $v_{p+i(q-p)} = 1$ for all $2 \le i < \frac{n-p}{q-p}$.*
2. *A binary vector v of length n is said to satisfy the backward propagation rule if for all $0 \le p < q < \min(n, 2p)$ such that $v_p = v_q = 1$ and $v_{2p-q} = 0$ we have that $v_{p-i(q-p)} = 0$ for all $2 \le i \le \min(\lfloor \frac{p}{q-p} \rfloor, \lfloor \frac{n-p}{q-p} \rfloor)$.*

Note that a binary vector v of length 12 satisfying $v_7 = v_9 = 1$ and the forward propagation rule also satisfies $v_{7+2(9-7)} = v_{11} = 1$. Note also that setting $p = 0$ in the forward propagation rule implies that $v_{iq} = 1$ for all i whenever $v_q = 1$.

Fundamental results on periodicity of words include the following unexpected result of Guibas and Odlyzko which gives a characterization of full word correlations.

Theorem 18. *[71] For correlation v of length n the following are equivalent:*

1. *There exists a word over the binary alphabet with correlation v.*
2. *There exists a word over some alphabet with correlation v.*
3. *The correlation v satisfies the forward and backward propagation rules.*

Corollary 2. *[71] For any word u over an alphabet A, there exists a binary word v of length $|u|$ such that $\mathcal{P}(v) = \mathcal{P}(u)$.*

Now, partial word correlations are defined according to the following definition.

Definition 6. *[24]*

1. *The binary correlation of a partial word u satisfying $\mathcal{P}(u) = \mathcal{P}'(u)$ is the binary vector of length $|u|$ such that $v_0 = 1$ and*

$$v_i = \begin{cases} 1 & \textit{if } \ i \in \mathcal{P}(u) \\ 0 & \textit{otherwise} \end{cases}$$

2. *The ternary correlation of a partial word u is the ternary vector of length $|u|$ such that $v_0 = 1$ and*

$$v_i = \begin{cases} 1 & \textit{if } \ i \in \mathcal{P}(u) \\ 2 & \textit{if } \ i \in \mathcal{P}'(u) \setminus \mathcal{P}(u) \\ 0 & \textit{otherwise} \end{cases}$$

Considering the partial word $abaca\diamond\diamond acaba$ which has periods 9 and 11 (and 12) and strictly weak period 5, its ternary correlation vector is 100002000101.

A characterization of binary correlations follows.

Theorem 19. *[24] Let n be a nonnegative integer. Then for any finite collection $u_1, u_2, \ldots, u_k$ of full words of length n over an alphabet A, there exists a partial word w of length n over the binary alphabet with $\mathcal{P}(w) = \mathcal{P}'(w) = \mathcal{P}(u_1) \cup \mathcal{P}(u_2) \cup \cdots \cup \mathcal{P}(u_k)$.*

Corollary 3. *[24] The set of valid binary correlations over an alphabet A with $\|A\| \geq 2$ is the same as the set of valid binary correlations over the binary alphabet. Phrased differently, if u is a partial word over an alphabet A, then there exists a binary partial word v of length $|u|$ such that $\mathcal{P}(v) = \mathcal{P}(u)$.*

Follows is a characterization of valid ternary correlations.

Theorem 20. *[24] A ternary vector v of length n is the ternary correlation of a partial word of length n over an alphabet A if and only if $v_0 = 1$ and*

1. *If $v_p = 1$, then for all $0 \leq i < \frac{n}{p}$ we have that $v_{ip} = 1$.*
2. *If $v_p = 2$, then there exists some $2 \leq i < \frac{n}{p}$ such that $v_{ip} = 0$.*

The proof is based on the following construction: For $n \geq 3$ and $0 < p < n$, let $n = kp + r$ where $0 \leq r < p$. Then define

$$\omega_p = \begin{cases} (ab^{p-1})^k & \text{if } \ r = 0 \\ (ab^{p-1})^k ab^{r-1} & \text{if } \ r > 0 \end{cases}$$

$$\psi_p = ab^{p-1} \diamond b^{n-p-1}$$

Then given a valid ternary correlation v of length n, the partial word

$$\left(\bigwedge\nolimits_{p>0 | v_p = 1} \omega_p \right) \wedge \left(\bigwedge\nolimits_{p | v_p = 2} \psi_p \right)$$

has ternary correlation v.

For example, given $v = 100002000101$, then $abbbb\diamond bbb\diamond b\diamond$ has correlation v as computed in the following figure:

$$\begin{array}{rl} \omega_9 & = a\ b\ b\ b\ b\ b\ b\ b\ b\ a\ b\ b \\ \omega_{11} & = a\ b\ b\ b\ b\ b\ b\ b\ b\ b\ b\ a \\ \underline{\psi_5} & \underline{= a\ b\ b\ b\ b\ \diamond\ b\ b\ b\ b\ b\ b} \\ & \ \ \ a\ b\ b\ b\ b\ \diamond\ b\ b\ b\ \diamond\ b\ \diamond \end{array}$$

The following corollary implies that every partial word has a "binary equivalent".

Corollary 4. *[24] The set of valid ternary correlations over an alphabet A with $\|A\| \geq 2$ is the same as the set of valid ternary correlations over the binary alphabet. Phrased differently, if u is a partial word over an alphabet A, then there exists a binary partial word v such that*

$$\textit{1. } |v| = |u| \quad \textit{2. } \mathcal{P}(v) = \mathcal{P}(u) \quad \textit{3. } \mathcal{P}'(v) = \mathcal{P}'(u)$$

In [74], Halava, Harju and Ilie gave a simple constructive proof of Theorem 18 which computes v in linear time. This result was later proved for partial words with one hole by extending Halava et al.'s approach [16]. More specifically, given a partial word u with one hole over an alphabet A, a partial word v over the binary alphabet exists such that Conditions 1–3 hold as well as the following condition

$$\textit{4. } H(v) \subset H(u)$$

However, Conditions 1–4 cannot be satisfied simultaneously in the two-hole case. For the partial word $abaca\diamond\diamond acaba$ can be checked by brute force to have no such binary equivalent (although it has the binary equivalent $abbbb\diamond bbb\diamond b\diamond$ as discussed above).

Open problem 10 *Characterize the partial words that have an equivalent over the binary alphabet $\{a, b\}$ satisfying Conditions 1–4.*

Open problem 11 *Design an efficient algorithm for computing a binary equivalent satisfying Conditions 1–4 when such equivalent exists.*

Open problem 12 *Can we always find an equivalent over the ternary alphabet $\{a, b, c\}$ that satisfies Conditions 1–4?*

2.5.2 Structural Properties of Correlations

A result of Rivals and Rahmann [108] states that Γ_n, the set of full word correlations of length n, is a lattice under set inclusion which does not satisfy the Jordan-Dedekind condition, a criterion which stipulates that all maximal chains between two elements of a poset are of equal length. Violating the Jordan-Dedekind condition implies that Γ_n is not distributive.

We now discuss corresponding results for partial words.

Theorem 21. *[24] The set Δ_n of partial word binary correlations of length n is a distributive lattice under $\subset$ where for $u, v \in \Delta_n$, $u \subset v$ if $\mathcal{P}(u) \subset \mathcal{P}(v)$, and thus satisfies the Jordan-Dedekind condition. Here*

1. *The meet of u and v, $u \cap v$, is the unique vector in Δ_n such that $\mathcal{P}(u \cap v) = \mathcal{P}(u) \cap \mathcal{P}(v)$.*
2. *The join of u and v, $u \cup v$, is the unique vector in Δ_n such that $\mathcal{P}(u \cup v) = \mathcal{P}(u) \cup \mathcal{P}(v)$.*
3. *The null element is 10^{n-1}.*
4. *The universal element is 1^n.*

The union of u and v, $u \cup v$, is the vector in Δ'_n defined as $(u \cup v)_i = 0$ if $u_i = v_i = 0$, 1 if either $u_i = 1$ or $v_i = 1$, and 2 otherwise. However, Δ'_n is not closed under union. Considering the example

$$\begin{aligned} u &= 1\,0\,2\,0\,0\,0\,1\,0\,1 \\ v &= 1\,0\,0\,0\,1\,0\,0\,0\,1 \\ (u \cup v) &= 1\,0\,2\,0\,1\,0\,1\,0\,1 \end{aligned}$$

there is no $i \geq 2$ such that $(u \cup v)_{i2} = 0$, and therefore $(u \cup v)$ is not a valid ternary correlation. However 101010101 is valid.

Theorem 22. *[24] The set Δ'_n of partial word ternary correlations of length n is a distributive lattice under $\subset$ where for $u, v \in \Delta'_n$, $u \subset v$ if $u_i = 1$ implies $v_i = 1$ and $u_i = 2$ implies $v_i = 1$ or $v_i = 2$. Here*

1. *The meet of u and v, $u \wedge v$, is the vector $(u \cap v)$ in Δ'_n defined by $\mathcal{P}(u \wedge v) = \mathcal{P}(u) \cap \mathcal{P}(v)$ and $\mathcal{P}'(u \wedge v) = \mathcal{P}'(u) \cap \mathcal{P}'(v)$.*
2. *The join of u and v, $u \vee v$, is the vector in Δ'_n defined by*

$$\mathcal{P}'(u \vee v) = \mathcal{P}'(u) \cup \mathcal{P}'(v)$$
$$\mathcal{P}(u \vee v) = \mathcal{P}(u) \cup \mathcal{P}(v) \cup B(u \cup v)$$

where $B(u \cup v)$ is the set of all $0 < p < n$ such that $(u \cup v)_p = 2$ and there exists no $i \geq 2$ satisfying $(u \cup v)_{ip} = 0$.

In the case of full words, some periods are implied by other periods because of the forward propagation rule. If a twelve-letter full word has periods 7 and 9 then it must also have period 11 since $11 = 7 + 2(9 - 7)$, so $\{7, 9, 11\}$ corresponds to the irreducible period set $\{7, 9\}$. Another result of Rivals and Rahmann shows that the set Λ_n of these irreducible period sets is not a lattice but does satisfy the Jordan-Dedekind condition as a poset [108].

However, forward propagation does not hold in the case of partial words as can be seen with the partial word *abbbbbb⋄b⋄bb* which has periods 7 and 9 but does *not* have period 11. The set $\{7, 9, 11\}$ is irreducible in the sense of partial words, but not in the sense of full words.

This leads us to the definition of generating sets.

Definition 7. *[24] A set $P \subset \{1, 2, \ldots, n-1\}$ generates the correlation $v \in \Delta_n$ provided that for each $0 < i < n$ we have that $v_i = 1$ if and only if there exists $p \in P$ and $0 < k < \frac{n}{p}$ such that $i = kp$.*

For instance, if $v = 1001001101$, then $\{3, 6, 7, 9\}$, $\{3, 6, 7\}$, $\{3, 7, 9\}$, and $\{3, 7\}$ generate v. However, the set $\{3, 7\}$ is the minimal generating set of v.

For every $v \in \Delta_n$ there is a minimal generating set $R(v)$ for v which we call the irreducible period set of v. Namely, this is the set of $p \in \mathcal{P}(v)$ such that for all $q \in \mathcal{P}(v)$ with $q \neq p$ we have that q does not divide p. Denoting by Φ_n the set of irreducible period sets of partial words of length n, we see that there is an obvious bijective correspondence between Φ_n and Δ_n given by

$$\begin{aligned} R : \Delta_n &\to \Phi_n \\ v &\mapsto R(v) \end{aligned}$$

$$\begin{aligned} E : \Phi_n &\to \Delta_n \\ P &\mapsto \textstyle\bigcup_{p \in P} \langle p \rangle_n \end{aligned}$$

For $n \geq 3$, we see immediately that the poset $(\Phi_n, \subset)$ is not a join semilattice since the sets $\{1\}$ and $\{2\}$ will never have a join because $\{1\}$ is always maximal. On the other hand, the following holds.

Proposition 1. *[24] The set Φ_n of irreducible period sets of partial words of length n is a meet semilattice under set inclusion which satisfies the Jordan-Dedekind condition. Here the null element is $\emptyset$, and the meet of two elements is simply their intersection.*

Open problem 13 *Is there an efficient enumeration algorithm for Δ_n?*

2.5.3 Counting Correlations

In this section, we look at the number of valid correlations of a given length. In the case of binary correlations, we give bounds and link the problem to one in number theory. In the case of ternary correlations, we give an exact count.

A primitive set of integers is a subset $S \subset \{1, 2, \ldots\}$ such that for any two distinct elements $s, s' \in S$ we have that neither s divides s' nor s' divides s. The irreducible period sets of correlations $v \in \Delta_n$ are precisely the finite primitive subsets of $\{1, 2, \ldots, n-1\}$.

A result of Erdös can be stated as follows.

Theorem 23. *[65] Let S be a finite primitive set of size k with elements less than n. Then $k \leq \lfloor \frac{n}{2} \rfloor$. Moreover, this bound is sharp.*

This bound shows that the number of binary correlations of length n is at most the number of subsets of $\{1, 2, \ldots, n-1\}$ of size at most $\lfloor \frac{n}{2} \rfloor$. Moreover, the sharpness of the bound gives us that

$$\|\Delta_n\| \geq 2^{\lfloor \frac{n}{2} \rfloor}$$

Thus

$$\frac{\ln 2}{2} \leq \frac{\ln \|\Delta_n\|}{n} \leq \ln 2$$

Open problem 14 *Refine this bound on the cardinality of Δ_n, the set of all partial word binary correlations of length n.*

Guibas and Odlyzko [71] showed that as $n \to \infty$

$$\tfrac{1}{2 \ln 2} + o(1) \leq \tfrac{\ln \|\Gamma_n\|}{(\ln n)^2} \leq \tfrac{1}{2 \ln(\frac{3}{2})} + o(1)$$

and Rivals and Rahmann [108] improved the lower bound to

$$\tfrac{\ln \|\Gamma_n\|}{(\ln n)^2} \geq \tfrac{1}{2 \ln 2} \left(1 - \tfrac{\ln \ln n}{\ln n}\right)^2 + \tfrac{0.4139}{\ln n} - \tfrac{1.47123 \ln \ln n}{(\ln n)^2} + O\left(\tfrac{1}{(\ln n)^2}\right)$$

where Γ_n is the set of all full word correlations of length n. Thus the bounds we give, which show explicitly that $\ln \|\Delta_n\| = \Theta(n)$, demonstrate that the number of partial word binary correlations is much greater than the number of full word correlations.

Lemma 1. *[24]*

1. *Let u be a partial word of length n. Then $p \in \mathcal{P}(u)$ if and only if $ip \in \mathcal{P}'(u)$ for all $0 < i \leq \lfloor \frac{n}{p} \rfloor$.*
2. *If $S \subset \{1, 2, \ldots, n-1\}$, then there exists a unique correlation $v \in \Delta'_n$ such that $\mathcal{P}'(v) \setminus \{n\} = S$.*

Consequently, the cardinality of Δ'_n, the set of valid ternary correlations of length n, is the same as the cardinality of the power set of $\{1, 2, \ldots, n-1\}$, and thus

$$\|\varDelta'_n\| = 2^{n-1}$$

We end this section with the following open problem.

Open problem 15 *Exhibit an algorithm to sample uniformly (weak) period sets through irreducible (weak) period sets.*

2.6 Primitive and Unbordered Partial Words

The two fundamental concepts of *primitiveness* and *borderedness* play an important role in several research areas including coding theory [5, 6, 117], combinatorics on words [45, 92, 93, 94, 96], computational biology [39, 100], data communication [41], data compression [49, 119, 123], formal language theory [57, 58], and text algorithms [38, 50, 51, 52, 69, 72, 84, 102, 118]. A primitive word is one that cannot be written as a power of another word, while an unbordered word is a primitive word such that none of its proper prefixes is one of its suffixes. For example, $\underline{ab}a\underline{ab}$ is bordered with border ab while $abaabb$ is unbordered. The number of primitive and unbordered words of a fixed length over an alphabet of a fixed size is well known, the number of primitive words being related to the Möbius function [92].

In this section, we discuss, in particular, the problems of counting primitive and unbordered partial words.

2.6.1 Primitiveness

A word u is primitive if there exists no word v such that $u = v^i$ with $i \geq 2$. A natural algorithmic problem is how can we decide efficiently whether a given word is primitive. The problem has a brute force quadratic solution: divide the input word into two parts and check whether the right part is a power of the left part. But how can we obtain a faster solution to the problem? Fast algorithms for testing primitivity of words can be based on the combinatorial result that a word u is primitive if and only if u is not an inside factor of its square uu, that is, $uu = xuy$ implies $x = \varepsilon$ or $y = \varepsilon$ [45]. Indeed, any linear time string matching algorithm can be used to test whether the string u is a proper factor of uu. If the answer is no, then the primitiveness of u has been verified [51]. So testing whether or not a word is primitive can be done in linear time in the length of the word.

Primitive partial words were defined in [9]: A partial word u is primitive if there exists no word v such that $u \subset v^i$ with $i \geq 2$. It turns out that a partial word u with one hole is primitive if and only if $uu \uparrow xuy$ for some partial words x, y implies $x = \varepsilon$ or $y = \varepsilon$ [9]. A linear time algorithm for testing primitivity of partial words with one hole can be based on this combinatorial result. As an application, the existence of a binary equivalent for any partial word with one hole satisfying Conditions 1–4 discussed in Section 2.5 was obtained [16]. In [11], a linear time algorithm was described to test primitivity on partial words with more than one hole. Here the concept of speciality related to commutativity on partial words, which was discussed in Section 2.2, is foundational in the design of the algorithm. More precisely, it was shown that if u is a primitive partial word with more than one hole satisfying $uu \uparrow xuy$ for some nonempty partial words x and y such that $|x| < |y|$, then u is $(|x|, |y|)$-special. The partial words $u = ab\diamond bbb\diamond b$, $x = a\diamond$, and $y = c\diamond bbcb$ illustrate the fact that the condition of speciality plays a role when dealing with partial words with more than one hole.

In [19], the very challenging problem of counting the number $P_{h,k}(n)$ (respectively, $P'_{h,k}(n)$) of primitive (respectively, nonprimitive) partial words with h holes of length n over a k-size alphabet was considered. There, formulas for $h = 1$ and $h = 2$ in terms of the well known formula for $h = 0$ were given. Denote by $T_{h,k}(n)$ the sum of $P_{h,k}(n)$ and $P'_{h,k}(n)$.

We first recall the counting for primitive full words. Since there are exactly k^n words of length n over a k-size alphabet and every nonempty word w has a unique primitive root v for which $w = v^{n/d}$ for some divisor d of n, the following relation holds:

$$k^n = \sum_{d|n} P_{0,k}(d)$$

Using the Möbius inversion formula, we obtain the following well-known expression for $P_{0,k}(n)$ [92, 105]:

$$P_{0,k}(n) = \sum_{d|n} \mu(d) k^{n/d}$$

where the Möbius function, denoted by μ, is defined as

$$\mu(n) = \begin{cases} 1 & \text{if } n = 1 \\ (-1)^i & \text{if } n \text{ is a product of } i \text{ distinct primes} \\ 0 & \text{if } n \text{ is divisible by the square of a prime} \end{cases}$$

The cases where $h = 1$ and $h = 2$ are stated in the next two theorems.

Theorem 24. *[19] The equality $P'_{1,k}(n) = nP'_{0,k}(n)$ holds.*

Theorem 25. *[19]*

1. *For an odd positive integer n:*

$$P'_{2,k}(n) = \binom{n}{2} P'_{0,k}(n)$$

2. *For an even positive integer n:*

$$P'_{2,k}(n) = \binom{n}{2} P'_{0,k}(n) - (k-1)T_{1,k}(\frac{n}{2})$$

Open problem 16 *Count the number $P'_{h,k}(n)$ of nonprimitive partial words with h holes of length n over a k-size alphabet for $h > 2$.*

Another problem to investigate is the following.

Open problem 17 *Study the language of primitive partial words as is done for full primitive words in [105].*

We end this section with the following remark. In [18], the authors obtained consequences of the generalizations of Fine and Wilf's periodicity result to partial words. In particular, they generalized the following combinatorial property: "For any word u over $\{a, b\}$, ua or ub is primitive." This property proves in some sense that there exist very many primitive words.

2.6.2 Borderedness

Unbordered partial words were also defined in [9]: A nonempty partial word u is unbordered if no nonempty partial words x_1, x_2, v, w exist such that $u = x_1v = wx_2$ and $x_1 \uparrow x_2$. If such nonempty words exist and x is such that $x_1 \subset x$ and $x_2 \subset x$, then we call u *bordered* and x a *border* of u. A border x of u is called *minimal* if $|x| > |y|$ implies that y is not a border of u. Note that there are two types of borders: x is an *overlapping* border if $|x| > |v|$, and a *nonoverlapping* border otherwise. The partial word $u = a\diamond\diamond ab$ is bordered with the nonoverlapping border ab and overlapping border aab, the first one being minimal, while the partial word $ab\diamond c$ is unbordered.

We call a bordered partial word u *simply bordered* if a minimal border x exists satisfying $|u| \geq 2|x|$.

Proposition 2. *[21] Let u be a nonempty bordered partial word. Let x be a minimal border of u, and set $u = x_1v = wx_2$ where $x_1 \subset x$ and $x_2 \subset x$. Then the following hold:*

1. *The partial word x is unbordered.*
2. *If x_1 is unbordered, then $u = x_1u'x_2 \subset xu'x$ for some u'.*

Note that Proposition 2 implies that if u is a full bordered word, then $x_1 = x$ is unbordered. In this case, $u = xu'x$ where x is the minimal border of u. Hence a bordered full word is always simply bordered.

Corollary 5. *[21] Every bordered full word of length n has a unique minimal border x. Moreover, x is unbordered and $|x| \leq \lfloor \frac{n}{2} \rfloor$.*

In [20], the problem of enumerating all unbordered partial words with h holes of length n over a k-letter alphabet was considered, a problem that yields some open questions for further investigation. We will denote by $U_{h,k}(n)$ the number of such words.

Let us start with the problem of enumerating all unbordered full words of length n over a k-letter alphabet which gives a conceptually simple and elegant recursive formula: $U_{0,k}(0) = 1$, $U_{0,k}(1) = k$, and for $n > 0$,

$$U_{0,k}(2n) = kU_{0,k}(2n-1) - U_{0,k}(n)$$
$$U_{0,k}(2n+1) = kU_{0,k}(2n)$$

These equalities can be seen from the fact that if a word has odd length $2n+1$ then it is unbordered if and only if it is unbordered after removing the middle letter. If a word has even length $2n$ then it is unbordered if and only if it is obtained from an unbordered word of length $2n-1$ by adding a letter next to the middle position unless doing so creates a word that is a perfect square.

Using these formulas and Proposition 2, we can easily obtain a formula for counting bordered full words. Let $B_k(j,n)$ be the number of full words of length n over a k-letter alphabet that have a minimal border of length j:

$$B_k(j,n) = U_{0,k}(j)k^{n-2j}$$

If we let $B_k(n)$ be the number of full words of length n over a k-letter alphabet with a border of any length, then we have that

$$B_k(n) = \sum_{j=1}^{\lfloor \frac{n}{2} \rfloor} B_k(j,n)$$

When we allow words to have holes, counting bordered partial words is made extremely more difficult by the failure of Corollary 5 since there is now the possibility of a minimal border that is overlapping as in $a\diamond bb$. We will first concern ourselves with the simply bordered partial words. Note that because borderedness in partial words is defined via containment, it no longer makes sense to talk about *the* minimal border of a partial word, there could be many possible borders of a certain length.

To see inside the structure of the partial words we are trying to count we first define a function. Let $f_{h,k}(i,j,n)$ be the number of partial words of length n with $h > 0$ holes over a k-letter alphabet that have a hole in position i and a minimal border of length j. When $i = 0$:

$$f_{h,k}(0,j,n) = \begin{cases} \binom{n-1}{h-1}k^{n-h} & \text{if } j = 1 \\ 0 & \text{if } j > 1 \end{cases}$$

It is clear that $f_{h,k}(i,j,n)$ has some symmetry, namely that, $f_{h,k}(i,j,n) = f_{h,k}(n-1-i,j,n)$. Throughout this section we will rely on this to consider only $i \leq \lfloor \frac{n}{2} \rfloor$.

We have some general formulas for the evaluation of $f_{h,k}(i,j,n)$.

Proposition 3. *[20]*
If $0 < i < j-1$ *and* $j < \frac{n}{2}$*, then*

$$f_{h,k}(i,j,n) = \sum_{h'=1}^{\min(h,2j)} f_{h',k}(i,j,2j)\binom{n-2j}{h-h'}k^{n-2j-h+h'}$$

It is possible to see from the formula in Proposition 3 that we need only really concern ourselves with the case when $j = \lfloor \frac{n}{2} \rfloor$.

There is a similar simplification that can be made if $j-1 < i$.

Proposition 4. *[20]*
If $j-1<i$, then

$$f_{h,k}(i,j,n) = 2\sum_{i'=0}^{j-1}\sum_{h'=0}^{\min(h-1,2j)} f_{h',k}(i',j,2j)\binom{n-2j-1}{h-1-h'}k^{n-2j-h+h'}$$

If we restrict our attention to the case when $h=1$, then we can present many explicit formulas for the values $f_{1,k}(i,j,n)$. The exceptional case when $i=0$ is easily dispensed with:

$$f_{1,k}(0,j,n) = \begin{cases} k^{n-1} & \text{if } j=1 \\ 0 & \text{if } j>1 \end{cases}$$

Note that in the case where $0<i<j-1$ and $j<\frac{n}{2}$, the formula in Proposition 3 reduces to the very simple equality

$$f_{1,k}(i,j,n) = f_{1,k}(i,j,2j)k^{n-2j}$$

Similarly, in the case where $j-1<i$, the formula in Proposition 4 reduces to

$$f_{1,k}(i,j,n) = U_{0,k}(j)k^{n-2j-1}$$

By the above discussion we can restrict our attention to the cases when $i>0$, $n=2m$ and $j=m$. These are partial words with a border that takes up exactly half the length of the word. We wish to find a complete formula for $f_{1,k}(i,m,2m)$ where $i=m-1-i'$.

We proceed by induction on i'. When $i'=0$, we have the following.

Lemma 2. *[20] For all $m\geq 2$, $f_{1,k}(m-1,m,2m) = U_{0,k}(m)$.*

Continuing with the first interesting case $i'=1$, we have the following lemma.

Lemma 3. *[20]*
For all $m\geq 3$, $f_{1,k}(m-2,m,2m) = U_{0,k}(m) - k(k-1)$.

This kind of analysis quickly becomes much more complicated though. The evaluation breaks up into cases depending on how the periodicity of the words interacts with the length of the border in modular arithmetic.

Lemma 4. *[20] For all $m\geq 4$, the following holds:*

$$f_{1,k}(m-3,m,2m) = \begin{cases} U_{0,k}(m) - k^2(k-1) - k(k-1) & \textit{if } m\equiv 1 \bmod 2 \\ U_{0,k}(m) - k(k-1)^2 - k(k-1) & \textit{if } m\equiv 0 \bmod 2 \end{cases}$$

Lemma 5. *[20] For all $m\geq 5$, the following holds:*

$$f_{1,k}(m-4,m,2m) = U_{0,k}(m) - k(k-1) - g_1(m) - g_2(m)$$

where

$$g_1(m) = \begin{cases} k(k-1)^2 & \textit{if } m \equiv 0 \bmod 2 \\ 0 & \textit{if } m \equiv 1 \bmod 2 \end{cases}$$

and

$$g_2(m) = \begin{cases} k^2(k-1)^2 & \textit{if } m \equiv 0 \bmod 3 \\ U_{0,k}(4) & \textit{if } m \equiv 1 \bmod 3 \\ k^2(k-1)^2 & \textit{if } m \equiv 2 \bmod 3 \end{cases}$$

To give an idea of how the values for $f_{1,k}(i, m, 2m)$ behave unpredictably, here is a table of values that has been put together through a brute force count:

i	0	1	2	3	4	5	6	7
$f_{1,2}(i,2,4)$	0	2						
$f_{1,2}(i,3,6)$	0	2	4					
$f_{1,2}(i,4,8)$	0	2	4	6				
$f_{1,2}(i,5,10)$	0	6	6	10	12			
$f_{1,2}(i,6,12)$	0	10	12	16	18	20		
$f_{1,2}(i,7,14)$	0	22	26	32	34	38	40	
$f_{1,2}(i,8,16)$	0	42	52	60	66	70	72	74

Open problem 18 *Compute the values $f_{1,k}(m-i, m, 2m)$ for $m > i$.*

Let $S_{h,k}(n)$ be the number of simply bordered partial words of length n with h holes over a k-letter alphabet. Clearly if $h > n$, then $S_{h,k}(n) = 0$. Note that when $h = 0$, $S_{h,k}(n) = B_k(n)$.

Theorem 26. *[20]*
If $0 < h \leq n$, then a formula for $S_{h,k}(n)$ is given by:

$$S_{h,k}(2m+1) = S_{h-1,k}(2m) + kS_{h,k}(2m)$$

$$S_{h,k}(2m) = \frac{2\sum_{i=0}^{m-1}\sum_{j=1}^{m} f_{h,k}(i,j,2m)}{h}$$

We can check that

$$S_{1,k}(n) = \sum_{i=0}^{n-1}\sum_{j=1}^{\lfloor \frac{n}{2} \rfloor} f_{1,k}(i,j,n)$$

Let $N_{h,k}(n)$ be the number of partial words with h holes, of length n, over a k-letter alphabet that are not simply bordered. Obviously we can find the value of this function by subtracting the value of $S_{h,k}(n)$ from the total number of partial words with those parameters, but it would be of interest to find a direct formula for $N_{h,k}(n)$. If $h = 0$, then

$$N_{0,k}(n) = U_{0,k}(n)$$

since a bordered full word that is not simply bordered is an unbordered full word. It is easy to see that $N_{1,k}(0) = 0$, $N_{1,k}(1) = 1$, $N_{1,k}(2) = 0$, and for $h > 1$ that $N_{h,k}(1) = 0$ and $N_{h,k}(2) = 0$. Now, for $h > 0$, the following formula holds for odd $n = 2m + 1$:

$$N_{h,k}(2m + 1) = kN_{h,k}(2m) + N_{h-1,k}(2m)$$

Open problem 19 *What is $N_{h,k}(2m)$?*

If we simplify the problem down to the $h = 1$ case, then we can again use the values $f_{1,k}(i, j, n)$ to give a formula for $N_{1,k}(n)$:

$$N_{1,k}(2m) = kN_{1,k}(2m - 1) + 2U_{0,k}(2m - 1) - \sum_{i=1}^{m} f_{1,k}(i, m, 2m)$$

but it rests on the evaluation of the $f_{1,k}(i, j, 2j)$'s as well.

Other interesting questions include the following.

Open problem 20 *Count the number $O_{h,k}(n)$ of overlapping bordered partial words of length n with h holes over a k-letter alphabet for $h > 0$.*

Open problem 21 *Count the number $U_{h,k}(n)$ of unbordered partial words of length n with h holes over a k-letter alphabet for $h > 0$.*

Another open question is suggested by the fact that every partial word of length 5 that has more than two holes is simply bordered. The partial word $a\diamond\diamond bb$ shows that this bound on the number of holes for length 5 is tight. For length 6, every partial word with more than 2 holes is simply bordered as well.

Open problem 22 *What is the maximum number of holes $M(n)$ a partial word of length n can have and still fail to be simply bordered? Some values for small n follow.*

n	$M(n)$
5	2
6	2
7	3
8	4
9	5
10	5
11	6
12	7
13	8
14	8
15	9

We end this section by discussing another open problem related to borderedness in the context of partial words.

In 1979, Ehrenfeucht and Silberger initiated a line of research to explore the relationship between the minimal period of a word w of length n, $p(w)$, and the maximum length of its unbordered factors, $\mu(w)$ [64]. Clearly, $\mu(w) \leq p(w)$. They conjectured that if $n \geq 2\mu(w)$, then $\mu(w) = p(w)$. In [3], a counterexample was given and it was conjectured that $3\mu(w)$ is the precise bound. In 1982, it was established that if $n \geq 4\mu(w) - 6$, then $\mu(w) = p(w)$ [61]. In 2003, the bound was improved to $3\mu(w) - 2$ in [76] where it is believed that the precise bound can be achieved with methods similar to those presented in that paper.

Open problem 23 *Investigate the relationship between the minimal weak period of a partial word and the maximum length of its unbordered factors.*

2.7 Equations on Partial Words

As was seen in Section 2.2, some of the most basic properties of words, like the commutativity and the conjugacy properties, can be expressed as solutions of the word equations $xy = yx$ and $xz = zy$ respectively. It is also well known that the equation $x^m = y^n z^p$ has only periodic solutions in a free semigroup, that is, if $x^m = y^n z^p$ holds with integers $m, n, p \geq 2$, then there exists a word w such that x, y, z are powers of w. This result, which received a lot of attention, was first proved by Lyndon and Schützenberger for free groups [96]. Their proof implied the case for free semigroups since every free semigroup can be embedded in a free group. Direct proofs for free semigroups appear in [46, 77, 92].

In this section, we study *equations on partial words*. When we speak about them, we replace the notion of equality with compatibility. But compatibility is not transitive! We already solved the commutativity equation $xy \uparrow yx$ as well as the conjugacy equation $xz \uparrow zy$ in Section 2.2. As an application of the commutativity equation, we mention the linear time algorithm for testing primitivity on partial words that was discussed in Section 2.6 [11], and as an application of the conjugacy equation, we mention the efficient algorithm for computing a critical factorization when one exists that was discussed in Section 2.4 [22, 35]. Here, we solve three equations: $x^m \uparrow y^n$, $x^2 \uparrow y^m z$, and $x^m \uparrow y^n z^p$.

First, let us consider the equation $x^m \uparrow y^n$, also called the "good pairs" equation. If x and y are full words, then $x^m = y^n$ for some positive integers m, n if and only if there exists a word z such that $x = z^k$ and $y = z^l$ for some integers k, l. When dealing with partial words x and y, if there exists a partial word z such that $x \subset z^k$ and $y \subset z^l$ for some integers k, l, then $x^m \uparrow y^n$ for some positive integers m, n.

For the converse, we need a couple of lemmas.

Lemma 6. *[13]*

Let x, y be partial words and let m, n be positive integers such that $x^m \uparrow y^n$ with $\gcd(m,n) = 1$. Call $|x|/n = |y|/m = p$. If there exists an integer i such that $0 \leq i < p$ and $x_{i,p}$ is not 1-periodic, then $D(y_{i,p})$ is empty.

Lemma 7. *[13]*

Let x be a partial word, let m, p be positive integers, and let i be an integer such that $0 \leq i < p$. Then the relation

$$x^m_{i,p} = x_{i,p} x_{(i-|x|) \bmod p, p} \cdots x_{(i-(m-1)|x|) \bmod p, p}$$

holds.

The "good pairs" theorem is stated as follows.

Theorem 27. *[13]*

Let x, y be partial words and let m, n be positive integers such that $x^m \uparrow y^n$ with $\gcd(m,n) = 1$. Assume that (x, y) is a good pair, that is,

1. *For all $i \in H(x)$ the word ${y^n}_{i,|x|}$ is 1-periodic,*
2. *For all $i \in H(y)$ the word ${x^m}_{i,|y|}$ is 1-periodic.*

Then there exists a partial word z such that $x \subset z^k$ and $y \subset z^l$ for some integers k, l.

The assumption of (x, y) being a good pair is necessary in the "good pairs" theorem. Indeed, $x^2 = (a\diamond b)^2 \uparrow (acbadb)^1 = y^1$ but $y(1)y(4) = cd$ is not 1-periodic, and there exists no partial word z as desired.

Corollary 6. *[13]*

Let x and y be primitive partial words such that (x, y) is a good pair. If $x^m \uparrow y^n$ for some positive integers m and n, then $x \uparrow y$.

Note that if both x and y are full words, then (x, y) is a good pair. The corollary hence implies that if x, y are primitive full words satisfying $x^m = y^n$ for some positive integers m and n, then $x = y$.

Second, we consider the "good triples" equation $x^2 \uparrow y^m z$. Here, it is assumed that m is a positive integer and z is a prefix of y.

Nontrivial solutions exist! A solution is trivial if x, y, z are contained in powers of a common word. The equation $x^2 \uparrow y^m z$ has nontrivial solutions. For instance, $(a\diamond\diamond a)^2 \uparrow (aab)^2 aa$ where $x = a\diamond\diamond a$, $y = aab$, and $z = aa$.

The "good triples" theorem follows.

Theorem 28. *[13]*

Let x, y, z be partial words such that z is a proper prefix of y. Then $x^2 \uparrow y^m z$ for some positive integer m if and only if there exist partial words

$$u, v, u_0, v_0, \ldots, u_{m-1}, v_{m-1}, z_x$$

such that $u \neq \varepsilon$, $v \neq \varepsilon$, $y = uv$,

$$x = (u_0 v_0) \dots (u_{n-1} v_{n-1}) u_n \tag{2.1}$$
$$= v_n (u_{n+1} v_{n+1}) \dots (u_{m-1} v_{m-1}) z_x \tag{2.2}$$

where $0 \leq n < m, u \uparrow u_i$ and $v \uparrow v_i$ for all $0 \leq i < m$, $z \uparrow z_x$, and where one of the following holds:

1. *$m = 2n$, $|u| < |v|$, and there exist partial words u', u'_n such that $z_x = u' u_n$, $z = u u'_n$, $u \uparrow u'$ and $u_n \uparrow u'_n$.*
2. *$m = 2n + 1$, $|u| > |v|$, and there exist partial words v'_{2n} and z'_x such that $u_n = v_{2n} z_x$, $u = v'_{2n} z'_x$, $v_{2n} \uparrow v'_{2n}$ and $z_x \uparrow z'_x$.*

A triple of partial words (x, y, z) which satisfy these properties we will refer to as a good triple.

Two corollaries can be deduced.

Corollary 7. *[13]*
Let x, y, z be partial words such that z is a prefix of y. Assume that x, y are primitive and that $x^2 \uparrow y^m z$ for some integer $m \geq 2$. If x has at most one hole and y is full, then $x \uparrow y$.

Corollary 8. *[13]*
Let x, y, z be words such that z is a prefix of y. If x, y are primitive and $x^2 = y^m z$ for some integer $m \geq 2$, then $x = y$.

Note that the corollaries do not hold when $m = 1$. Indeed, the words $x = aba$, $y = abaab$ and $z = a$ provide a counterexample. Also, the first corollary does not hold when x is full and y has one hole as is seen by setting $x = abaabb$, $y = ab\diamond$ and $z = \varepsilon$.

Third, let us consider the equation $x^m y^n \uparrow z^p$. The case of full words is well known.

Theorem 29. *[96]*
Let x, y, z be full words and let m, n, p be integers such that $m \geq 2, n \geq 2$ and $p \geq 2$. Then the equation $x^m y^n = z^p$ has only trivial solutions, that is, x, y, and z are each a power of a common element.

When we deal with partial words, the equation $x^m y^n \uparrow z^p$ certainly has a solution when x, y, and z are contained in powers of a common word (we call such solutions the trivial solutions). However, there may be nontrivial solutions as is seen with the compatibility relation

$$(a\diamond b)^2 (b\diamond a)^2 \uparrow (abba)^3$$

We will classify solutions as Type 1 (or trivial) solutions when there exists a partial word w such that x, y, z are contained in powers of w, and as Type 2 solutions when the partial words x, y, z satisfy $x \uparrow z$ and $y \uparrow z$. Note that if z is full, then Type 2 solutions are trivial solutions.

The case $p \geq 4$ is stated in the following theorem.

Theorem 30. *[13] Let x, y, z be primitive partial words such that (x, z) and (y, z) are good pairs. Let m, n, p be integers such that $m \geq 2, n \geq 2$ and $p \geq 4$. Then the equation $x^m y^n \uparrow z^p$ has only solutions of Type 1 or Type 2 unless one of the following holds:*

1. *$x^2 \uparrow z^k z_p$ for some integer $k \geq 2$ and nonempty prefix z_p of z,*
2. *$z^2 \uparrow x^l x_p$ for some integer $l \geq 2$ and nonempty prefix x_p of x.*

Open problem 24 *Solve the equation $x^m y^n \uparrow z^p$ on partial words for integers $m \geq 2, n \geq 2$ and $p \in \{2, 3\}$.*

2.8 Unavoidable Sets of Partial Words

A set of (full) words X over a finite alphabet A is *unavoidable* if no two-sided infinite word over A avoids X, that is, X is unavoidable if every two-sided infinite word over A has a factor in X. For instance, the set $X = \{a, bbb\}$ is unavoidable (if a two-sided infinite word w does not have a as a factor, then w consists only of b's). This concept was explicitly introduced in 1983 in connection with an attempt to characterize the rational languages among the context-free ones [63]. It is clear from the definition that from each unavoidable set we can extract a finite unavoidable subset, so the study can be reduced to finite unavoidable sets. There is a vast literature on unavoidable sets of words and we refer the reader to [44, 93, 109, 110] for more information.

Unavoidable sets of partial words were introduced recently in [15], where the problem of classifying such sets of small cardinality was initiated, in particular, those with two elements. The authors showed that this problem reduces to the one of classifying unavoidable sets of the form

$$\{a\diamond^{m_1}a \dots a\diamond^{m_k}a, b\diamond^{n_1}b \dots b\diamond^{n_l}b\}$$

where $m_1, \dots, m_k, n_1, \dots, n_l$ are nonnegative integers and a, b are distinct letters. They gave an elegant characterization of the special case of this problem when $k = 1$ and $l = 1$. They proposed a conjecture characterizing the case where $k = 1$ and $l = 2$ and proved one direction of the conjecture. They then gave partial results towards the other direction and in particular proved that the conjecture is easy to verify in a large number of cases. Finally, they proved that verifying this conjecture is sufficient for solving the problem for larger values of k and l. In [27], the authors built on the previous work by examining, in particular, unavoidable sets of size three.

In [15], the question was raised as to whether there is an efficient algorithm to determine if a finite set of partial words is unavoidable. In [26], it was shown that this problem is NP-hard by using techniques similar to those used in a recent paper on the complexity of computing the capacity of codes that avoid forbidden difference patterns [37]. This is in contrast with the well known feasibility results for unavoidability of a set of full words [93].

The contents of Section 2.8 is as follows: In Section 2.8.1, we review basics on unavoidable sets of partial words. In Section 2.8.2, we discuss classifying such sets of size two. And in Section2.8.3, we discuss testing unavoidability of sets of partial words.

2.8.1 Unavoidable Sets

We first define some basic terminology. A two-sided infinite word over A is a total function $w : \mathbb{Z} \to A$. A finite word u is a factor of w if there exists some $i \in \mathbb{Z}$ such that $u = w(i)w(i+1)\dots w(i+|u|-1)$. A period of w is a positive integer p such that $w(i) = w(i+p)$ for all $i \in \mathbb{Z}$. If w has period p for some p, then we call w periodic. If v is a finite word, then $v^{\mathbb{Z}}$ denotes the two-sided infinite word w with period $|v|$ satisfying $w(0)\dots w(|v|-1) = v$. If X is a set of partial words, then $\hat{X}$ denotes the set of all full words compatible with a member of X. For instance, if $X = \{a\diamond\diamond a, b\diamond b\}$, then $\hat{X} = \{aaaa, aaba, abaa, abba, bab, bbb\}$.

The concept of an unavoidable set of full words is defined as follows.

Definition 8. *Let $X \subset A^*$.*

1. *A two-sided infinite word w avoids X if no factor of w is a member of X.*
2. *The set X is unavoidable if no two-sided infinite word over A avoids X, that is, X is unavoidable if every two-sided infinite word over A has a factor in X.*

If $A = \{a, b\}$, then the following sets are unavoidable: $X_1 = \{\varepsilon\}$ (ε is a factor of every two-sided infinite word); $X_2 = \{a, bbb\}$; $X_3 = \{aa, ab, ba, bb\}$ (this is the set of all words of length 2); and for any $n \in \mathbb{N}$, A^n is unavoidable.

If $X \subset A^*$ is finite, then the following three statements are equivalent: (1) X is unavoidable; (2) There are only finitely many words in A^* with no member of X as a factor; and (3) No periodic two-sided infinite word avoids X.

An unavoidable set of partial words is defined as follows.

Definition 9. *Let $X \subset A_\diamond^*$.*

1. *A two-sided infinite word w avoids X if no factor of w is a member of $\hat{X}$.*
2. *The set X is unavoidable if no two-sided infinite word over A avoids X, that is, X is unavoidable if every two-sided infinite word over A has a factor in $\hat{X}$.*

If $A = \{a, b\}$, then the following sets are unavoidable: $X_1 = \{a\diamond, \diamond b\}$; $X_2 = \{\diamond^n\}$ for any nonnegative integer n as well as any set containing X_2 as a subset (let us call such sets the *trivial* unavoidable sets); and $X_3 = \{a, bbb\}$ since of course Definition 9 is equivalent to Definition 8 if every member of X is full. We will explore some less trivial examples soon.

By the definition of $\hat{X}$, a two-sided infinite word w has a factor in $\hat{X}$ if and only if that factor is compatible with a member of X. Thus the two-sided infinite words which avoid $X \subset A_\diamond^*$ are exactly those which avoid $\hat{X} \subset A^*$, and $X \subset A_\diamond^*$ is unavoidable if and only if $\hat{X} \subset A^*$ is unavoidable. Thus with regards to unavoidability, a set of partial words serves as a representation of a set of full words. The set $\{a\diamond\diamond a, b\diamond b\}$ represents

$$\{aaaa, aaba, abaa, abba, bab, bbb\}$$

We will shortly prove that this set is unavoidable.

The smaller X is, the more information is gained by identifying X as unavoidable. Thus it is natural to begin investigating the unavoidable sets of partial words of small cardinality. Of course, every two-sided infinite word avoids the empty set and thus, there are no unavoidable sets of size 0. Unless the alphabet is unary, the only unavoidable sets of size 1 are trivial. If the alphabet is unary, then every nonempty set is unavoidable and in that case there is only one two-sided infinite word. Thus the unary alphabet is not interesting so we will not consider it further. Classifying the unavoidable sets of size 2 is the focus of the next section.

2.8.2 Classifying Unavoidable Sets of Size Two

If X is a two-element unavoidable set, then every two-sided infinite unary word has a factor compatible with a member of X. In particular, X cannot have fewer elements than the alphabet. Thus if X has size 2, then the alphabet is unary or binary. We hence assume that the alphabet is binary say with distinct letters a and b since we said above that the unary alphabet is not interesting. So one element of X is compatible with a factor of $a^{\mathbb{Z}}$ and the other element is compatible with a factor of $b^{\mathbb{Z}}$, since this is the only way to guarantee that both $a^{\mathbb{Z}}$ and $b^{\mathbb{Z}}$ will not avoid X. Thus we can restrict our attention to sets of the form

$$X_{m_1,\ldots,m_k|n_1,\ldots,n_l} = \{a\diamond^{m_1}a\ldots a\diamond^{m_k}a, b\diamond^{n_1}b\ldots b\diamond^{n_l}b\} \tag{2.3}$$

for some nonnegative integers $m_1, \ldots, m_k$ and $n_1, \ldots, n_l$. For which integers $m_1, \ldots, m_k, n_1, \ldots, n_l$ is $X_{m_1,\ldots,m_k|n_1,\ldots,n_l}$ unavoidable?

A simplification is stated in the next lemma.

Lemma 8. *[15] If p is a nonnegative integer, then set*

$$X = X_{m_1,\ldots,m_k|n_1,\ldots,n_l}$$

and

$$Y = X_{p(m_1+1)-1,\ldots,p(m_k+1)-1|p(n_1+1)-1,\ldots,p(n_l+1)-1}$$

Then X is unavoidable if and only if Y is unavoidable.

The easiest place to start is with small values of k and l. Of course, the set $\{a, b\diamond^{n_1} b \dots b\diamond^{n_l} b\}$ is always unavoidable for if w is a two-sided infinite word which does not have a as a factor, then $w = b^{\mathbb{Z}}$. This handles the case where $k = 0$ (and symmetrically $l = 0$).

An elegant characterization for the case where $k = l = 1$ is stated in the following theorem.

Theorem 31. *[15] The set $X_{m|n} = \{a\diamond^m a, b\diamond^n b\}$ is avoidable if and only if $m+1$ and $n+1$ have the same greatest power of 2 dividing them.*

The next natural step is to look at $k = 1$ and $l = 2$, that is, sets of the form

$$X_{m|n_1,n_2} = \{a\diamond^m a, b\diamond^{n_1} b\diamond^{n_2} b\}$$

On the one hand, we have identified a large number of avoidable sets of the form $\{a\diamond^m a, b\diamond^n b\}$. For $X_{m|n_1,n_2}$ to be avoidable it is sufficient that $\{a\diamond^m a, b\diamond^{n_1} b\}$, $\{a\diamond^m a, b\diamond^{n_2} b\}$ or $\{a\diamond^m a, b\diamond^{n_1+n_2+1} b\}$ be avoidable. On the other hand, the structure of words avoiding $\{a\diamond^m a, b\diamond^{n_1} b\diamond^{n_2} b\}$ is not nearly as nice as those avoiding $\{a\diamond^m a, b\diamond^n b\}$. Thus a simple characterization seems unlikely, unless perhaps there are no unavoidable sets of this form at all. But there are! The set

$$\{a\diamond^7 a, b\diamond b\diamond^3 b\}$$

is unavoidable. Seeing that it is provides a nice example of the techniques that can be used. Referring to the figure below,

...	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	...
...							b		b					...
...													a	...
...					b									...
...											a			...
...									a					...
...	b													...
...							a							...

suppose instead that there exists a two-sided infinite word w which avoids it. We know from Theorem 31 that $\{a\diamond^7 a, b\diamond b\}$ is unavoidable, thus w must have a factor compatible with $b\diamond b$. Say without loss of generality that $w(0) = w(2) = b$. This implies that $w(6) = a$, which in turn implies that $w(-2) = b$. Then we have that $w(-2) = w(0) = b$, forcing $w(4) = a$. This propagation continues: $w(-4) = w(-2) = b$ and so $w(2) = a$, which makes $w(-6) = b$ giving $w(0) = a$, a contradiction.

The perpetuating patterns phenomenon of the previous example is a special case of a more general result.

Theorem 32. *[15] If $m = n_2 - n_1 - 1$ or $m = 2n_1 + n_2 + 2$, and the highest power of 2 dividing $n_1 + 1$ is less than the highest power of 2 dividing $m + 1$, then $X_{m|n_1,n_2}$ is unavoidable.*

Here are other unavoidability results for $k = 1$ and $l = 2$.

Proposition 5. *[15] The set $X_{m|n_1,n_2}$ is unavoidable if Conditions 1 or Conditions 2 or Conditions 3 hold:*

1. *$\{a\diamond^m a, b\diamond^{n_1} b\}$ is unavoidable, $m = 2n_1 + n_2 + 2$ or $m = n_2 - n_1 - 1$, and $n_1 + 1$ divides $n_2 + 1$.*
2. *$n_1 < n_2$, $2m = n_1 + n_2$ and $m - n_1$ divides $m + 1$.*
3. *$m = 6$, $n_1 = 1$ and $n_2 = 3$.*

Extensive experimentation suggests that these results (and their symmetric equivalents) give a complete characterization of when $X_{m|n_1,n_2}$ is unavoidable.

Conjecture 1 *[15] The set $X_{m|n_1,n_2}$ is unavoidable if and only if one of the following conditions (or symmetric equivalents) holds:*

1. *$\{a\diamond^m a, b\diamond^{n_1} b\}$ is unavoidable, $m = 2n_1 + n_2 + 2$ or $m = n_2 - n_1 - 1$, and $n_1 + 1$ divides $n_2 + 1$.*
2. *$m = n_2 - n_1 - 1$ or $m = 2n_1 + n_2 + 2$, and the highest power of 2 dividing $n_1 + 1$ is less than the highest power of 2 dividing $m + 1$.*
3. *$n_1 < n_2$, $2m = n_1 + n_2$ and $m - n_1$ divides $m + 1$.*
4. *$m = 6$, $n_1 = 1$ and $n_2 = 3$.*

Open problem 25 *Is Conjecture 1 true or false?*

If true, then Conjecture 1 implies that the unavoidable sets of size two have been completely classified as stated in the following proposition.

Proposition 6. *[15] If Conjecture 1 is true, then $X_{m_1,\ldots,m_k|n_1,\ldots,n_l}$ is avoidable for $k = 1$ and $l \geq 3$, and for $k > 1$ and $l \geq 2$.*

In order to prove the conjecture, only one direction remains. We must show that if none of the aforementioned conditions hold, then $X_{m|n_1,n_2}$ is avoidable. There are some partial results towards this goal. In particular there is an easy way of verifying the conjecture for even values of m.

Proposition 7. *[15] Assume m is even and $2m \leq \min(n_1, n_2)$. Then $X_{m|n_1,n_2}$ is avoidable.*

Thus for any fixed even m we only need to verify the conjecture for finitely many values of n_1 and n_2, which is generally easy. For

1. $m = 0$: $X_{0|n_1,n_2}$ is always avoidable, and indeed this is the case.
2. $m = 2$: $X_{2|n_1,n_2}$ is avoidable except for $n_1 = 1, n_2 = 3$ or $n_1 = 3, n_2 = 1$. It is easy to find avoiding two-sided infinite words for other values of n_1 and n_2 less than 5 when $m = 2$. This is all that is necessary to confirm the conjecture for $m = 2$.

In this way the conjecture has been verified for all even m up to very large values via computer.

The odd values of m seem to be much more difficult. The following proposition shows that the conjecture is true for $m = 1$.

Proposition 8. *[15] Conjecture 1 is true for $m = 1$, that is, $X_{1|n_1,n_2}$ is unavoidable if and only if n_1 and n_2 are even numbers with $|n_1 - n_2| = 2$.*

Other results on the avoidability of $X_{m|n_1,n_2}$ include the following.

Proposition 9. *[15]*

1. *Let $s \in \mathbb{N}$ with $s < m - 2$. Then for $n > 2(m+1)^2 + m - 1$, $X_{m|m+s,n} = \{a\diamond^m a, b\diamond^{m+s} b\diamond^n b\}$ is avoidable. Intuitively this means that if m and n_1 are relatively close in value, then the set of integers n_2 which make $X_{m|n_1,n_2}$ unavoidable is finite.*
2. *If $\max(n_1, n_2) < m < n_1 + n_2 + 2$, then $X_{m|n_1,n_2}$ is avoidable.*
3. *The set $X = \{a\diamond^m a, bbb\}$ is avoidable.*

Classifying the unavoidable sets of partial words of size greater than or equal to two remains an open question.

Open problem 26 *Classify the unavoidable sets of partial words of size $l \geq 2$ over a k-letter alphabet where $k \leq l$.*

2.8.3 Testing Unavoidability

Efficient algorithms to determine if a finite set of full words is unavoidable are well known [45, 93]. For example, we can check whether there is a loop in the finite automaton of Aho and Corasick [1] recognizing $A^* \setminus A^*XA^*$. Another approach is the following. We say that a set of words Y is obtained from a finite set of words X by an elementary derivation if

1. *Type 1 elementary derivation:* There exist words $x, y \in X$ such that x is a proper prefix of y, and $Y = X \setminus \{y\}$ (this will be denoted by $X \xrightarrow{1} Y$).
2. *Type 2 elementary derivation:* There exists a word $x = ya \in X$ with $a \in A$ such that, for each letter $b \in A$ there is a suffix z of y such that $zb \in X$, and $Y = (X \setminus \{x\}) \cup \{y\}$ (this will be denoted by $X \xrightarrow{2} Y$).

A *derivation* is a sequence of elementary derivations. We say that Y is derived from X if Y is obtained from X by a derivation. If Y is derived from X, then X is unavoidable if and only if Y is unavoidable.

Example 2. The following sequence of elementary derivations shows that $X = \{aaaa, aaba, abaa, abba, bab, bbb\}$ derives $\{\varepsilon\}$:

$$
\begin{aligned}
X &\xrightarrow{2} \{aaaa, aaba, aba, abba, bab, bbb\}\\
&\xrightarrow{2} \{aaaa, aaba, aba, abb, bab, bbb\}\\
&\xrightarrow{2} \{aaaa, aaba, ab, bab, bbb\}\\
&\xrightarrow{2} \{aaa, aaba, ab, bab, bbb\}\\
&\xrightarrow{2} \{aa, aaba, ab, bab, bbb\}\\
&\xrightarrow{1} \{aa, ab, bab, bbb\}\\
&\xrightarrow{2} \{a, ab, bab, bbb\}\\
&\xrightarrow{1} \{a, bab, bbb\}\\
&\xrightarrow{2} \{a, ba, bbb\}\\
&\xrightarrow{2} \{a, ba, bb\}\\
&\xrightarrow{2} \{a, b, bb\}\\
&\xrightarrow{2} \{a, b\}\\
&\xrightarrow{2} \{\varepsilon, b\}\\
&\xrightarrow{1} \{\varepsilon\}
\end{aligned}
$$

The notion of a derivation gives an algorithm to check whether a set is unavoidable: A finite set X is unavoidable if and only if there is a derivation from X to the set $\{\varepsilon\}$. The above derivation shows that $\{aaaa, aaba, abaa, abba, bab, bbb\}$ is unavoidable.

These algorithms to determine if a finite set of full words is unavoidable, like the one just described, can be used to decide if a finite set of partial words X is unavoidable by determining the unavoidability of $\hat{X}$. However this incurs a dramatic loss in efficiency, as each partial word u in X can contribute as many as $\|A\|^{\|H(u)\|}$ elements to $\hat{X}$. The above derivation shows that $\{a\diamond\diamond a, b\diamond b\}$ is unavoidable as is confirmed by Theorem 31 since $m+1 = 2+1 = 3 = 2^0 3$ and $n+1 = 1+1 = 2 = 2^1$.

In [15], the question was raised as to whether there is an efficient algorithm to determine if a finite set of partial words is unavoidable. In [26], it was proved that testing the unavoidability of a finite set of partial words is much harder to handle than the similar problem for full words. Indeed, the following theorem holds (note that the case $k = 1$ is trivial).

Theorem 33. *[26] The problem of deciding whether a finite set of partial words over a k-letter alphabet where $k \geq 2$ is unavoidable is NP-hard.*

The proof proceeds by reduction from the 3SAT problem that is known to be NP-complete (see [70]). In the 3SAT problem, we are given n binary variables $x_1, \ldots, x_n$ and m clauses that each contain three literals (a literal can be a variable or its negation), and we search a truth assignment for the variables such that each clause has at least one true literal.

In [26], the following related questions on avoidability of sets of partial words were raised.

Open problem 27 *Is the decision problem of the avoidability of a set of partial words in NP?*

A similar (stronger) question is the following one.

Open problem 28 *For any set of partial words X, does there always exist a two-sided infinite periodic word that avoids X, whose period is polynomial in the size of X?*

2.9 Freeness of Partial Words

In [99], Manea and Mercaş introduce freeness of partial words. There, they extend in a natural way the concepts of square- and overlap-freeness of words to partial words. In [31, 30], some more basic freeness properties of partial words are investigated generalizing the well-known freeness properties of full words.

A one-sided infinite word over the alphabet A is a function from $\mathbb{N}$ to A. The Thue-Morse word is an example of a one-sided infinite word defined by iterating a morphism. Let $\phi : \{a, b\}^* \to \{a, b\}^*$ be the morphism defined by $\phi(a) = ab$ and $\phi(b) = ba$. We define $t_0 = a$ and $t_i = \phi^i(a)$, for all $i \geq 1$. Note that $t_{i+1} = \phi(t_i)$ and that $t_{i+1} = t_i\overline{t_i}$, where $\bar{x}$ is the word obtained from x by replacing each occurrence of a with b and each occurrence of b with a. Thus, the limit (the infinite word) $t = \lim_{i\to\infty} t_i$ exists. The Thue-Morse word is defined as t, a fixed point for the morphism ϕ. Computations show that $t_1 = ab$, $t_2 = abba$, $t_3 = abbabaab$, $t_4 = abbabaabbaababba$, and

$$t_5 = abbabaabbaaba\underline{b}babaababbaabbabaab \tag{2.4}$$

and so on.

A one-sided infinite word w is *k-free* if there is no word x such that x^k is a factor of w (a word that is 2-free is also called square-free and a word that is 3-free is called cube-free). It is called *overlap-free* if it does not contain any factor of the form $cycyc$ with $c \in A$. Any overlap-free word is clearly k-free for all $k \geq 3$.

Theorem 34. *[120, 121] The Thue-Morse infinite word t is overlap-free and hence k-free for all $k \geq 3$.*

A one-sided infinite partial word w over the alphabet A is a partial function from $\mathbb{N}$ to A. We call w *k-free* if for any nonempty factor $x_1 \ldots x_k$ of w, no partial word x exists such that $x_i \subset x$ for all $1 \leq i \leq k$. And it is said to be *overlap-free* if for any factor $c_1y_1c_2y_2c_3$ of w no letter $c \in A$ and partial word y over A exist such that $c_i \subset c$ for all $1 \leq i \leq 3$ and $y_j \subset y$ for all $1 \leq j \leq 2$. In [99], the authors propose an efficient algorithm to test whether or not a partial word of length n is k-free. Both the time and space complexities of the algorithm are $O(\frac{n}{k})$. In case of full words, the time complexity can be reduced to $O(n \log n)$ using suffix arrays [98]. In [99], the authors also give an efficient algorithm to construct in $O(n)$ time a cube-free (and hence k-free for all $k \geq 3$) partial word with n holes, and modify the algorithm in the case of a four-letter alphabet to produce such a partial word of minimal length $3n-2$ (which is the minimal length among all the possible cube-free words with n holes regardless of the alphabet over which these words are constructed).

Theorem 35. *[99] For $k \geq 3$, there exist infinitely many k-free infinite partial words over a two-letter alphabet containing an arbitrary number of holes.*

Note that it is enough to show the result for $k = 3$. The idea of the proof is to show that there exist infinitely many cube-free infinite partial words containing exactly one hole over a two-letter alphabet. In order to do this, observe that if the underlined b in Equality 2.4 is replaced by $\diamond$, then the resulting partial word is still cube-free. Since there is an infinite number of occurrences of t_5 in t, any replacement of the underlined b in such occurrences leads to an infinite partial word with one hole that is cube-free. The result follows since there is an infinite number of nonoverlapping occurrences of t_5 in t.

A surprising result holds for an alphabet of size four.

Theorem 36. *[99] There exists an infinite cube-free word over a four-letter alphabet in which we can randomly replace letters by holes and obtain in this way an infinite partial word that is cube-free as long as each pair of two consecutive holes are separated by at least two letters of the alphabet. Moreover, such a word does not exist over a three-letter alphabet.*

We discuss the concept of square-freeness of partial words in Section 2.9.1 and of overlap-freeness of partial words in Section 2.9.2.

2.9.1 Square-Freeness

Let us now consider the $k = 2$ case. A well known result from Thue states that over a three-letter alphabet there exist infinitely many infinite words that are square-free [120, 121]. To generalize Thue's result, we wish to find a square-free partial word with infinitely many holes, and an infinite full word that remains square-free even after replacing an arbitrary selection of letters with holes. Unfortunately, every partial word containing at least one hole and having length at least two contains a square (either $a\diamond$ or $\diamond a$ cannot be avoided, where a denotes a letter from the alphabet). Furthermore, it is obvious that if we replace $2n$ consecutive letters in a full word with holes, then the corresponding factor of the resulting partial word will be a square.

Motivated by these observations, we call a word *non-trivial square*-free if it contains no factors of the form $w^k, k \geq 2$, except when $|w| \in \{1, 2\}$ and $k = 2$. Notice that the cube aaa is considered to be a non-trivial square. For the sake of readability, we shall use the terms *non-trivial square* and *square* interchangeably. The study of non-trivial squares is not new. In [106], several iterating morphisms are given for infinite words avoiding non-trivial squares. In particular, the authors give an infinite binary word avoiding both cubes xxx and squares yy with $|y| \geq 4$ and an infinite binary word avoiding all squares except 0^2, 1^2, and $(01)^2$ using a construction that is somewhat simpler than the original one from Fraenkel and Simpson [68].

Remark 1. When we introduce holes into arbitrary positions of a word, we impose the restriction that every two holes must have at least two non-hole symbols between them.

With this restriction, the study of square-free partial words becomes much more subtle and interesting.

Theorem 37. *[31] There exists an infinite word over an eight-letter alphabet that remains square-free after replacing an arbitrary selection of its letters with holes, and none exists over a smaller alphabet.*

A suggested problem for investigation is the following. Let $g(n)$ be the length of a longest binary full word containing at most n distinct squares. How does the sequence $\{g(n)\}$ behave? A complete answer appears in [68].

Open problem 29 *Compute the maximum number of distinct squares in a partial word with h holes of length n over a k-letter alphabet.*

2.9.2 Overlap-Freeness

A well known result of Thue states that over a binary alphabet there exist infinitely many overlap-free words [120, 121]. In [99], the question was raised as to whether there exist overlap-free infinite partial words, and to construct them over a binary alphabet if such exist. The following result settles this question.

Theorem 38. *[31] There exist overlap-free infinite partial words with one hole over a two-letter alphabet, and none exists with more than one hole.*

The following result relates to a three-letter alphabet.

Theorem 39. *[31] There exist infinitely many overlap-free infinite partial words with an arbitrary number of holes over a three-letter alphabet.*

For the following result, we adhere to the restriction described in Remark 1 when replacing an arbitrary selection of letters in a word with holes.

Theorem 40. *[31] There exists an infinite overlap-free word over a six-letter alphabet that remains overlap-free after an arbitrary selection of its letters are changed to holes, and none exists over a four-letter alphabet.*

The case of a five-letter alphabet remains open.

Open problem 30 *Does there exist an infinite word over a five-letter alphabet that remains overlap-free after an arbitrary insertion of holes?*

Other problems are suggested in [31].

Open problem 31 *Extend the concept of square-free (respectively, overlap-free or cube-free) morphism to partial words.*

From [31, 99], some of the properties of this kind of morphisms already start to be obvious. A further analysis might give additional properties that such morphisms should fulfill. Following the approach of Dejean [56], another interesting problem to analyze is the following.

Open problem 32 *Identify the exact value of k (related to k-freeness) for a given alphabet size. This value would represent the repetitiveness threshold in an n-letter alphabet.*

If for full words this value is known for alphabets up to size 11 and it is conjectured that for bigger size alphabets the value is $\frac{n+1}{n}$, for partial words this value has not yet been investigated.

2.10 Other Open Problems

The theory of *codes* has been widely developed in connection with combinatorics on words [5]. In [7, 32], a new line of research was initiated by introducing *pcodes* in connection with combinatorics on partial words, and a theoretical framework for pcodes was developed by revisiting the theory of codes of words, as exposited in [5], starting from pcodes of partial words. Pcodes are defined in terms of the compatibility relation as follows.

Definition 10. *[7] Let X be a nonempty set of partial words over an alphabet A. Then X is called a* pcode *over A if for all positive integers m, n and partial words $u_1, \ldots, u_m, v_1, \ldots, v_n \in X$, the condition*

$$u_1u_2 \ldots u_m \uparrow v_1v_2 \ldots v_n$$

implies $m = n$ and $u_i = v_i$ for $i = 1, \ldots, m$.

An area of current interest for the study of pcodes is data communication where some information may be missing, lost, or unknown. While a code of words X does not allow two distinct decipherings of some word in X^+, a pcode of partial words Y does not allow two distinct compatible decipherings in Y^+. Various ways have been described for defining and analyzing pcodes. In particular, many pcodes can be obtained as antichains with respect to certain partial orderings. Adapting a graph technique related to dominoes [6, 73, 79], the pcode property was shown to be decidable for finite sets of partial words.

For example, the set $X = \{a\diamond, a\diamond b\}$ is a pcode over $\{a,b\}$, but the set $Y = \{u_1, u_2, u_3, u_4\}$ where $u_1 = a\diamond b, u_2 = aa\diamond bba, u_3 = \diamond b$, and $u_4 = ba$ is not a pcode over $\{a,b\}$ since $u_1u_3u_3u_4u_3 \uparrow u_2u_3u_1$ is a nontrivial compatibility relation over Y.

It is well known that the two-element set of words $\{u, v\}$ is a code if and only if $uv \neq vu$. However, this is not true in general for partial words. For instance, the set $\{u, v\}$ where $u = a\diamond b$ and $v = abbaab$ satisfies $uv \not\uparrow vu$, but $\{u, v\}$ is not a pcode since $u^2 \uparrow v$.

Open problem 33 *Find a necessary and sufficient condition for a two-element set of partial words to be a pcode.*

Other suggested problems are the following.

Open problem 34 *Investigate the concept of* tiling periodicity *introduced recently by Karhumäki, Lifshits and Rytter's [81]. There, the authors suggest a number of questions for further work on this new concept.*

Punctured languages are sets whose elements are partial words. In [91], Lischke investigated to which extent restoration of punctured languages is possible if the number of holes or the proportion of holes per word, respectively, is bounded, and studied their relationships for different boundings. The considered restoration classes coincide with similarity classes according to some kind of similarity for languages. Thus all results he can also formulate in the language of similarity. He shows some hierarchies of similarity classes for each class of the Chomsky hierarchy, and proves the existence of linear languages which are not δ-similar to any regular language for any $\delta < \frac{1}{2}$.

Open problem 35 *For $\frac{1}{2} \leq \delta$, do there exist linear languages which are not δ-similar to any regular language? If they exist, then they must be* non-slender.

References

1. A.V. Aho and M.J. Corasick. Efficient string machines, an aid to bibliographic research. *Comm. ACM*, 18:333–340, 1975.
2. J.P. Allouche and J. Shallit. *Automatic Sequences: Theory, Applications, Generalizations.* Cambridge University Press, 2003.
3. R. Assous and M. Pouzet. Une caractérisation des mots périodiques. *Discrete Math.*, 25:1–5, 1979.
4. J. Berstel and L. Boasson. Partial words and a theorem of fine and wilf. *Theoret. Comput. Sci.*, 218:135–141, 1999.
5. J. Berstel and D. Perrin. *Theory of Codes.* Academic Press, Orlando, FL, 1985.
6. F. Blanchet-Sadri. On unique, multiset, and set decipherability of three-word codes. *IEEE Trans. Inform. Theory*, 47:1745–1757, 2001.
7. F. Blanchet-Sadri. Codes, orderings, and partial words. *Theoret. Comput. Sci.*, 329:177–202, 2004.

8. F. Blanchet-Sadri. Periodicity on partial words. *Comput. Math. Appl.*, 47:71–82, 2004.
9. F. Blanchet-Sadri. Primitive partial words. *Discrete Appl. Math.*, 148:195–213, 2005.
10. F. Blanchet-Sadri. *Algorithmic Combinatorics on Partial Words.* Chapman & Hall/CRC Press, 2007.
11. F. Blanchet-Sadri and A.R. Anavekar. Testing primitivity on partial words. *Discrete Appl. Math.*, 155:279–287, 2007. (`www.uncg.edu/mat/primitive`).
12. F. Blanchet-Sadri, D. Bal, and G. Sisodia. Graph connectivity, partial words and a theorem of Fine amd Wilf. *Information and Computation*, to appear(`www.uncg.edu/mat/research/finewilf3`).
13. F. Blanchet-Sadri, D.D. Blair, and R.V. Lewis. Equations on partial words. In R. Královic and P. Urzyczyn, editors, *MFCS 2006. 31st International Symposium on Mathematical Foundations of Computer Science*, LNCS, vol. 4162, Springer, pp. 167-178.
14. F. Blanchet-Sadri, L. Bromberg, and K. Zipple. Tilings and quasiperiods of words. Preprint (`www.uncg.edu/cmp/research/tilingperiodicity`).
15. F. Blanchet-Sadri, N.C. Brownstein, and J. Palumbo. Two element unavoidable sets of partial words. In T. Harju, J. Karhumäki, and A. Lepistö, editors, *DLT 2007. 11th International Conference on Developments in Language Theory*, LNCS, Vol. 4588, Springer, pp. 96-107.
16. F. Blanchet-Sadri and A. Chriscoe. Local periods and binary partial words: an algorithm. *Theoret. Comput. Sci.*, 314:189–216, 2004. (`www.uncg.edu/mat/AlgBin`).
17. F. Blanchet-Sadri, E. Clader, and O. Simpson. Border correlations of partial words. Preprint (`www.uncg.edu/cmp/research/bordercorrelation`).
18. F. Blanchet-Sadri, K. Corcoran, and J. Nyberg. Fine and wilf's periodicity result on partial words and consequences. In *LATA 2007, 1st International Conference on Language and Automata Theory and Applications*, GRLMC Report 35/07, Tarragona.
19. F. Blanchet-Sadri and M. Cucuringu. Counting primitive partial words. Preprint.
20. F. Blanchet-Sadri, M. Cucuringu, and J. Dodge. Counting unbordered partial words. Preprint.
21. F. Blanchet-Sadri, C.D. Davis, J. Dodge, R. Mercaş, and M. Moorefield. Unbordered partial words. Preprint (`www.uncg.edu/mat/border`).
22. F. Blanchet-Sadri and S. Duncan. Partial words and the critical factorization theorem. *J. Combin. Theory Ser. A*, 109:221–245, 2005. (`www.uncg.edu/mat/cft`).
23. F. Blanchet-Sadri, J. Fowler, J. Gafni, and K. Wilson. Combinatorics on partial word correlations. Preprint (`www.uncg.edu/cmp/research/correlations2`).
24. F. Blanchet-Sadri, J. Gafni, and K. Wilson. Correlations of partial words. In W. Thomas and P. Weil, editors, *STACS 2007*, volume 4393, pages 97–108, Berlin, 2007. (`www.uncg.edu/mat/research/correlations`).
25. F. Blanchet-Sadri and R.A. Hegstrom. Partial words and a theorem of fine and wilf revisited. *Theoret. Comput. Sci.*, 270:401–419, 2002.
26. F. Blanchet-Sadri, R. Jungers, and J. Palumbo. Testing avoidability of sets of partial words is hard. Preprint.
27. F. Blanchet-Sadri, A. Kalcic, and T. Weyand. Unavoidable sets of partial words of size three. Preprint (`www.uncg.edu/cmp/research/unavoidablesets2`).

28. F. Blanchet-Sadri and D.K. Luhmann. Conjugacy on partial words. *Theoret. Comput. Sci.*, 289:297–312, 2002.
29. F. Blanchet-Sadri, T. Mandel, and G. Sisodia. Connectivity in graphs associated with partial words. Preprint (`www.uncg.edu/cmp/research/finewilf4`).
30. F. Blanchet-Sadri, R. Mercaş, and G. Scott. Counting distinct squares in partial words. Preprint (`www.uncg.edu/cmp/research/freeness`).
31. F. Blanchet-Sadri, R. Mercaş, and G. Scott. A generalization of thue freeness for partial words. Preprint (`www.uncg.edu/cmp/research/freeness`).
32. F. Blanchet-Sadri and M. Moorefield. Pcodes of partial words. Preprint(`www.uncg.edu/mat/pcode`).
33. F. Blanchet-Sadri, T. Oey, and T. Rankin. Partial words and generalized fine and wilf's theorem for an arbitrary number of weak periods. Preprint (`www.uncg.edu/mat/research/finewilf2`).
34. F. Blanchet-Sadri, B. Shirey, and G. Gramajo. Periods, partial words, and a result of guibas and odlyzko. Preprint (`www.uncg.edu/mat/bintwo`).
35. F. Blanchet-Sadri and N.D. Wetzler. Partial words and the critical factorization theorem revisited. *Theoret. Comput. Sci.*, to appear. (`www.uncg.edu/mat/research/cft2`).
36. F. Blanchet-Sadri and J. Zhang. On the critical factorization theorem. Preprint.
37. V.D. Blondel, R. Jungers, and V. Protasov. On the complexity of computing the capacity of codes that avoid forbidden difference patterns. *IEEE Trans. Inform. Theory*, 52:5122–5127, 2006.
38. R.S. Boyer and J.S Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.
39. D. Breslauer, T. Jiang, and Z. Jiang. Rotations of periodic strings and short superstrings. *J. of Algorithms*, 24:340–353, 1997.
40. J. Buhler, U. Keich, and Y. Sun. Designing seeds for similarity search in genomic dna. *J. Comput. System Sci.*, 70:342–363, 2005.
41. P. Bylanski and D.G.W. Ingram. Digital transmission systems. *IEE*, 1980.
42. M.G. Castelli, F. Mignosi, and A. Restivo. Fine and wilf's theorem for three periods and a generalization of sturmian words. *Theoret. Comput. Sci.*, 218:83–94, 1999.
43. Y. Césari and M. Vincent. Une caractérisation des mots périodiques. *C.R. Acad. Sci. Paris*, 268:1175–1177, 1978.
44. C. Choffrut and K. Culik II. On extendibility of unavoidable sets. *Discrete Appl. Math.*, 9:125–137, 1984.
45. C. Choffrut and J. Karhumäki. Combinatorics of words. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, pages 329–438. Springer, Berlin, 1997.
46. D.D. Chu and H.S. Town. Another proof on a theorem of lyndon and schützenberger in a free monoid. *Soochow J. Math.*, 4:143–146, 1978.
47. S. Constantinescu and L. Ilie. Generalised fine and wilf's theorem for arbitrary number of periods. *Theoret. Comput. Sci.*, 339:49–60, 2005.
48. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.
49. M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Text compression using antidictionaries. *LNCS*, 1644:261–270, 1999.
50. M. Crochemore and D. Perrin. Two-way string matching. *J. of the ACM*, 38:651–675, 1991.

51. M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.
52. M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, NJ, 2003.
53. A. de Luca. On the combinatorics of finite words. *Theoret. Comput. Sci.*, 218:13–39, 1999.
54. A. de Luca and S. Varricchio. *Regularity and Finiteness Conditions*, volume 1, chapter 11, pages 747–810. Springer, Berlin, 1997.
55. A. de Luca and S. Varricchio. *Finiteness and Regularity in Semigroups and Formal Languages*. Springer, Berlin, 1999.
56. F. Dejean. Sur un théorème de thue. *J. Combin. Theory Ser. A*, 13:90–99, 1972.
57. P. Dömösi. Some results and problems on primitive words. In *11th International Conference on Automata and Formal Languages*, 2005.
58. P. Dömösi, S. Horváth, and M. Ito. *Primitive Words and Context-Free Languages*.
59. J.P. Duval. Périodes locales et propagation de périodes dans un mot. *Theoret. Comput. Sci.*, 204: 87-98, 1998
60. J.P. Duval. Périodes et répétitions des mots du monoïde libre. *Theoret. Comput. Sci.*, 9:17–26, 1979.
61. J.P. Duval. Relationship between the period of a finite word and the length of its unbordered segments. *Discrete Math.*, 40:31–44, 1982.
62. J.P. Duval, R. Kolpakov, G. Kucherov, T. Lecroq, and A. Lefebvre. Linear-time computation of local periods. *Theoret. Comput. Sci.*, 326:229–240, 2004.
63. A. Ehrenfeucht, D. Haussler, and G. Rozenberg. On regularity of context-free languages. *Theoret. Comput. Sci.*, 27:311–322, 1983.
64. A. Ehrenfeucht and D.M. Silberger. Periodicity and unbordered segments of words. *Discrete Math.*, 26:101–109, 1979.
65. P. Erdös. Note on sequences of integers no one of which is divisible by another. *J. London Math. Soc.*, 10:126–128, 1935.
66. M. Farach-Colton, G.M. Landau, S.C. Sahinalp, and D. Tsur. Optimal spaced seeds for approximate string matching. In L. Caires, G.F. Italiano, L. Monteiro, C. Palanidessi, and M. Yung, editors, *ICALP 2005*, LNCS, vol. 3580, Springer, pp. 1251-1262, 2005.
67. N.J. Fine and H.S. Wilf. Uniqueness theorems for periodic functions. In *Proc. Amer. Math. Soc.*, volume 16, pages 109–114, 1965.
68. A.S. Fraenkel and R.J. Simpson. How many squares must a binary sequence contain? *Electron. J. Combin.*, 2, 1995.
69. Z. Galil and J. Seiferas. Time-space optimal string matching. *J. Comput. System Sci.*, 26:280–294, 1983.
70. M.R. Garey and D.S. Johnson. *Computers and Intractability -A Guide to the Theory of NP-Completeness*. Freeman, 1979.
71. L.J. Guibas and A.M. Odlyzko. Periods in strings. *J. Combin. Theory Ser. A*, 30:19–42, 1981.
72. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, Cambridge, 1997.
73. F. Guzmán. Decipherability of codes. *J. Pure Appl. Algebra*, 141:13–35, 1999.
74. V. Halava, T. Harju, and L. Ilie. Periods and binary words. *J. Combin. Theory Ser. A*, 89:298–303, 2000.

75. T. Harju. *Combinatorics on Words*, chapter 19, pages 381–392. Springer, Berlin, 2006.
76. T. Harju and D. Nowotka. Periodicity and unbordered segments of words. *Bull. Eur. Assoc. Theor. Comput. Sci. EATCS*, 80:162–167, 2003.
77. T. Harju and D. Nowotka. The equation $x^i = y^j z^k$ in a free semigroup. *Semigroup Forum*, 68:488–490, 2004.
78. T. Head, G. Paun, and D. Pixton. *Language Theory and Molecular Genetics*, volume 2, chapter 7, pages 295–360. Springer, Berlin, 1997.
79. T. Head and A. Weber. Deciding multiset decipherability. *IEEE Trans. Inform. Theory*, 41:291–297, 1995.
80. J. Justin. On a paper by castelli, mignosi, restivo. *Theoret. Inform. Appl.*, 34:373–377, 2000.
81. J. Karhumäki, Y. Lifshits, and W. Rytter. Tiling periodicity. In *CPM 2007, 18th Annual Symposium on Combinatorial Pattern Matching*, 2007.
82. L. Kari, G. Rozenberg, and A. Salomaa. *L Systems*, volume 1, chapter 7, pages 253–328. Springer, Berlin, 1997.
83. U. Keich, M. Li, B. Ma, and J. Tromp. On spaced seeds for similarity search. *Discrete Appl. Math.*, 138:253–263, 2004.
84. D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. on Comput.*, 6:323–350, 1977.
85. R. Kolpakov and G. Kucherov. Finding approximate repetitions under hamming distance. *Lecture Notes in Computer Science*, 2161:170–181, 2001.
86. R. Kolpakov and G. Kucherov. Finding approximate repetitions under hamming distance. *Theoret. Comput. Sci.*, 33:135–156, 2003.
87. G. Landau and J. Schmidt. An algorithm for approximate tandem repeats. *Lecture Notes in Computer Science*, 684:120–133, 1993.
88. G.M. Landau, J.P. Schmidt, and D. Sokol. An algorithm for approximate tandem repeats. *J. Comput. Biology*, 8:1–18, 2001.
89. P. Leupold. Languages of partial words - how to obtain them and what properties they have. *Grammars*, 7:179–192, 2004.
90. P. Leupold. Partial words for dna coding. *LNCS*, 3384:224–234, 2005.
91. G. Lischke. Restorations of punctured languages and similarity of languages. *Math. Logic Quart.*, 52:20–28, 2006.
92. M. Lothaire. *Combinatorics on Words.* Cambridge University Press, Cambridge, 1997.
93. M. Lothaire. *Algebraic Combinatorics on Words.* Cambridge University Press, Cambridge, 2002.
94. M. Lothaire. *Applied Combinatorics on Words.* Cambridge University Press, Cambridge, 2005.
95. R.C. Lyndon and P.E. Schupp. *Combinatorial Group Theory.* Springer, Berlin, 2001.
96. R.C. Lyndon and M.P. Schützenberger. The equation $a^m = b^n c^p$ in a free group. *Michigan Math. J.*, 9; 289–298, 1962.
97. B. Ma, J. Tromp, and M. Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18:440–445, 2002.
98. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. on Comput.*, 22:935–948, 1993.
99. F. Manea and R. Mercas. Freeness of partial words. Preprint.

100. D. Margaritis and S. Skiena. Reconstructing strings from substrings in rounds. In *FOCS 1995, 36th Annual Symposium on Foundations of Computer Science*, pages 613–620, 1995.
101. E.M. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23:262–272, 1976.
102. F. Mignosi, A. Restivo, and S. Salemi. A periodicity theorem on words and applications. *LNCS*, 969:337–348, 1995.
103. F. Nicolas and E. Rivals. Hardness of optimal spaced seed design. In Apostolico, A., Crochemore, M. and Park, K., editors, *CPM 2005, 16th Annual Symposium on Combinatorial Pattern Matching*, LNCS, vol. 3537, Srpinger, pages 144–155, 2005.
104. L. Noé and G. Kucherov. Improved hit criteria for dna local alignment. *BMC Bioinformatics*, 5, 2004.
105. H. Petersen. On the language of primitive words. *Theoret. Comput. Sci.*, 161:141–156, 1996.
106. N. Rampersad, J. Shallit, and M. w Wang. Avoiding large squares in infinite binary words. *Theoret. Comput. Sci.*, 339:19–34, 2005.
107. G. Richomme. Sudo-lyndon. *Bull. Eur. Assoc. Theor. Comput. Sci. EATCS*, 92:143–149, 2007.
108. E. Rivals and S. Rahmann. Combinatorics of periods in strings. *J. Combin. Theory Ser. A*, 104:95–113, 2003.
109. L. Rosaz. Unavoidable languages, cuts and innocent sets of words. *RAIRO Theoret. Inform. Appl.*, 29:339–382, 1995.
110. L. Rosaz. Inventories of unavoidable languages and the word-extension conjecture. *Theoret. Comput. Sci.*, 201:151–170, 1998.
111. J.P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM J. Comput.*, 27:972–992, 1998.
112. J. Setubal and J.Meidanis. *Introduction to Computational Molecular Biology.* PWS Publishing Company, Boston, MA, 1997.
113. A.M. Shur and Y.V. Gamzova. Periods' interaction property for partial words. In T. Harju and J. Karhümaki, editors, *Words 2003*, volume 27, pages 75–82, 2003.
114. A.M. Shur and Y.V. Gamzova. Partial words and the periods' interaction property. *Izv. RAN*, 68:199–222, 2004. (see Shur, A.M., Gamzova, Y.V.: Partial words and the interaction property of periods. Izv. Math. **68** (2004) 405–428, for the English translation).
115. A.M. Shur and Y.V. Konovalova. On the periods of partial words. *LNCS*, vol. 2136, Springer, pp. 657–665, 2001.
116. H.J. Shyr. *Free Monoids and Languages.* Hon Min Book Company, Taichung, Taiwan, 1991.
117. H.J. Shyr and G. Thierrin. Disjunctive languages and codes. *LNCS*, vol.56, Springer, pp. 171–176, 1977.
118. W.F. Smyth. *Computing Patterns in Strings.* Pearson Addison-Wesley, 2003.
119. J.A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, Rockville, MD, 1988.
120. A. Thue. Uber unendliche zeichenreihen. *Norske Vid. Selsk. Skr. I, Mat. Nat. Kl. Christiana*, 7:1–22, 1906. Reprinted in Nagell, T., Selberg, A., Selberg, S., Thalberg, K. (eds.): Selected Mathematical Papers of Axel Thue. Oslo, Norway, Universitetsforlaget (1977) 139–158.

121. A. Thue. Uber die gegenseitige lage gleicher teile gewisser zeichenreihen. *Norske Vid. Selsk. Skr. I, Mat. Nat. Kl. Christiana*, 12:1–67, 1912. Reprinted in Nagell, T., Selberg, A., Selberg, S., Thalberg, K. (eds.): Selected Mathematical Papers of Axel Thue. Oslo, Norway, Universitetsforlaget (1977) 139–158.
122. R. Tijdeman and L. Zamboni. Fine and wilf words for any periods. *Indag. Math.*, 14:135–147, 2003.
123. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, 23:337–343, 1977.

3

Alignments and Approximate String Matching

Maxime Crochemore[1] and Thierry Lecroq[2]

[1] Department of Computer Science, King's College London
London WC2R 2LS, UK
Maxime.Crochemore@kcl.ac.uk
and
Université Paris Est, France
[2] LITIS, Université de Rouen
76821 Mont-Saint-Aignan Cedex, France
Thierry.Lecroq@univ-rouen.fr

The alignments constitute one of the processes used to compare strings. They allow to visualize the resemblance between strings. This chapter deals with several methods that perform the comparison of two strings in this sense. The extension to comparison methods of more than two strings is delicate, leads to algorithms whose execution time is at least exponential, and is not treated here.

The alignments are based on notions of distance or of similarity between strings. The computations are usually performed by dynamic programming. A typical example is the computation of the longest subsequence common to two strings since it shows the algorithmic techniques to implement in order to obtain an efficient computation. In particular, the reduction of the memory space obtained by one of the algorithms constitute a strategy that can often be applied in the solutions to close problems.

Section 3.1.1 describes the basic techniques for the computation of the edit (or alignment) distance and the production of the associated alignments. The chosen methodology allows to highlight a global resemblance between two strings using assumptions that simplify the computation. The search for local similarities between two strings is examined in Section 3.1.2.

The possibility of reduction of the memory space required by the computations is presented in Section 3.1.3 concerning the computation of longest common subsequences.

We are then interested Section 3.2 in the approximate search for fixed strings. More generally, approximate pattern matching consists in locating all the occurrences of factors inside a text y, of length n, that are similar to a string x, of length m. It consists in producing the positions of the factors of y that are at distance at most k from x, for a given natural integer k. We assume in the rest that $k < m \leq n$. We consider the edit distance for measuring the approximation.

The edit distance between two strings u and v, that are not necessarily of same length, is the minimal cost of the elementary edit operations between these two strings. The method at the basis for approximate pattern matching is a natural extension of the alignment method by dynamic programming of Section 3.1. It can be improved by using a restricted notion of distance obtained by considering the minimal number of edit operations rather than the sum of their costs. With this distance, the problem is known under the name of approximate pattern matching with k differences. Section 3.2 presents several solutions.

The Hamming distance between two strings u and v of same length is the number of positions in which the two strings possess different letters. With this distance, the problem is known under the name of approximate pattern matching with k mismatches. It is treated in Section 3.3.

We examine then, in Section 3.4, the case of the search for short patterns. This gives excellent practical results and is very flexible as long as the conditions of its utilization are fulfilled. The *Shift-Or algorithm* of Section 3.4 is a method that is both very fast in practice and very easy to implement. The method is flexible enough to be adapted to a wide range of similar approximate matching problems.

3.1 Alignments

An **alignment** of two strings x and y of length m and n respectively consists in aligning their symbols on vertical lines. Formally an alignment of two strings $x, y \in V$ is a word w on the alphabet $(V \cup \{\lambda\}) \times (V \cup \{\lambda\}) \setminus (\{(\lambda, \lambda)\}$ (λ is the empty word) whose projection on the first component is x and whose projection of the second component is y.

Thus an alignment $w = (\overline{x}_0, \overline{y}_0)(\overline{x}_1, \overline{y}_1) \cdots (\overline{x}_{p-1}, \overline{y}_{p-1})$ of length p is such that $x = \overline{x}_0\overline{x}_1 \cdots \overline{x}_{p-1}$ and $y = \overline{y}_0\overline{y}_1 \cdots \overline{y}_{p-1}$ with $\overline{x}_i \in V \cup \{\lambda\}$ and $\overline{y}_i \in V \cup \{\lambda\}$ for $0 \leq i \leq p-1$. The alignment is represented as follows

$$\begin{array}{llll} \overline{x}_0 & \overline{x}_1 & \cdots & \overline{x}_{p-1} \\ \overline{y}_0 & \overline{y}_1 & \cdots & \overline{y}_{p-1} \end{array}$$

with the symbol $-$ instead of the symbol λ.

An example is presented in Fig. 3.1.

```
A C G - - A
A T G C T A
```

Fig. 3.1. Alignment of ACGA and ATGCTA.

3.1.1 Global alignment

A global alignment of two strings x and y can be obtained by computing the distance between x and y. The notion of distance between two strings is widely used to compare files. The `diff` command of UNIX operating system implements an algorithm based on this notion, in which lines of the files are treated as symbols. The output of a comparison made by `diff` gives the minimum number of operations (substitute a symbol, insert a symbol, or delete a symbol) to transform one file into the other.

Let us define the edit distance between two strings x and y as follows: it is the minimum number of elementary edit operations that enable to transform x into y. The elementary edit operations are:

- the substitution of a character of x at a given position by a character of y,
- the deletion of a character of x at a given position,
- the insertion of a character of y in x at a given position.

A cost is associated with each elementary edit operation. For $a, b \in V$:

- $\mathit{Sub}(a, b)$ denotes the cost of the substitution of the character a by the character b,
- $\mathit{Del}(a)$ denotes the cost of the deletion of the character a,
- $\mathit{Ins}(a)$ denotes the cost of the insertion of the character a.

This means that the costs of the edit operations are independent of the positions where the operations occur. We can now define the edit distance of two strings x and y by

$$\mathit{edit}(x, y) = \min\{\text{cost of } \gamma \mid \gamma \in \mathit{\Gamma}_{x,y}\}$$

where $\mathit{\Gamma}_{x,y}$ is the set of all the sequences of edit operations that transform x into y, and the cost of an element $\gamma \in \mathit{\Gamma}_{x,y}$ is the sum of the costs of its elementary edit operations.

In order to compute $\mathit{edit}(x, y)$ for two strings x and y of length m and n respectively, we make use of a two-dimensional table T of $m + 1$ rows and $n + 1$ columns such that

$$T[i, j] = \mathit{edit}(x[0\,..\,i], y[0\,..\,j])$$

for $i = 0, \ldots, m-1$ and $j = 0, \ldots, n-1$. It follows $\mathit{edit}(x, y) = T[m-1, n-1]$.

The values of the table T can be computed by the following recurrence formula:

$$\begin{aligned}
T[-1, -1] &= 0\,, \\
T[i, -1] &= T[i-1, -1] + \mathit{Del}(x[i])\,, \\
T[-1, j] &= T[-1, j-1] + \mathit{Ins}(y[j])\,, \\
T[i, j] &= \min \begin{cases} T[i-1, j-1] + \mathit{Sub}(x[i], y[j])\,, \\ T[i-1, j] + \mathit{Del}(x[i])\,, \\ T[i, j-1] + \mathit{Ins}(y[j])\,, \end{cases}
\end{aligned}$$

for $i = 0, 1, \ldots, m-1$ and $j = 0, 1, \ldots, n-1$.

The value at position $[i, j]$ in the table T only depends on the values at the three neighbor positions $[i-1, j-1]$, $[i-1, j]$ and $[i, j-1]$.

The direct application of the above recurrence formula gives an exponential time algorithm to compute $T[m-1, n-1]$. However the whole table T can be computed in quadratic time, technique known as "dynamic programming". This is a general technique that is used to solve the different kinds of alignments.

The computation of the table T proceeds in two steps. First it initializes the first column and first row of T, this is done by a call to a generic function MARGIN which is an argument of the algorithm and that depends on the kind of alignment that is considered. Second it computes the remaining values of T, that is done by a call to a generic function FORMULA which is an argument of the algorithm and that depends on the kind of alignment that is considered.

```
GENERIC-DP($x, m, y, n$, MARGIN, FORMULA)
1 MARGIN($T, x, m, y, n$)
2 for $j \leftarrow 0$ to $n-1$ do
3     for $i \leftarrow 0$ to $m-1$ do
4         $T[i, j] \leftarrow$ FORMULA($T, x, i, y, j$)
5 return $T$
```

Fig. 3.2. Computation of the table T by dynamic programming.

Computing a global alignment of x and y can be done by a call to GENERIC-DP with the following arguments
(x, m, y, n, GLOBAL-MARGIN, GLOBAL-FORMULA)
(see Fig. 3.2, 3.3 and 3.4). The computation of all the values of the table T can thus be done in quadratic space and time: $O(m \times n)$.

```
GLOBAL-MARGIN($T, x, m, y, n$)
1 $T[-1, -1] \leftarrow 0$
2 for $i \leftarrow 0$ to $m-1$ do
3     $T[i, -1] \leftarrow T[i-1, -1] + Del(x[i])$
4 for $j \leftarrow 0$ to $n-1$ do
5     $T[-1, j] \leftarrow T[-1, j-1] + Ins(y[j])$
```

Fig. 3.3. Margin initialization for the computation of a global alignment.

An optimal alignment (with minimal cost) can then be produced by a call to the function ONE-ALIGNMENT($T, x, m-1, y, n-1$) (see Fig. 3.5). It consists in tracing back the computation of the values of the table T from

```
GLOBAL-FORMULA($T, x, i, y, j$)
1 return $\min\{T[i-1, j-1] + Sub(x[i], y[j])$,
    $T[i-1, j] + Del(x[i])$,
    $T[i, j-1] + Ins(y[j])\}$
```

Fig. 3.4. Computation of $T[i, j]$ for a global alignment.

position $[m-1, n-1]$ to position $[-1, -1]$. At each cell $[i, j]$ the algorithm determines among the three values $T[i-1, j-1] + Sub(x[i], y[j])$, $T[i-1, j] + Del(x[i])$ and $T[i, j-1] + Ins(y[j])$ which has been used to produce the value of $T[i, j]$. If $T[i-1, j-1] + Sub(x[i], y[j])$ has been used it adds $(x[i], y[j])$ to the optimal alignment and proceeds recursively with the cell at $[i-1, j-1]$. If $T[i-1, j] + Del(x[i])$ has been used it adds $(x[i], -)$ to the optimal alignment and proceeds recursively with cell at $[i-1, j]$. If $T[i, j-1] + Ins(y[j])$ has been used it adds $(-, y[j])$ to the optimal alignment and proceeds recursively with cell at $[i, j-1]$. Recovering all the optimal alignments can be done by a similar technique.

An example of global alignment is given in Fig. 3.6.

```
ONE-ALIGNMENT($T, x, i, y, j$)
 1 if $i = -1$ and $j = -1$ then
 2    return $(\lambda, \lambda)$
 3 else if $i = -1$ then
 4       return ONE-ALIGNMENT($T, x, -1, y, j-1$) $\cdot (\lambda, y[j])$
 5    else if $j = -1$ then
 6       return ONE-ALIGNMENT($T, x, i-1, y, -1$) $\cdot (x[i], \lambda)$
 7    else if $T[i, j] = T[i-1, j-1] + Sub(x[i], y[j])$ then
 8          return ONE-ALIGNMENT($T, x, i-1, y, j-1$) $\cdot (x[i], y[j])$
 9       else if $T[i, j] = T[i-1, j] + Del(x[i])$ then
10          return ONE-ALIGNMENT($T, x, i-1, y, j$) $\cdot (x[i], \lambda)$
11       else return ONE-ALIGNMENT($T, x, i, y, j-1$) $\cdot (\lambda, y[j])$
```

Fig. 3.5. Recovering an optimal alignment.

3.1.2 Local alignment

A local alignment of two strings x and y consists in finding the segment of x that is closer to a segment of y. The notion of distance used to compute global alignments cannot be used in that case since the segments of x closer to segments of y would only be the empty segment or individual characters. This is why a notion of similarity is used based on a scoring scheme for edit operations.

T j		-1	0	1	2	3	4	5
i		$y[j]$	`A`	`T`	`G`	`C`	`T`	`A`
-1	$x[i]$	$\boxed{0}$	1	2	3	4	5	6
0	`A`	1	$\boxed{0}$	$\boxed{1}$	$\boxed{2}$	3	4	5
1	`C`	2	1	1	2	$\boxed{2}$	3	4
2	`G`	3	2	2	1	2	$\boxed{3}$	4
3	`A`	4	3	3	2	2	3	$\boxed{3}$

Fig. 3.6. Global alignment of `ACGA` and `ATGCTA`. The values of the above table have been obtained with the following unitary costs: $\mathit{Sub}(a, b) = 1$ if $a \neq b$ and $\mathit{Sub}(a, a) = 0$, $\mathit{Del}(a) = \mathit{Ins}(a) = 1$ for $a, b \in V$.

A score (instead of a cost) is associated with each elementary edit operation. For $a, b \in V$:

- $\mathit{Sub}_S(a, b)$ denotes the score of substituting the character b for the character a,
- $\mathit{Del}_S(a)$ denotes the score of deleting the character a,
- $\mathit{Ins}_S(a)$ denotes the score of inserting the character a.

This means that the scores of the edit operations are independent of the positions where the operations occur. For two characters a and b, a positive value of $\mathit{Sub}_S(a, b)$ means that the two characters are close to each other, and a negative value of $\mathit{Sub}_S(a, b)$ means that the two characters are far apart.

We can now define the edit score of two strings x and y by

$$\mathit{sco}(x, y) = \max\{\text{score of } \gamma \mid \gamma \in \mathit{\Gamma}_{x,y}\}$$

where $\mathit{\Gamma}_{x,y}$ is the set of all the sequences of edit operations that transform x into y and the score of an element $\sigma \in \mathit{\Gamma}_{x,y}$ is the sum of the scores of its elementary edit operations.

An optimal local alignment between the strings x and y is a pair of strings (u, v) for which u is a factor of x, v is a factor of y and $\mathit{sco}(u, v)$ is maximal. For performing its computation, we consider a table T defined, for $i = -1, 0, \ldots, m-1$ and $j = -1, 0, \ldots, n-1$, by: $T[i, j]$ is the maximal similarity between a suffix of $x[0\,.\,.\,i]$ and a suffix of $y[0\,.\,.\,j]$. Or also

$$T[i, j] = \max\{\mathit{sco}(x[\ell\,.\,.\,i], y[k\,.\,.\,j]) \mid 0 \leq \ell \leq i \text{ and } 0 \leq k \leq j\} \cup \{0\}$$

is the score of the local alignment in $[i, j]$.

The values of the table T can be computed by the following recurrence formula:

$$
\begin{aligned}
T[-1,-1] &= 0\,,\\
T[i,-1] &= 0\,,\\
T[-1,j] &= 0\,,\\
T[i,j] &= \max\begin{cases}
T[i-1,j-1] + \mathit{Sub}_S(x[i],y[j])\,,\\
T[i-1,j] + \mathit{Del}_S(x[i])\,,\\
T[i,j-1] + \mathit{Ins}_S(y[j])\,,\\
0\,,
\end{cases}
\end{aligned}
$$

for $i = 0, 1, \ldots, m-1$ and $j = 0, 1, \ldots, n-1$.

Computing the values of T for a local alignment of x and y can be done by a call to Generic-DP with the following arguments
$(x, m, y, n, \text{Local-margin}, \text{Local-formula})$
in $O(mn)$ time and space complexity (see Fig. 3.2, 3.7 and 3.8). Recovering a local alignment can be done in a way similar to what is done in the case of a global alignment (see Fig. 3.5) but the trace back procedure must start at a position of a maximal value in T rather than at position $[m-1, n-1]$.

An example of local alignment is given in Fig. 3.9.

```
Local-margin($T, x, m, y, n$)
1 $T[-1,-1] \leftarrow 0$
2 for $i \leftarrow 0$ to $m-1$ do
3     $T[i,-1] \leftarrow 0$
4 for $j \leftarrow 0$ to $n-1$ do
5     $T[-1,j] \leftarrow 0$
```

Fig. 3.7. Margin initialization for computing a local alignment.

```
Local-formula($T, x, i, y, j$)
1 return max{$T[i-1,j-1] + Sub_S(x[i],y[j])$,
      $T[i-1,j] + Del_S(x[i])$,
      $T[i,j-1] + Ins_S(y[j])$,
      0}
```

Fig. 3.8. Recurrence formula for computing a local alignment.

3.1.3 Longest Common Subsequence of Two Strings

A subsequence of a string x is obtained by deleting zero or more characters from x. More formally $w[0\,.\,.\,i-1]$ is a subsequence of $x[0\,.\,.\,m-1]$ if there

(a)

T	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	E	R	D	A	W	C	Q	P	G	K	W	Y
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0	0	0	0
0	E	0	1	0	$\boxed{0}$	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	$\boxed{1}$	0	0	0	0	0	0	0	0
2	W	0	0	0	0	0	$\boxed{2}$	1	0	0	0	0	1	0
3	A	0	0	0	0	1	$\boxed{1}$	0	0	0	0	0	0	0
4	C	0	0	0	0	0	0	$\boxed{2}$	1	0	0	0	0	0
5	Q	0	0	0	0	0	0	1	$\boxed{3}$	$\boxed{2}$	1	0	0	0
6	G	0	0	0	0	0	0	0	2	1	$\boxed{3}$	2	1	0
7	K	0	0	0	0	0	0	0	1	0	2	$\boxed{4}$	3	2
8	L	0	0	0	0	0	0	0	0	0	1	3	2	1

(b)

```
A W A C Q - G K
A W - C Q P G K
```

Fig. 3.9. Computation of an optimal local alignment of $x =$ EAWACQGKL and $y =$ ERDAWCQPGKWY with scores: $Sub_S(a, a) = 1$, $Sub_S(a, b) = -3$ and $Del_S(a) = Ins_S(a) = -1$ for $a, b \in V$, $a \neq b$. **(a)** Values of table T. **(b)** The corresponding alignment.

exists an increasing sequence of integers $(k_j \mid j = 0, \ldots, i-1)$ such that for $0 \leq j \leq i-1$, $w[j] = x[k_j]$. We say that a string is an $lcs(x, y)$ if it is a **longest common subsequence** of the two strings x and y. Note that two strings can have several longest common subsequences. Their common length is denoted by $llcs(x, y)$.

A brute-force method to compute an $lcs(x, y)$ would consist in computing all the subsequences of x, checking if they are subsequences of y, and keeping the longest ones. The string x of length m has potentially 2^m subsequences, and so this method could take $O(2^m)$ time, which is impractical even for fairly small values of m.

However $llcs(x, y)$ can be computed with a two-dimensional table T by the following recurrence formula:

$$
\begin{aligned}
T[-1, -1] &= 0\,, \\
T[i, -1] &= 0\,, \\
T[-1, j] &= 0\,, \\
T[i, j] &= \begin{cases} T[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{T[i-1, j], T[i, j-1]\} & \text{otherwise,} \end{cases}
\end{aligned}
$$

for $i = 0, 1, \ldots, m-1$ and $j = 0, 1, \ldots, n-1$. Then $T[i, j] = llcs(x[0\,..\,i], y[0\,..\,j])$ and $llcs(x, y) = T[m-1, n-1]$.

Computing $T[m-1, n-1]$ can be done by a call to Generic-DP with the following arguments $(x, m, y, n,$ Local-margin, Lcs-formula$)$ in $O(mn)$ time and space complexity (see Fig. 3.2, 3.7 and 3.10).

```
FORMULA-LCS($T, x, i, y, j$)
1 if $x[i] = y[j]$ then
2     return $T[i-1, j-1] + 1$
3 else return $\max\{T[i-1, j], T[i, j-1]\}$
```

Fig. 3.10. Recurrence formula for computing an *lcs*.

It is possible afterward to trace back a path from position $[m-1, n-1]$ to exhibit an $lcs(x, y)$ in a similar way as for producing a global alignment (see Fig. 3.5). An example is presented in Fig. 3.11.

T	j	-1	0	1	2	3	4	5	6	7	8
i		$y[j]$	`C`	`A`	`G`	`A`	`T`	`A`	`G`	`A`	`G`
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0
0	`A`	0	0	$\boxed{1}$	1	1	1	1	1	1	1
1	`G`	0	0	1	$\boxed{2}$	2	2	2	2	2	2
2	`C`	0	1	1	2	2	2	2	2	2	2
3	`G`	0	1	1	2	2	2	2	$\boxed{3}$	3	3
4	`A`	0	1	2	2	3	3	3	3	$\boxed{4}$	4

Fig. 3.11. The value $T[4, 8] = 4$ is $llcs(x, y)$ for $x =$ `AGCGA` and $y =$ `CAGATAGAG`. String `AGGA` is an lcs of x and y.

3.1.4 Reducing the Space: Hirschberg Algorithm

If only the length of an $lcs(x, y)$ is required, it is easy to see that only one row (or one column) of the table T needs to be stored during the computation. The space complexity becomes $O(\min(m, n))$ as can be checked on the algorithm of Fig. 3.12. The Hirschberg algorithm computes an $lcs(x, y)$ in linear space and not only the value $llcs(x, y)$. The computation uses the algorithm of Fig. 3.12.

Let us define

$$\begin{aligned} T^*[i, n] = T^*[m, j] &= 0, \quad \text{for } 0 \le i \le m \quad \text{and} \quad 0 \le j \le n \\ T^*[m-i, n-j] &= \ llcs((x[i\,..\,m-1])^R, (y[j\,..\;n-1])^R) \\ &\qquad \text{for } 0 \le i \le m-1 \quad \text{and} \quad 0 \le j \le n-1 \end{aligned}$$

and

$$M(i) = \max_{0 \le j < n} \{T[i, j] + T^*[m-i, n-j]\}$$

where the string w^R is the reverse (or mirror image) of the string w. The following property is the key observation to compute an $lcs(x, y)$ in linear space:

```
LLCS($x, m, y, n$)
 1 for $i \leftarrow -1$ to $m-1$ do
 2     $C[i] \leftarrow 0$
 3 for $j \leftarrow 0$ to $n-1$ do
 4     $last \leftarrow 0$
 5     for $i \leftarrow -1$ to $m-1$ do
 6         if $last > C[i]$ then
 7             $C[i] \leftarrow last$
 8         else if $last < C[i]$ then
 9             $last \leftarrow C[i]$
10         else if $x[i] = y[j]$ then
11             $C[i] \leftarrow C[i] + 1$
12             $last \leftarrow last + 1$
13 return $C$
```

Fig. 3.12. $O(m)$-space algorithm to compute $llcs(x, y)$.

```
HIRSCHBERG($x, m, y, n$)
 1 if $m = 0$ then
 2     return $\lambda$
 3 else if $m = 1$ then
 4         if $x[0] \in y$ then
 5             return $x[0]$
 6         else return $\lambda$
 7     else $j \leftarrow \lfloor n/2 \rfloor$
 8         $C \leftarrow$ LLCS($x, m, y[0 \,..\, j-1], j$)
 9         $C^* \leftarrow$ LLCS($x^R, m, y[j \,..\, n-1]^R, n-j$)
10         $k \leftarrow m-1$
11         $M \leftarrow C[m-1] + C^*[m-1]$
12         for $j \leftarrow -1$ to $m-2$ do
13             if $C[j] + C^*[j] > M$ then
14                 $M \leftarrow C[j] + C^*[j]$
15                 $k \leftarrow j$
16         return HIRSCHBERG($x[0 \,..\, k-1], k, y[0 \,..\, j-1], j$)$\cdot$
       HIRSCHBERG($x[k \,..\, m-1], m-k, y[j \,..\, n-1], n-j$)
```

Fig. 3.13. $O(\min(m, n))$-space computation of $lcs(x, y)$.

$$M(i) = T[m-1, n-1], \quad \text{for } 0 \le i < m\,.$$

In the algorithm shown in Fig. 3.13 the integer j is chosen as $n/2$. After $T[i, j-1]$ and $T^*[m-i, n-j]$ ($0 \le i < m$) are computed, the algorithm finds an integer k such that $T[i, k] + T^*[m-i, n-k] = T[m-1, n-1]$. Then, recursively, it computes an $lcs(x[0 \,..\, k-1], y[0 \,..\, j-1])$ and an $lcs(x[k \,..\, m-1], y[j \,..\, n-1])$, and concatenates them to get an $lcs(x, y)$.

The running time of the Hirschberg algorithm is still $O(mn)$ but the amount of space required for the computation becomes $O(\min(m,n))$ instead of being quadratic when computed by dynamic programming.

3.2 Approximate String Matching with Differences

Approximate string matching is the problem of finding all approximate occurrences of a pattern x of length m in a text y of length n. Approximate occurrences of x are segments of y that are close to x according to a specific distance: the distance between segments and x must be not greater than a given integer k. With the edit distance (or Levenshtein distance), the problem is known as approximate string matching with k differences. The standard solutions to solve this problem consist in using the dynamic programming technique introduced in Section 3.1. We describe three variations around this technique.

Dynamic programming

We first examine a problem a bit more general for which the cost of the edit operations is not necessarily one unit. Aligning x with a factor of y amounts to align x with a prefix of y considering that the insertion of any number of letters of y at the beginning of x is not penalizing. With the table T of Section 3.1.1 we check that, to solve the problem, it is sufficient then to initialize to zero the values of the first line of the table. The positions of the occurrences are then associated with all the values of the last line of the table that are less than k.

To perform the search for approximate factors, we utilize the table R defined by

$$R[i,j] = \min\{\mathit{edit}(x[0\,.\,.\,i], y[\ell\,.\,.\,j]) \mid \ell = 0,1,\ldots,j+1\},$$

for $i = -1,0,\ldots,m-1$ and $j = -1,0,\ldots,n-1$, where *edit* is the edit distance of Section 3.1. The computation of the values of the table R utilizes the recurrence relations that follow.

For $i = 0,1,\ldots,m-1$ and $j = 0,1,\ldots,n-1$, we have:

$$\begin{aligned}
R[-1,-1] &= 0,\\
R[i,-1] &= R[i-1,-1] + \mathit{Del}(x[i]),\\
R[-1,j] &= 0,\\
R[i,j] &= \min\begin{cases} R[i-1,j-1] + \mathit{Sub}(x[i],y[j]),\\ R[i-1,j] + \mathit{Del}(x[i]),\\ R[i,j-1] + \mathit{Ins}(y[j]).\end{cases}
\end{aligned} \tag{3.1}$$

```
K-DIFF-DP$(x, m, y, n, k)$
$R[-1,-1] \leftarrow 0$
for $i \leftarrow 0$ to $m-1$ do
    $R[i,-1] \leftarrow i + Del(x[i])$
for $j \leftarrow 0$ to $n-1$ do
    $R[-1,j] \leftarrow 0$
    for $i \leftarrow 0$ to $m-1$ do
        $R[i,j] \leftarrow \min \begin{cases} R[i-1,j-1] + Sub(x[i],y[j]) \\ R[i-1,j] + Del(x[i]) \\ R[i,j-1] + Ins(y[j]) \end{cases}$
    if $R[m-1,j] \leq k$ then
        OUTPUT$(j)$
```

Fig. 3.14. Approximate string matching with k differences by dynamic programming.

(a)

R	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	C	A	G	A	T	A	A	G	A	G	A	A
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0	0	0	0
0	G	1	1	1	0	1	1	1	1	0	1	0	1	1
1	A	2	2	1	1	0	1	1	1	1	0	1	0	1
2	T	3	3	2	2	1	0	1	2	2	1	1	1	1
3	A	4	4	3	3	2	1	0	1	2	2	2	1	1
4	A	5	5	4	4	3	2	1	0	1	2	3	2	1

(b)

```
  G A T A A                    G A T A A
C A G A T - A A G A G A A    C A G A T A A G A G A A


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


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


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

Fig. 3.15. Search for $x =$ GATAA in $y =$ CAGATAAGAGAA with one difference, considering unit costs for the edit operations. **(a)** Values of table R. **(b)** The seven alignments of x with factors of y ending at positions 5, 6, 7 and 11 on y. We note that the fourth and sixth alignments give no extra information comparing to the second.

The search algorithm K-DIFF-DP whose code is given in Fig. 3.14 and that translates the recurrence of the previous proposition performs the approximate search. An example is given in Fig. 3.15.

We note that the space used by the algorithm K-DIFF-DP can be reduced to a single column by reproducing the technique of Section 3.1.3. Besides, this technique is implemented by the algorithm K-DIFF-CUT-OFF (see Fig. 3.16). As a conclusion we get the following result.

The operation K-DIFF-DP(x, m, y, n, k) that finds the factors u of y for which $\mathit{edit}(u, x) \leq k$ (edit edit distance with any costs) executes in time $O(m \times n)$ and can be realized in space $O(m)$.

Diagonal monotony

In the rest of the section, we consider that the costs of the edit operations are unitary. This is a simple case for which we can describe more efficient computation strategies that those described above. The restriction allows to state a property of monotony on the diagonals that is at the basis of the presented variations.

Since we assume that $\mathit{Sub}(a, b) = \mathit{Del}(a) = \mathit{Ins}(b) = 1$ for $a, b \in V$, $a \neq b$, the recurrence relation 3.1 simplifies and becomes

$$\begin{aligned} R[-1,-1] &= 0, \\ R[i,-1] &= i+1, \\ R[-1,j] &= 0, \\ R[i,j] &= \min \begin{cases} R[i-1,j-1] & \text{if } x[i] = y[j], \\ R[i-1,j-1]+1 & \text{if } x[i] \neq y[j], \\ R[i-1,j]+1, \\ R[i,j-1]+1. \end{cases} \end{aligned} \tag{3.2}$$

for $i = 0, 1, \ldots, m-1$ and $j = 0, 1, \ldots, n-1$.

A diagonal d of the table R consists of the positions $[i, j]$ for which $j - i = d$ ($-m \leq d \leq n$). The property of diagonal monotony expresses that the sequence of values on each diagonal of the table R increases with i and that the difference between two consecutive values is at most one (see Fig. 3.15). Before formally stating the property, we give intermediate results. The first result means that two adjacent values on a column of the table R differ from at most one. The second result is symmetrical to the first one for the lines of R.

For each position j on the string y, we have

$$-1 \leq R[i,j] - R[i-1,j] \leq 1$$

for $i = 0, 1, \ldots, m-1$.

For each position i on the string x, we have

$$-1 \le R[i,j] - R[i,j-1] \le 1$$

for $j = 0, 1, \ldots, n-1$.

We now can state the result concerning the property of monotony on the diagonals announced above:

For $i = 0, 1, \ldots, m-1$ and $j = 0, 1, \ldots, n-1$, we have:

$$R[i-1,j-1] \le R[i,j] \le R[i-1,j-1] + 1.$$

Partial computation

The property of monotony on the diagonals is exploited in the following way to avoid to compute some values in the table R that are greater than k, the maximal number of allowed differences. The values are still computed column by column, in the increasing order of the positions on y and for each column in the increasing order of the positions on x, as done by the algorithm K-diff-DP. When a value equal to $k+1$ is found in a column, it is useless to compute the next values in the same diagonal since those latter are all strictly greater than k. For pruning the computation, we keep, in each column, the largest position at which is found an admissible value. If q_j is this position, for a given column j, only the values of lines -1 to $q_j + 1$ are computed in the next column (of index $j+1$).

The algorithm K-diff-cut-off, given in Fig. 3.16, realizes this method.

```
K-diff-cut-off$(x, m, y, n, k)$
 1  for $i \leftarrow -1$ to $k-1$ do
 2      $C_1[i] \leftarrow i+1$
 3  $p \leftarrow k$
 4  for $j \leftarrow 0$ to $n-1$ do
 5      $C_2[-1] \leftarrow 0$
 6      for $i \leftarrow 0$ to $p$ do
 7          if $x[i] = y[j]$ then
 8              $C_2[i] \leftarrow C_1[i-1]$
 9          else $C_2[i] \leftarrow \min\{C_1[i-1], C_2[i-1], C_1[i]\} + 1$
10      $C_1 \leftarrow C_2$
11      while $C_1[p] > k$ do
12          $p \leftarrow p-1$
13      if $p = m-1$ then
14          Output$(j)$
15      $p \leftarrow \min\{p+1, m-1\}$
```

Fig. 3.16. Approximate string matching with k differences by partial computation.

The column -1 is initialized until line $k-1$ that corresponds to the value k. For the next columns of index $j = 0, 1, \ldots, n-1$, the values are computed until line

$$p_j = \min \begin{cases} 1 + \max\{i \mid 0 \le i \le m-1 \text{ and } R[i, j-1] \le k\}, \\ m-1. \end{cases}$$

The table R is implemented with the help of two tables C_2 and C_1 that allow to memorize respectively the values of the column during the computation and the values of the previous column. The process is similar to the one that is used in the algorithm LLCS of Section 3.1.4. At each iteration of the loop Lines 7–10, we have:

$$\begin{aligned} C_1[i-1] &= R[i-1, j-1], \\ C_2[i-1] &= R[i-1, j], \\ C_1[i] &= R[i, j-1]. \end{aligned}$$

We compute then the value $C_2[i]$ that is also $R[i, j]$. We find thus at this line an implementation of Relation 3.2. An example of computation is given in Fig. 3.17.

R	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	C	A	G	A	T	A	A	G	A	G	A	A
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0	0	0	0
0	G	1	1	1	0	1	1	1	1	0	1	0	1	1
1	A		2	1	1	0	1	1	1	1	0	1	0	1
2	T				2	1	0	1	2	2	1	1	1	1
3	A						1	0	1	2	2	2	1	1
4	A							1	0	1	2			1

Fig. 3.17. Pruning of the computation of the dynamic programing table for the search for $x =$ `GATAA` in $y =$ `CAGATAAGAGAA` with one difference (see Figure 3.15). We notice that seventeen values of table R (those that are not shown) are not useful for the computation of occurrences of approximate factors of x in y.

We note that the memory space used by the algorithm K-DIFF-CUT-OFF is $O(m)$. Indeed, only two columns are memorized. This is possible since the computation of the values for one column only needs those of the previous column.

Diagonal computation

The variant of search with differences that we consider now consists in computing the values of the table R according to the diagonals and by taking into account the property of monotony. The interesting positions on the diagonals are those where changes of values happen. These changes are incrementation because of the chosen distance.

For a number q of differences and a diagonal d, we denote by $L[q, d]$ the index i of the line on which $R[i, j] = q$ for the last time on the diagonal

R	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	`C`	`A`	`G`	`A`	`T`	`A`	`A`	`G`	`A`	`G`	`A`	`A`
-1	$x[i]$						0							
0	`G`							1						
1	`A`								1					
2	`T`									2				
3	`A`										2			
4	`A`											3		

Fig. 3.18. Values of table R on diagonal 5 for the approximate search for $x =$ `GATAA` in $y =$ `CAGATAAGAGAA`. The last occurrences of each value on the diagonal are in gray. The lines where they occur are stored in table L by the algorithm of diagonal computation. We thus have $L[0,5] = -1$, $L[1,5] = 1$, $L[2,5] = 3$, $L[3,5] = 4$.

$j - i = d$. The idea of the definition of $L[q,d]$ is shown in Fig. 3.18. Formally, for $q = 0, 1, \ldots, k$ and $d = -m, -m+1, \ldots, n-m$, we have

$$L[q,d] = i$$

if and only if i is the maximal index, $-1 \le i < m$, for which there exists an index j, $-1 \le j < n$, with

$$R[i,j] \le q \text{ and } j - i = d.$$

In other words, for fixed q, the values $L[q,d]$ mark the lowest borderline of the values less than q in the table R (gray values in Fig. 3.19).

The definition of $L[q,d]$ implies that q is the smallest number of differences between $x[0\,..\,L[q,d]]$ and a factor of the text ending at position $d + L[q,d]$ on y. It moreover implies that the letters $x[L[q,d]+1]$ and $y[d + L[q,d] + 1]$ are different when they are defined.

The values $L[q,d]$ are computed by iteration on d, for q going from 0 to $k+1$. The principle of the computation relies on Recurrence 3.2 and the above statements. A simulation of the computation on the table R is presented in Fig. 3.19.

For the approximate pattern matching with k differences problem, only the values $L[q,d]$ for which $q \le k$ are necessary. If $L[q,d] = m-1$, it means that there is an occurrence of the string x at the diagonal d with at most q differences. The occurrence ending at position $d+m-1$, this is only valid if $d + m \le n$. We get another approximate occurrences at the end of y when $L[q,d] = i$ and $d+i = n-1$; in this case the number of differences is $q+m-1-i$.

The algorithm K-DIFF-DIAG, given in Fig. 3.21 performs the approximate search for x in y by computing the values $L[q,d]$. It uses the function *lcp* where *lcp*(u,v) gives the length of the longest common prefix of two strings u and v. Let us note that the first possible occurrence of an approximate factor of x in y can end at position $m-1-k$ on y, this corresponds to diagonal $-k$. The last possible occurrence starts at position $n-m+k$ on y, this corresponds

(a)

R	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	C	A	G	A	T	A	A	G	A	G	A	A
-1	$x[i]$	0	0	0	0	0	0	0	0	0				
0	G				0					0				
1	A					0					0			
2	T						0							
3	A							0						
4	A								0					

(b)

R	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	C	A	G	A	T	A	A	G	A	G	A	A
-1	$x[i]$	0	0	0	0	0	0	0	0	0				
0	G	1	1	1	0	1	1	1	1	0				
1	A			1	1	0	1	1	1	1	0			
2	T					1	0	1			1	1		
3	A						1	0	1				1	
4	A							1	0	1				1

Fig. 3.19. Simulation of the diagonal computation for the search for x = GATAA in y = CAGATAAGAGAA with one difference (see Figure 3.15). **(a)** Values computed during the first step (Lines 8–13 for $q = 0$ of Algorithm L-DIFF-DIAG); they detect the occurrence of x at right position 6 on y (since $R[4, 6] = 0$). **(b)** Values computed during the second step (Lines 8–13 for $q = 1$); they indicate the approximate factors of x with one difference at right positions 5, 7 and 11 on y (since $R[4, 5] = R[4, 7] = R[4, 11] = 1$).

to diagonal $n - m + k$. Thus only diagonals going from $-k$ to $n - m + k$ are considered during the computation (the initialization is also done on the diagonals $-k - 1$ and $n - m + k + 1$ to simplify the writing of the algorithm). Fig. 3.20 shows the table L obtained on the example of Fig. 3.15.

d	-2	-1	0	1	2	3	4	5	6	7	8	9
$q = -1$		-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
$q = 0$	-1	-1	-1	-1	4	-1	-1	-1	-1	1	-1	
$q = 1$		0	1	4	4	4	1	1	2	4		

Fig. 3.20. Values of table L of the diagonal computation when x = GATAA, y = CAGATAAGAGAA and $k = 1$. Lines $q = 0$ and $q = 1$ correspond to a state of the computation simulated on table R in Figure 3.19. Values $4 = |\texttt{GATAA}| - 1$ on line $q = 1$ indicate the presence of occurrences of x with at most one difference ending at positions $1 + 4$, $2 + 4$, $3 + 4$ and $7 + 4$ on y.

The algorithm K-DIFF-DIAG computes the table L.

For every string x of length m, every string y of length n and every integer k such that $k < m \leq n$, the operation K-DIFF-DIAG(x, m, y, n, k) computes the approximate occurrences of x in y with at most k differences.

```
K-DIFF-DIAG$(x, m, y, n, k)$
 1 for $d \leftarrow -1$ to $n - m + k + 1$ do
 2     $L[-1, d] \leftarrow -2$
 3 for $q \leftarrow 0$ to $k - 1$ do
 4     $L[q, -q - 1] \leftarrow q - 1$
 5     $L[q, -q - 2] \leftarrow q - 1$
 6 for $q \leftarrow 0$ to $k$ do
 7     for $d \leftarrow -q$ to $n - m + k - q$ do
 8         $\ell \leftarrow \max \begin{cases} L[q-1, d-1] \\ L[q-1, d] + 1 \\ L[q-1, d+1] + 1 \end{cases}$
 9         $\ell \leftarrow \min\{\ell, m - 1\}$
10         $L[q, d] \leftarrow \ell + |lcp(x[\ell + 1 \mathinner{..} m - 1], y[d + \ell + 1 \mathinner{..} n - 1])|$
11         if $L[q, d] = m - 1$ or $d + L[q, d] = n - 1$ then
12             OUTPUT$(d + m - 1)$
```

Fig. 3.21. Approximate string matching with k differences by diagonals.

In the way that the algorithm K-DIFF-DIAG is described, the memory space for the computation is principally used by the table L. We note that it is sufficient to memorize a single line to correctly perform the computation, this gives an implementation in space $O(n)$. It is however possible to reduce the space to $O(m)$ obtaining a space comparable to algorithm K-DIFF-CUT-OFF.

If the computation of $lcp(u, v)$ is realized in time $O(|lcp(u, v)|)$, the algorithm K-DIFF-DIAG executes in time $O(m \times n)$. But it is possible to prepare the strings x and y in such a way that any $lcp(u, v)$ query is answered in constant time. For this, we utilize the suffix tree, of the string $z = x\$y$ where $\$ \notin alph(y)$. The string

$$w = lcp(x[\ell + 1 \mathinner{..} m - 1], y[d + \ell + 1 \mathinner{..} n - 1])$$

is nothing else but the string $lcp(x[\ell + 1 \mathinner{..} m - 1]\$y, y[d + \ell + 1 \mathinner{..} n - 1])$ since $\$ \notin alph(y)$. Let f and g be the external nodes of the suffix tree associated with suffixes of $x[\ell + 1 \mathinner{..} m - 1]\$y$ and $y[d + \ell + 1 \mathinner{..} n - 1]$ of the string z. Their common prefix of maximal length is then the label of the path leading from the initial state to the lowest node that is a common ancestor to f and g. This reduces the computation of w to the computation of this node.

The problem of the common ancestor that we are interested in here is the one for which the tree is static. A linear preprocessing of the tree allows to get a response in constant time to the queries (see notes). The consequence of this result is that on a fixed alphabet, after preparation of the strings x and y in linear time, it is possible to execute the algorithm K-DIFF-DIAG in time $O(k \times n)$.

3.3 Approximate String Matching with Mismatches

In this section, we are interested in the search for all the occurrences of a string x of length m in a string y of length n with at most k mismatches ($k \in N$, $k < m \leq n$). The Hamming distance between two strings u and v of same length is the number of mismatches between u and v and is defined by:

$$\mathit{Ham}(u, v) = \mathit{card}\{i \mid u[i] \neq v[i], i = 0, 1, \ldots, |u| - 1\}.$$

The problem can then be expressed as the search for all the positions $j = 0, 1, \ldots, n - m$ on y that satisfy the inequality $\mathit{Ham}(x, y[j \mathinner{.\,.} j + m - 1]) \leq k$.

3.3.1 Search automaton

A natural solution to this problem consists in using an automaton that recognizes the language $V^*\{w \mid \mathit{Ham}(x, w) \leq k\}$. To do this, we can consider the non-deterministic automaton defined as follows:

- each state is a pair (ℓ, i) where ℓ is the level of the state and i is its depth, with $0 \leq \ell \leq k$, $-1 \leq i \leq m - 1$ and $\ell \leq i + 1$;
- the initial state is $(0, -1)$;
- the terminal states are of the form $(\ell, m - 1)$ with $0 \leq \ell \leq k$;
- the transitions are, for $0 \leq \ell \leq k$, $0 \leq i < m - 1$ and $a \in V$, either of the form $((0, -1), a, (0, -1))$, or of the form $((\ell, i), x[i + 1], (\ell, i + 1))$, or of the form $((\ell, i), a, (\ell + 1, i + 1))$ if $a \neq x[i + 1]$ and $0 \leq \ell \leq k - 1$.

The automaton possesses $k + 1$ levels, each level ℓ allowing to recognize the prefixes of x with ℓ mismatches. The transitions of the form $((\ell, i), a, (\ell, i + 1))$ correspond to the equality of letters while those of the form $((\ell, i), a, (\ell + 1, i + 1))$ correspond to the inequality of letters. The loop on the initial state allows to find all the occurrences of the searched factors. During the analysis of the text with the automaton, if a terminal state $(\ell, m - 1)$ is reached, this indicates the presence of an occurrence of x with exactly ℓ mismatches.

It is clear that the automaton possesses $(k + 1) \times (m + 1 - \frac{k}{2})$ states and that it can be build in time $O(k \times m)$. An example is shown in Fig. 3.22. Unfortunately, the total number of states obtained by determinizing the automaton is

$$\Theta(\min\{m^{k+1}, (k + 1)!(k + 2)^{m-k+1}\}).$$

We can check that a direct simulation of the automaton produces a search algorithm whose execution time is $O(m \times n)$ using the dynamic programming as in the previous section. Actually by using a method adapted to the problem we get, in the rest, an algorithm that performs the search in time $O(k \times n)$. This produces a solution of same complexity as the one of algorithm K-DIFF-DIAG that nevertheless solves a more general problem. But the solution that follows is based on a simple management of lists without using a search algorithm for common ancestor.

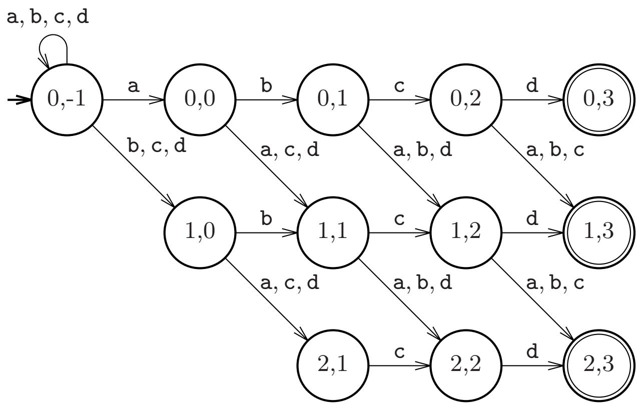


Fig. 3.22. The (non-deterministic) automaton of approximate pattern matching with two mismatches for the string abcd on the alphabet $V = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$.

3.3.2 Specific implementation

We show how to reduce the execution time of the simulation of the previous automaton. To obtain the desired time, we utilize during the search a queue F of positions that stores detected mismatches. Its update is done by letter comparisons, but also by merging with queues associated with string x. The sequences that they represent are defined as follows.

For a shift q of x, $1 \leq q \leq m-1$, $G[q]$ is the increasing sequence, of maximal length $2k+1$, of the positions on x of the leftmost mismatches between $x[q\,..\,m-1]$ and $x[0\,..\,m-q-1]$. The sequences are determined during a preprocessing phase that is described at the end of the section.

The searching phase consists in performing attempts at all the positions $j = 0, 1, \ldots, n-m$ on y. During the attempt at position j, we scan the factor $y[j\,..\,j+m-1]$ of the text and the generic situation is the following (see Fig. 3.23): the prefix $y[j\,..\,g]$ of the window has already been scanned during

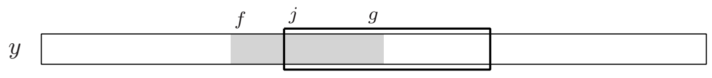


Fig. 3.23. Variables of Algorithm K-MISMATCHES. During the attempt at position j, variables f and g spot a previous attempt . The mismatches between $y[f\,..\,g]$ and $x[0\,..\,g-f]$ are stored in the queue F.

a previous attempt at position f, $f < j$, and no comparison already happens on the suffix $y[g+1\,..\,n-1]$ of the text. During the comparison of the already scanned part of the text, $y[j\,..\,g]$, around k tests can be necessary. Fig. 3.24 shows a computation example.

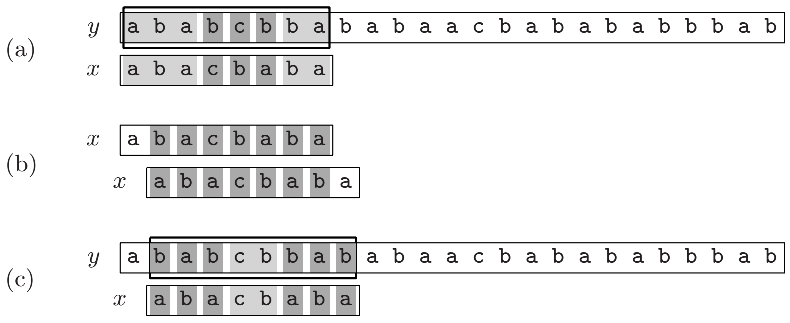

(a)
y a b a b c b b a b a b a a c b a b a b a b b b a b
x a b a c b a b a
(b)
x a b a c b a b a
x a b a c b a b a
(c)
y a b a b c b b a b a b a a c b a b a b a b b b a b
x a b a c b a b a

Fig. 3.24. Search with mismatches of the string x = abacbaba in the text y = ababcbbababaacbababababbbab. **(a)** Occurrence of the string with exactly three mismatches at position 0 on y. The queue F of mismatches contains positions 3, 4 and 5 on x. **(b)** Shift of length 1. There are seven mismatches between $x[0\,.\,.\,6]$ and $x[1\,.\,.\,7]$, this corresponds to the fact that $G[1]$ contains the sequence $\langle 1,2,3,4,5,6,7\rangle$ (see Figure 3.26). **(c)** Attempt at position 1: the factor $y[1\,.\,.\,7]$ has already been considered but the letter $y[8] =$ b has never been compared yet. The mismatches at positions 0, 1, 5 and 6 on x can be deduced from the merge of the queues F and $G[1]$. Three letter comparisons are necessary at positions 2, 3 and 4 in order to find the mismatch at position 2 since these three positions are simultaneously in F and $G[1]$. An extra comparison provides the mismatch at position 7.

The positions of the mismatches detected during the attempt at position f are stored in a queue F. Their computation is done by scanning the positions in increasing order. For the search with k mismatches, we only keep in F at most $k+1$ mismatches (the leftmost ones). Considering a possible $(k+1)$-th mismatch amounts to compute the longest prefix of x that possesses exactly k mismatches with the aligned factor of y.

The code of the search algorithm with mismatches, K-MISMATCHES, is given in Fig. 3.25. The processing at position j proceeds in two steps. It first starts by comparing the factors $x[0\,.\,.\,g-j]$ and $y[j\,.\,.\,g]$ using the queues F and $G[j-f]$. The comparison amounts to perform a merge of these two queues (Line 8); this merge is described further. The second step is only applied when the obtained sequence contains less than k positions. It resumes the scanning of the window by simple letter comparisons (Lines 11–18). This is during this step that an occurrence of an approximate factor can be detected.
An example of table G and of successive values of the queue F of the mismatches is presented in Fig. 3.26.

In the algorithm K-MISMATCHES, the positions stored in the queues F or J are positions on x. They indicate mismatches between x and the factor aligned at position f on y. Thus, if p occurs in the queue, we have $x[p] \neq y[f+p]$. When the variable f is updated, the origin of the factor of y is replaced by j, and we should thus perform a translation, that is to say to decrease the

```
K-MISMATCHES($x, m, G, y, n, k$)
 1  $F \leftarrow$ EMPTY-QUEUE()
 2  $(f, g) \leftarrow (-1, -1)$
 3  for $j \leftarrow 0$ to $n - m$ do
 4      if LENGTH($F$) $> 0$ and HEAD($F$) $= j - f - 1$ then
 5          DEQUEUE($F$)
 6      if $j \leq g$ then
 7          $J \leftarrow$ MIS-MERGE($f, j, g, F, G[j - f]$)
 8      else $J \leftarrow$ EMPTY-QUEUE()
 9      if LENGTH($J$) $\leq k$ then
10          $F \leftarrow J$
11          $f \leftarrow j$
12          do
13              $g \leftarrow g + 1$
14              if $x[g - j] \neq y[g]$ then
15                  ENQUEUE($F, g - j$)
16          while LENGTH($F$) $\leq k$ and $g < j + m - 1$
17          if LENGTH($F$) $\leq k$ then
18              OUTPUT($j$)
```

Fig. 3.25. Approximate string matching with k mismatches.

positions by the quantity $j - f$. This is realized in the algorithm MIS-MERGE during the addition of a position in the output queue.

If the merge realized by the algorithm MIS-MERGE executes in linear time, the execution time of the algorithm K-MISMATCHES is $O(k \times n)$ in space $O(k \times m)$.

3.3.3 Merge

The aim of the operation MIS-MERGE($f, j, g, F, G[j - f]$) (Line 8 of the algorithm K-MISMATCHES) is to produce the sequence of positions of the mismatches between the strings $x[0 \,..\, g-j]$ and $y[j \,..\, g]$, relying on the knowledge of the mismatches stored in the queues F and $G[j-f]$. This algorithm is given in Fig. 3.28.

The positions p in F mark the mismatches between $x[0 \,..\, g-f]$ and $y[f \,..\, g]$, but only those that satisfy the inequality $f + p \geq j$ (by definition of F we already have $f + p \leq g$) are useful to the computation The objective of the test in Line 5 of the algorithm K-MISMATCHES is precisely to delete from F the useless values. The positions q of $G[j - f]$ denote the mismatches between $x[j - f \,..\, m - 1]$ and $x[0 \,..\, m - j + f - 1]$. Those that are useful must satisfy the inequality $f + q \leq g$ (we already have $f + q \geq j$). The test in Line 19 of the algorithm MIS-MERGE takes into account this constraint. Fig. 3.27 illustrates the merge (see also Fig. 3.24).

Let us consider a position p on x such that $j \leq f + p \leq g$. If p occurs in F, this means that $y[f + p] \neq x[p]$. If p is in $G[j - f]$, this means that

i	$x[i]$	$G[i]$
0	a	$\langle\rangle$
1	b	$\langle 1,2,3,4,5,6,7\rangle$
2	a	$\langle 3,4,5\rangle$
3	c	$\langle 3,6,7\rangle$
4	b	$\langle 4,5,6,7\rangle$
5	a	$\langle\rangle$
6	b	$\langle 6,7\rangle$
7	a	$\langle\rangle$

(a)

j	$y[j]$	F
0	a	$\langle 3,4,5\rangle$
1	b	$\langle 0,1,2,5\rangle$
2	a	$\langle 2,3\rangle$
3	b	$\langle 0,1,2,3\rangle$
4	c	$\langle 0,2,3\rangle$
5	b	$\langle 0,3,4,5\rangle$
6	b	$\langle 0,1,2,3\rangle$
7	a	$\langle 3,4,6,7\rangle$
8	b	$\langle 0,1,2,3\rangle$
9	a	$\langle 3,4,5,6\rangle$
10	b	$\langle 0,1\rangle$
11	a	$\langle 1,2,3,4\rangle$
12	a	$\langle 1,2,3\rangle$
13	c	$\langle 3,4,5,7\rangle$
14	b	$\langle 0,1,2,3\rangle$
15	a	$\langle 3,4,5,7\rangle$
16	b	$\langle 0,1,2,3\rangle$
17	a	$\langle 3,5,6,7\rangle$

(b)

Fig. 3.26. Queues used for the approximate search with three mismatches of $x =$ abacbaba in $y =$ ababcbbababaacbabababbbab. **(a)** Values of table G for string abacbaba. The queue $G[3]$ for instance contains 3, 6 and 7, positions on x of the mismatches between its suffix cbaba and its prefix abacb. **(b)** Successive values of queue F of the mismatches computed by Algorithm K-MISMATCHES. The values at positions 0, 2, 4, 10 and 12 on y possess less than three elements, which reveals the presence of occurrences of x with at most three mismatches at these positions. At position 0, for instance, the factor ababcbba of y possesses exactly three mismatches with x: they are at positions 3, 4 and 5 on x.

$x[p] \neq x[p-j+f]$. Four situations can arise for a position p whether it occurs or not in F and $G[j-f]$. (see Fig. 3.24 and 3.27):

1. The position p is neither in F nor in $G[j-f]$. We have $y[f+p] = x[p]$ and $x[p] = x[p-j+f]$, thus $y[f+p] = x[p-j+f]$.
2. The position p is in F but not in $G[j-f]$. We have $y[f+p] \neq x[p]$ and $x[p] = x[p-j+f]$, thus $y[f+p] \neq x[p-j+f]$.
3. The position p is in $G[j-f]$ but not in F. We have $y[f+p] = x[p]$ and $x[p] \neq x[p-j+f]$, thus $y[f+p] \neq x[p-j+f]$.
4. The position p is in F and in $G[j-f]$. We have $y[f+p] \neq x[p]$ and $x[p] \neq x[p-j+f]$, this does not allow to conclude on the equality between $y[f+p]$ and $x[p-j+f]$.

Among the enumerated cases, only the last three can lead to a mismatch between the letters $y[f+p]$ and $x[p-j+f]$. Only the last case requires an

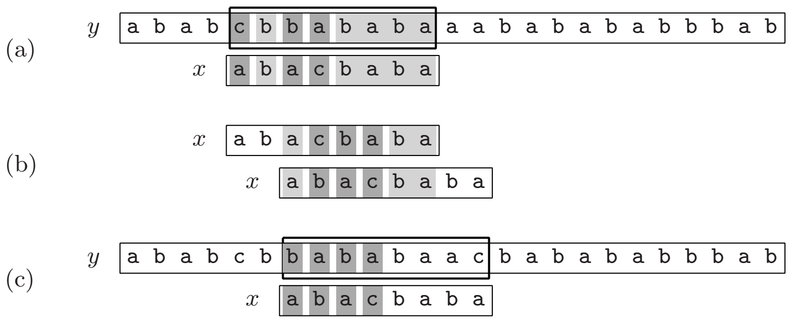

(a)
y a b a b c b b a b a b a a a b a b a b a b b b a b
x a b a c b a b a
(b)
x a b a c b a b a
x a b a c b a b a
(c)
y a b a b c b b a b a b a a c b a b a b a b b b a b
x a b a c b a b a

Fig. 3.27. Merge during the search with three mismatches of x = `abacbaba` in y = `ababcbbababaacbababababbbab`. **(a)** Occurrence of x at position 4 on y with three mismatches at positions 0, 2 and 3 on x; $F = \langle 0, 2, 3\rangle$. **(b)** There are three mismatches between $x[2\,..\,7]$ and $x[0\,..\,5]$; $G[2] = \langle 3, 4, 5\rangle$. **(c)** The sequences conserved for the merge are $\langle 2, 3\rangle$ and $\langle 3, 4, 5\rangle$, and this latter produces the sequence $\langle 2, 3, 4, 5\rangle$ of positions of the four first mismatches between x and $y[6\,..\,13]$. A single letter comparison is necessary at position 3, between $x[1]$ and $y[7]$, since the other positions only occur in one of the two sequences.

extra comparison of letters. They are processed in this respective order at Lines 7–8, 10–11 and 12–15 of the algorithm of merge.

The algorithm Mis-merge (see Fig. 3.28) executes in linear time.

3.3.4 Correctness proof

The correctness proof of the algorithm K-mismatches relies on the proof of the function Mis-merge. One of the main arguments of the proof is a property of the Hamming distance that is stated below.

Let u, v and w be three strings of same length. Let us set $d = \mathit{Ham}(u, v)$, $d' = \mathit{Ham}(v, w)$, and assume $d' \le d$. We then have:

$$d - d' \le \mathit{Ham}(u, w) \le d + d'.$$

When the operation Mis-merge$(f, j, g, F, G[j - f])$ is executed in the algorithm K-mismatches, the next conditions are satisfied:

1. $f < j \le g \le f + m - 1$;
2. $F = \langle p \mid x[p] \ne y[f + p]$ and $j \le f + p \le g)\rangle$;
3. $x[g - f] \ne y[g]$;
4. Length$(F) \le k + 1$;
5. $G = \langle p \mid x[p] \ne x[p - j + f]$ and $j \le f + p \le g'\rangle$ for an integer g' such that $j \le g' \le f + m - 1$.

Moreover, if $g' < f + m - 1$, Length$(G) = 2k + 1$ by definition of G. By taking these conditions as assumption we get the following result.

Let J = Mis-merge$(f, j, g, F, G[j - f])$. If Length$(J) \le k$,

```
Mis-merge($f, j, g, F, G$)
 1 $J \leftarrow$ Empty-Queue()
 2 while Length($J$) $\leq k$ and Length($F$) $> 0$
           and Length($G$) $> 0$ do
 3   $p \leftarrow$ Head($F$)
 4   $q \leftarrow$ Head($G$)
 5   if $p < q$ then
 6     Dequeue($F$)
 7     Enqueue($J, p - j + f$)
 8   else if $q < p$ then
 9     Dequeue($G$)
10     Enqueue($J, q - j + f$)
11   else Dequeue($F$)
12     Dequeue($G$)
13     if $x[p - j + f] \neq y[f + p]$ then
14       Enqueue($J, p - j + f$)
15 while Length($J$) $\leq k$ and Length($F$) $> 0$ do
16   Dequeued($F, p$)
17   Enqueue($J, p - j + f$)
18 while Length($J$) $\leq k$ and Length($G$) $> 0$
           and Head($G$) $\leq g - f$ do
19   Dequeued($G, q$)
20   Enqueue($J, q - j + f$)
21 return $J$
```

Fig. 3.28. Algorithm for merging queues.

$$J = \langle p \mid x[p] \neq y[j+p] \text{ and } j \leq j+p \leq g \rangle,$$

and, in the contrary case,

$$Ham(y[j\,..\,g], x[0\,..\,g-j]) > k.$$

The result that follows is on the correctness of algorithm K-mismatches. It assumes that the sequences $G[q]$ are computed in accordance with their definition.

If $x, y \in V^*$, $m = |x|$, $n = |y|$, $k \in N$ and $k < m \leq n$, the algorithm K-mismatches detects all the positions $j = 0, 1, \ldots, n-m$ on y for which $Ham(x, y[j\,..\,j+m-1]) \leq k$.

3.3.5 Preprocessing

The aim of the preprocessing phase is to compute the values of the table G that is required by the algorithm K-mismatches. Let us recall that for a shift q of x, $1 \leq q \leq m-1$, $G[q]$ is the increasing sequence of positions on x of the leftmost mismatches between $x[q\,..\,m-1]$ and $x[0\,..\,m-q-1]$, and that this sequence is limited to $2k+1$ elements.

The algorithm Pre-K-mismatches is given in Fig. 3.29. The computation of the sequences $G[q]$ is realized in an elementary way by the function whose code follows.

```
Pre-K-mismatches($x, m, k$)
for $q \leftarrow 1$ to $m-1$ do
    $G[q] \leftarrow$ Empty-Queue()
    $i \leftarrow q$
    while Length($G[q]$) $< 2k+1$ and $i < m$ do
        if $x[i] \neq x[i-q]$ then
            Enqueue($G[q], i$)
        $i \leftarrow i+1$
return $G$
```

Fig. 3.29. Preprocessing for the approximate string matching with mismatches.

The execution time of the algorithm is $O(m^2)$, but it is possible to prepare the table in time $O(k \times m \times \log m)$.

3.4 Shift-Or Algorithm

We are interested in this Section in the case of the search for short patterns. We first present an algorithm to solve the exact string matching problem, but that extends readily to the approximate string matching problems.

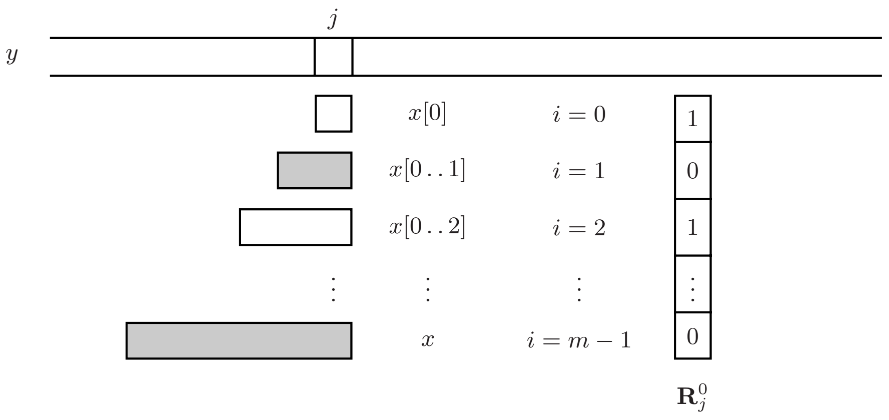


Fig. 3.30. Meaning of vector $\mathbf{R}_j^0$. Each matching prefix of x is associated with value 1 in R_j^0.

Let $\mathbf{R}^0$ be a bit array of size m. Vector $\mathbf{R}^0_j$ is the value of the entire array $\mathbf{R}^0$ after text character $y[j]$ has been processed (see Fig. 3.30). It contains information about all matches of prefixes of x that end at position j in the text. It is defined, for $0 \le i \le m-1$, by

$$\mathbf{R}^0_j[i] = \begin{cases} 0 & \text{if } x[0\,..\,i] = y[j-i\,..\,j], \\ 1 & \text{otherwise.} \end{cases}$$

Therefore, $\mathbf{R}^0_j[m-1] = 0$ is equivalent to saying that an (exact) occurrence of the pattern x ends at position j in y.

The vector $\mathbf{R}^0_j$ can be computed after $\mathbf{R}^0_{j-1}$ by the following recurrence relation:

$$\mathbf{R}^0_j[i] = \begin{cases} 0 & \text{if } \mathbf{R}^0_{j-1}[i-1] = 0 \text{ and } x[i] = y[j], \\ 1 & \text{otherwise,} \end{cases}$$

and

$$\mathbf{R}^0_j[0] = \begin{cases} 0 & \text{if } x[0] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

The transition from $\mathbf{R}^0_{j-1}$ to $\mathbf{R}^0_j$ can be computed very fast as follows. For each $a \in V$, let S_a be a bit array of size m defined, for $0 \le i \le m-1$, by

$$S_a[i] = 0 \quad \text{iff} \quad x[i] = a.$$

The array S_a denotes the positions of the character a in the pattern x. All arrays S_a are preprocessed before the search starts. And the computation of $\mathbf{R}^0_j$ reduces to two operations, SHIFT and OR:

$$\mathbf{R}^0_j = \text{SHIFT}(\mathbf{R}^0_{j-1}) \quad \text{OR} \quad S_{y[j]}\,.$$

An example is given in Fig. 3.31.

Approximate String Matching with k Mismatches

The Shift-Or algorithm easily adapts to support approximate string matching with k mismatches. To simplify the description, we shall present the case where at most one substitution is allowed.

We use arrays $\mathbf{R}^0$ and S as before, and an additional bit array $\mathbf{R}^1$ of size m. Vector $\mathbf{R}^1_{j-1}$ indicates all matches with at most one substitution up to the text character $y[j-1]$. The recurrence on which the computation is based splits into two cases.

- There is an exact match on the first i characters of x up to $y[j-1]$ (i.e., $\mathbf{R}^0_{j-1}[i-1] = 0$). Then, substituting $x[i]$ to $y[j]$ creates a match with one substitution (see Fig. 3.32). Thus,

$$\mathbf{R}^1_j[i] = \mathbf{R}^0_{j-1}[i-1]\,.$$

$S_{\mathtt{A}}$	$S_{\mathtt{C}}$	$S_{\mathtt{G}}$	$S_{\mathtt{T}}$
1	1	0	1
0	1	1	1
1	1	1	0
0	1	1	1
0	1	1	1

	C	A	G	A	T	A	A	G	A	G	A	A
G	1	1	0	1	1	1	1	0	1	0	1	1
A	1	1	1	0	1	1	1	1	0	1	0	1
T	1	1	1	1	0	1	1	1	1	1	1	1
A	1	1	1	1	1	0	1	1	1	1	1	1
A	1	1	1	1	1	1	0	1	1	1	1	1

Fig. 3.31. String $x = \mathtt{GATAA}$ occurs at position 2 in $y = \mathtt{CAGATAAGAGAA}$.

- There is a match with one substitution on the first i characters of x up to $y[j-1]$ and $x[i] = y[j]$. Then, there is a match with one substitution of the first $i+1$ characters of x up to $y[j]$ (see Fig. 3.33). Thus,

$$\mathbf{R}^1_j[i] = \begin{cases} \mathbf{R}^1_{j-1}[i-1] & \text{if } x[i] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

This implies that $\mathbf{R}^1_j$ can be updated from $\mathbf{R}^1_{j-1}$ by the relation:

$$\mathbf{R}^1_j = \big(\text{SHIFT}\big(\mathbf{R}^1_{j-1}\big) \quad \text{OR} \quad S_{y[j]}\big) \quad \text{AND} \quad \text{SHIFT}\big(\mathbf{R}^0_{j-1}\big)\,.$$

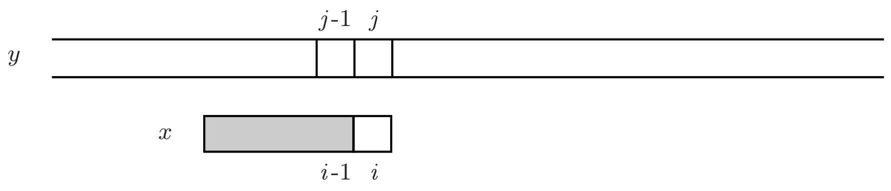


Fig. 3.32. If $\mathbf{R}^0_{j-1}[i-1] = 0$ then $\mathbf{R}^1_j[i] = 0$.

An example is presented in Fig. 3.34.

Approximate String Matching with k Differences

We show in this section how to adapt the Shift-Or algorithm to the case of only one insertion, and then dually to the case of only one deletion. The method is based on the following elements.

One insertion is allowed: here, vector $\mathbf{R}^1_{j-1}$ indicates all matches with at most one insertion up to text character $y[j-1]$. $\mathbf{R}^1_{j-1}[i-1] = 0$ if the first i

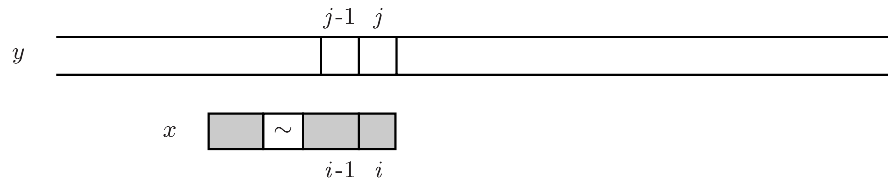


Fig. 3.33. $\mathbf{R}^1_j[i] = \mathbf{R}^1_{j-1}[i-1]$ if $x[i] = y[j]$.

	C	A	G	A	T	A	A	G	A	G	A	A
G	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	1	0	1	0	0	1	0	1	0	0
T	1	1	1	1	0	1	1	1	1	0	1	0
A	1	1	1	1	1	0	1	1	1	1	0	1
A	1	1	1	1	1	1	0	1	1	1	1	0

Fig. 3.34. String $x =$ `GATAA` occurs at positions 2 and 7 in $y =$ `CAGATAAGAGAA` with no more than one mismatch.

characters of x ($x[0\,.\,.\,i-1]$) match i symbols of the last $i+1$ text characters up to $y[j-1]$. Array $\mathbf{R}^0$ is maintained as before, and we show how to maintain array $\mathbf{R}^1$. Two cases arise.

- There is an exact match on the first $i+1$ characters of x ($x[0\,.\,.\,i]$) up to $y[j-1]$. Then inserting $y[j]$ creates a match with one insertion up to $y[j]$ (see Fig. 3.35). Thus,
$$\mathbf{R}^1_j[i] = \mathbf{R}^0_{j-1}[i]\,.$$
- There is a match with one insertion on the i first characters of x up to $y[j-1]$. Then if $x[i] = y[j]$ there is a match with one insertion on the first $i+1$ characters of x up to $y[j]$ (see Fig. 3.36). Thus,
$$\mathbf{R}^1_j[i] = \begin{cases} \mathbf{R}^1_{j-1}[i-1] & \text{if } x[i] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

This shows that $\mathbf{R}^1_j$ can be updated from $\mathbf{R}^1_{j-1}$ with the formula

$$\mathbf{R}^1_j = \big(\mathrm{SHIFT}\big(\mathbf{R}^1_{j-1}\big) \quad \mathrm{OR} \quad S_{y[j]}\big) \quad \mathrm{AND} \quad \mathbf{R}^0_{j-1}\,.$$

An example is given in Fig. 3.37.

One deletion is allowed: we assume here that $\mathbf{R}^1_{j-1}$ indicates all possible matches with at most one deletion up to $y[j-1]$. As in the previous solution, two cases arise.

- There is an exact match on the first i characters of x ($x[0\,.\,.\,i-1]$) up to $y[j]$ (i.e., $\mathbf{R}^0_j[i-1] = 0$). Then, deleting $x[i]$ creates a match with one deletion (see Fig. 3.38). Thus,

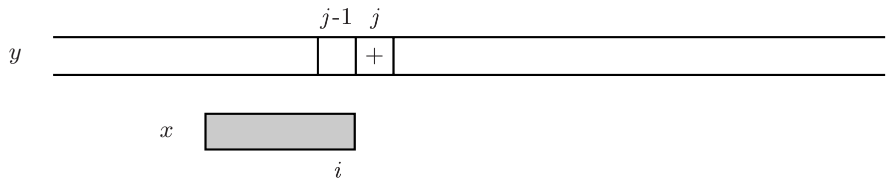


Fig. 3.35. If $\mathbf{R}^0_{j-1}[i] = 0$ then $\mathbf{R}^1_j[i] = 0$.

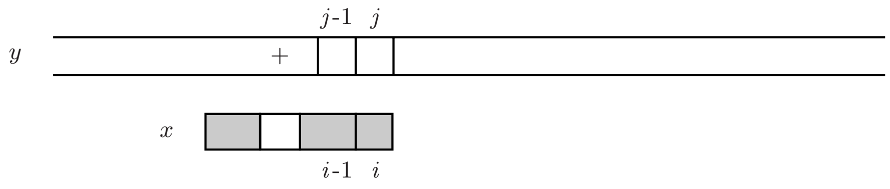


Fig. 3.36. $\mathbf{R}^1_j[i] = \mathbf{R}^1_{j-1}[i-1]$ if $x[i] = y[j]$.

	C	A	G	A	T	A	A	G	A	G	A	A
G	1	1	1	0	1	1	1	1	0	1	0	1
A	1	1	1	1	0	1	1	1	1	0	1	0
T	1	1	1	1	1	0	1	1	1	1	1	1
A	1	1	1	1	1	1	0	1	1	1	1	1
A	1	1	1	1	1	1	1	0	1	1	1	1

Fig. 3.37. `GATAAG` is an occurrence of $x =$ `GATAA` with exactly one insertion in $y =$ `CAGATAAGAGAA`.

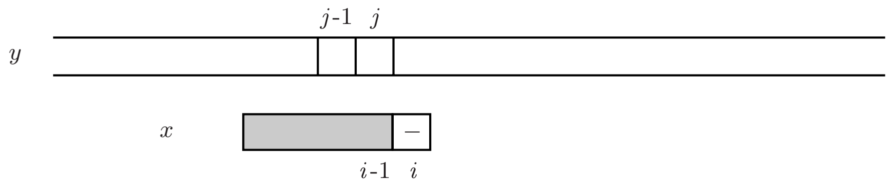


Fig. 3.38. If $\mathbf{R}^0_j[i] = 0$ then $\mathbf{R}^1_j[i] = 0$.

$$\mathbf{R}^1_j[i] = \mathbf{R}^0_j[i-1]\,.$$

- There is a match with one deletion on the first i characters of x up to $y[j-1]$ and $x[i] = y[j]$. Then, there is a match with one deletion on the first $i+1$ characters of x up to $y[j]$ (see Fig. 3.39). Thus,

$$\mathbf{R}^1_j[i] = \begin{cases} \mathbf{R}^1_{j-1}[i-1] & \text{if } x[i] = y[j],\\ 1 & \text{otherwise.} \end{cases}$$

The discussion provides the following formula used to update $\mathbf{R}^1_j$ from $\mathbf{R}^1_{j-1}$:

$$\mathbf{R}^1_j = \big(\text{SHIFT}\big(\mathbf{R}^1_{j-1}\big) \quad \text{OR} \quad S_{y[j]}\big) \quad \text{AND} \quad \text{SHIFT}\big(\mathbf{R}^0_j\big).$$

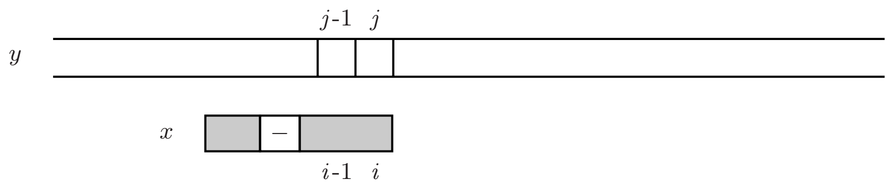


Fig. 3.39. $\mathbf{R}^1_j[i] = \mathbf{R}^1_{j-1}[i-1]$ if $x[i] = y[j]$.

An example is presented in Fig. 3.40.

	C	A	G	A	T	A	A	G	A	G	A	A
G	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	0	0	1	0	0	0	0	0	0	0
T	1	1	1	0	0	1	1	1	0	1	0	1
A	1	1	1	1	0	0	1	1	1	1	1	0
A	1	1	1	1	1	0	0	1	1	1	1	1

Fig. 3.40. `GATA` and `ATAA` are two occurrences with one deletion of $x =$ `GATAA` in $y =$ `CAGATAAGAGAA`

Wu–Manber Algorithm

We present in this section a general solution for the approximate string matching problem with at most k differences of the types: insertion, deletion, and substitution. It is an extension of the problems presented above. The following algorithm maintains $k+1$ bit arrays $\mathbf{R}^0, \mathbf{R}^1, \ldots, \mathbf{R}^k$ that are described now. The vector $\mathbf{R}^0$ is maintained similarly as in the exact matching case. The other vectors are computed with the formula $(1 \leq \ell \leq k)$

$$\begin{aligned}\mathbf{R}^\ell_j = \big(&\text{SHIFT}\big(\mathbf{R}^\ell_{j-1}\big) \quad \text{OR} \quad S_{y[j]}\big)\\ &\text{AND} \quad \text{SHIFT}\big(\mathbf{R}^{\ell-1}_j\big)\\ &\text{AND} \quad \text{SHIFT}\big(\mathbf{R}^{\ell-1}_{j-1}\big)\\ &\text{AND} \quad \mathbf{R}^{\ell-1}_{j-1}\end{aligned}$$

which can be rewritten into

$$\begin{aligned}\mathbf{R}^\ell_j = \big(&\text{SHIFT}\big(\mathbf{R}^\ell_{j-1}\big) \quad \text{OR} \quad S_{y[j]}\big)\\ &\text{AND} \quad \text{SHIFT}\big(\mathbf{R}^{\ell-1}_j \quad \text{AND} \quad \mathbf{R}^{\ell-1}_{j-1}\big)\\ &\text{AND} \quad \mathbf{R}^{\ell-1}_{j-1}.\end{aligned}$$

	C	A	G	A	T	A	A	G	A	G	A	A
G	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	0	0	0	0	0	0
T	1	1	1	0	0	0	1	1	0	0	0	0
A	1	1	1	1	0	0	0	1	1	1	0	0
A	1	1	1	1	1	0	0	0	1	1	1	0

Fig. 3.41. Here $x =$ `GATAA` and $y =$ `CAGATAAGAGAA` and $k = 1$. The output 5, 6, 7, and 11 corresponds to the segments `GATA`, `GATAA`, `GATAAG`, and `GAGAA` which approximate the pattern `GATAA` with no more than one difference.

```
WM($x, m, y, n, k$)
for each character $a \in V$ do
    $S_a \leftarrow 1^m$
for $i \leftarrow 0$ to $m-1$ do
    $S_{x[i]}[i] \leftarrow 0$
$\mathbf{R}^0 \leftarrow 1^m$
for $\ell \leftarrow 1$ to $k$ do
    $\mathbf{R}^\ell \leftarrow$ SHIFT($\mathbf{R}^{\ell-1}$)
for $j \leftarrow 0$ to $n-1$ do
    $T \leftarrow \mathbf{R}^0$
    $\mathbf{R}^0 \leftarrow$ SHIFT($\mathbf{R}^0$)   OR   $S_{y[j]}$
    for $\ell \leftarrow 1$ to $k$ do
        $T' \leftarrow \mathbf{R}^\ell$
        $\mathbf{R}^\ell \leftarrow$ (SHIFT($\mathbf{R}^\ell$)   OR   $S_{y[j]}$)   AND
            (SHIFT(($T$   AND   $\mathbf{R}^{\ell-1}$))   AND   $T$
        $T \leftarrow T'$
    if $\mathbf{R}^k[m-1] = 0$ then
        OUTPUT($j$)
```

Fig. 3.42. Wu–Manber approximate string matching algorithm.

An example is given in Fig. 3.41.

The method, called the Wu–Manber algorithm, is implemented in Fig. 3.42. It assumes that the length of the pattern is no more than the size of the memory word of the machine, which is often the case in applications.

The preprocessing phase of the algorithm takes $O(\sigma m + km)$ memory space, and runs in time $O(\sigma m + k)$. The time complexity of its searching phase is $O(kn)$.

3.5 Bibliographic notes

The techniques described in this chapter are overused in molecular biology for comparing sequences of chains of nucleic acids (DNA or RNA) or of amino acids (proteins). The books of Deonier, Tavaré and Waterman [9], Setubal and Meidanis [28], Gusfield [10] and Böckenhauer and Bongartz [4] constitute excellent introductions to problems of the domain. The book of Sankoff and Kruskal [26] contains numerous applications of alignments. The book of Crochemore, Hancart and Lecroq [7] presents in detail, together with their correctness proofs, the algorithms for computing alignments and solving the approximate string matching problems.

The notion of longest common subsequence to two strings is used for file comparison. The command `diff` of the UNIX system implements an algorithm based on this notion by considering that the lines of the files are letters of the alphabet. Among the algorithms at the basis of this command are those of Hunt and Szymanski [14] and of Myers [20]. A general presentation of the algorithms for searching for common subsequences can be found in an article by Apostolico [1]. Wong and Chandra [32] shown that the algorithm LCS-SIMPLE is optimal in a model where we limit the access to letters to equality tests. Without this condition, Hirschberg [13] gave a (lower) bound $\Omega(n \times \log n)$. On a bounded alphabet, Masek and Paterson [18] gave an algorithm running in time $O(n^2/\log n)$. The extension of this result to the general computation of alignments is an open problem (see Apostolico and Giancarlo [2]). Using the Lempel-Ziv factorization of the two strings, Crochemore, Landau and Ziv-Ukelson designed an algorithm for computing alignments running in time $O(hn^2/\log n)$ where $h \leq 1$ is the entropy of the text.

The initial algorithm of global alignment, from Needleman and Wunsch [24], runs in cubic time. The algorithm of Wagner and Fischer [31], as well as the algorithm of local alignment of Smith and Waterman [29], run in quadratic time (see [10], page 234). The method of dynamic programming was introduced by Bellman (1957; see [6]). Sankoff [25] discusses the introduction of the dynamic programming in the processing of molecular sequences.

The algorithm LCS is from Hirschberg [12]. A generalization of this method has been proposed by Myers and Miller [21].

Charras and Lecroq created the site [5], accessible on the Web, where animations of alignment algorithms are available.

The book of Navarro and Raffinot [23] is an excellent introduction to exact and approximate string matching.

The algorithm K-DIFF-CUT-OFF is from to Ukkonen [30]. The algorithm K-DIFF-DIAG together with its implementation with the help of the computation of common ancestors was described by Landau and Vishkin [16]. Harel and Tarjan [11] presented the first algorithm running in constant time that solves the problem of the common ancestor to two nodes of a tree. An improved version is from Schieber and Vishkin [27].

Landau and Vishkin [15] conceived the algorithm K-MISMATCHES. The size of the automaton of Section 3.3 was established by Melichar [19].

The approximate pattern matching for short strings in the way of the algorithm K-DIFF-SHORT-PATTERN is from Wu and Manber [33] and also from Baeza-Yates and Gonnet [3].

A synthesis and experimental results on the approximate pattern matching is presented by Navarro [22].

References

1. A. Apostolico. String Editing and Longest Common Subsequences. G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, 1997, 361–398. Springer-Verlag.
2. A. Apostolico and R. Giancarlo. Sequence alignment in molecular biology. *J. Comput. Bio.*, 5(2):173–196, 1998.
3. R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Comm. ACM*, 35(10):74–82, 1992.
4. H.-J. Böckenhauer and D. Bongartz. *Algorithmic Aspects of Bioinformatics.* Springer-Verlag, 2007.
5. C. Charras and T. Lecroq. *Sequence Comparison.* `http://monge.univ-mlv.fr/∽lecroq/seqcomp/`
6. T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.
7. M. Crochemore, C. Hancart and T. Lecroq. *Algorithms on strings.* Cambridge University Press, 2007.
8. M. Crochemore, G. M. Landau and M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. *SIAM J. Comput.*, 32(6):1654–1673, 2003.
9. R. C. Deonier, S. Tavaré and M. S. Waterman. *Computational Genome Analysis: An Introduction (Statistics for Biology & Health).* Springer-Verlag, 2005.
10. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.
11. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
12. D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Comm. ACM*, 18(6):341–343, 1975.
13. D. S. Hirschberg. An Information-Theoretic Lower Bound for the Longest Common Subsequence Problem. *Inform. Process. Lett.*, 7(1):40–41, 1978.
14. J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Comm. ACM*, 20(5):350–353, 1977.
15. G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoret. Comput. Sci.*, 43(2–3):239–249, 1986.
16. G. M. Landau and U. Vishkin. Fast string matching with k differences. *J. Comput. System Sci.*, 37(1):63–78, 1988.
17. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Sov. Phys. Dokl.*, 6:707–710, 1966.
18. W. J. Masek and M. S. Paterson. A faster algorithm for computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–13, 1980.
19. B. Melichar. Approximate String Matching by Finite Automata. *Computer Analysis of Images and Patterns*, V. Hlavác and R. Sára, editors, Lecture Notes in Computer Science, volume 970, 342–349, Springer-Verlag, Berlin, 1995.
20. E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

21. E. W. Myers and W. Miller. Optimal alignment in linear space. *CABIOS*, 4(1):11–17, 1988.
22. G. Navarro. A guided tour to approximate string matching. *ACM Comp. Surv.*, 33(1):31–88, 2001.
23. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002.
24. S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.
25. D. Sankoff. The early introduction of dynamic programming into computational biology. *Bioinformatics*, 16:41–47, 2000.
26. D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Cambridge University Press, second edition, 1999.
27. B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.
28. J. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology.* PWS Publishing Company, 1997.
29. T. F. Smith and M. S. Waterman. Identification of common molecular sequences. *J. Mol. Biol.*, 147:195–197, 1981.
30. E. Ukkonen. Algorithms for approximate string matching. *Inform. and Control*, 64(1–3):100–118, 1985.
31. R. A. Wagner and M. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.
32. C. K. Wong and A. K. Chandra. Bounds for the string editing problem. *J. ACM*, 23(1):13–16, 1976.
33. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. ACM*, 35(10):83–91, 1992.

4

An Introductory Course on Communication Complexity

Carsten Damm

Institut für Informatik, Universität Göttingen
Lotzestr. 16-18, D-37073 Göttingen
damm@informatik.uni-goettingen.de

Summary. This text was written for a short course on communication complexity given at the 5th International PhD School in Formal Languages and Applications at Tarragona, Spain (June 5/6, 2006). The course was planned for three lectures.The text is intended to beginners and it may also serve as a guide for further reading. Some easy exercises are spread out through the text with solutions given at the end. The present version is a revision of the one used in class.

4.1 Motivation, Definition, and Background

4.1.1 Introduction

Whenever several persons, computers, or components of a system jointly complete a certain task, that none of them can do alone, there is need for communication. Sometimes, part of the actions of the participants consists of obvious information exchange between parties (Alice knows the family name, and Bob the first name of a person. By communicating they can find out the person's phone number), sometimes communication takes place invisibly and implicitely *inside* a system (bits exchanged in the processing unit of a personal computer). "How much of communication" is absolutely necessary to complete the task? If I knew this minimum "amount of communication", what does it tell me about the complexity of the task?

Communication complexity deals with questions of this type in a variety of settings. Properly defined, communication problems focus attention to the combinatorial core of an otherwise very complicated setup. Because of this concentration to the essential it allows to derive interesting computational bounds with incontrovertible rigour and is therefore at the heart of several lower bound proofs of computational complexity. Everyone interested in complexity of computations should have some ideas from communication complexity in the toolbox.

This text is organized as follows. In the current chapter we introduce some preliminaries, give an example application of communication complexity and mention some useful references. The next chapter is devoted to lower bound techniques for communication complexity. Chapter 4.3 and 4.4 deal with some problems of interest for formal language theory. In Chapter 4.5 we survey some other communication setups that are studied in the literature. Throughout the text there are exercises. Usually the exercises refer to the topic introduced immediately before. It makes therefore sense to discuss them right where they appear.

I conclude the introduction with a possible schedule for a short course on communication complexity based on the present material.

LECTURE 1 1.2 Notation and definitions, 1.3 Some "benchmark functions" for communication complexity, 1.4 An application, 1.5 Some history and some references,

LECTURE 2 2.1 The range bound and the tiling method, 2.2 The fooling set method, 2.3 The rank method, 2.4 Comparison of lower bound methods

LECTURE 3 3. Communication complexity and Chomsky hierarchy, 4. Communication complexity applications for finite automata and Turing machines, 5.1 Different modes of communication, 5.2 Different partitions, 5.3 Different games

This plan can be completed by discussing a short research paper dealing with surprising communication protocols. A very nice example is given in [4] (others include, e.g., the protocols in [12] or in [19]). Actually [4] was chosen for the PAPER session to this little course at the PhD school in Tarragona. This self-contained, two-page paper is available online (see references) and its exposition can hardly be improved. So I decided to not cover it directly in this text. Instead a set of slides from our PAPER session can be sent on email-request.

4.1.2 Notations and Definitions

First we need some general mathematical notions and notations. Throughout this text we use log to denote the base 2 logarithm. If A is a finite set, $|A|$ denotes the number of elements in it.

Exercise 1. Suppose you are about to make a catalogue of the books on your bookshelf. You decide to label the books by binary strings. If you have N books on the shelf, how many bits (binary digits) do you need for the labels? How many decimal digits do you need if you use decimal labels instead?

A pair (A_1, A_2) of subsets of A is a *partition* of A if (1) $A_1 \cup A_2 = A$ and (2) $A_1 \cap A_2 = \emptyset$. More general, a family $\{A_1, \ldots, A_T\}$ of subsets of A such that (1) $\bigcup_{i=1}^{T} A_i = A$ and (2) $A_i \cap A_j = \emptyset$ if $i \neq j$ is also called a partition of A. This is obviously equivalent to saying, that each element of A belongs to exactly one of the subsets A_i.

For vectors $\mathbf{v} \in \{0,1\}^n$ the *weight of* $\mathbf{v}$ is defined to be $\sum_{i=1}^{n} v_i$. The weight of $\mathbf{v}$ is denoted $\|\mathbf{v}\|$.

Now let's start with some simple notions from communication complexity. We want to formally describe the situation, that something is to be jointly computed by several participants, none of which has complete information on the input.

The actors in a communication setting are called *parties* or *players*. In a two-party setting, the parties are almost always called *Alice* and *Bob*. Their task is to compute a function value $f(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ is Alice's part of the input and $\mathbf{y}$ Bob's. Alice has absolutely no information about $\mathbf{y}$ and Bob has no idea, what $\mathbf{x}$ is.

f is a function of shape $f : X \times Y \to Z$, for some non-empty sets X, Y, and Z. For most examples in this text we consider $X = Y = \{0,1\}^n$, where n is fixed and $Z = \{0,1\}$. This means, that input bits are partitioned into two equally-sized sets of bits. One set is given to Alice, the other to Bob. So, Alice has an n-bit-vector $\mathbf{x}$, Bob has an n-bit-vector $\mathbf{y}$, and they want to know the bit $z = f(\mathbf{x}, \mathbf{y})$. The function as such is known to both players, but they only want to determine this particular value. The point is, that the players are not charged for computation time or memory usage, but for the number of bits they need to exchange until they both know z.

The following *trivial strategy* enables Alice and Bob to compute z: (1) When the game starts Alice sends her input $\mathbf{x}$ to Bob. (2) Bob, now knowing the complete input pair $(\mathbf{x}, \mathbf{y})$ can compute z and sends this bit to Alice. This takes $n+1$ bits of communication and it works for every function f. But, e.g., for the *parity function* PARITY : $\{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ this would be a very bad strategy. This function is defined by

$$\mathrm{PARITY}(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 & \text{, if } \|\mathbf{x}\| + \|\mathbf{y}\| \text{ is odd,} \\ 0 & \text{, else.} \end{cases}$$

PARITY$(\mathbf{x}, \mathbf{y})$ is simply the parity of *all* input bits. A much better strategy for PARITY is: (1) Alice sends the parity of her input bits. (2) Bob adds the parity of his input bits and sends the result back to Alice. This takes only 2 bits of communication.

Exercise 2. For a natural number $m \geq 1$ consider the function MOD$_k$: $\{0,1\}^n \times \{0,1\}^n \to \{0,1,\ldots,m-1\}$ defined by

$$\mathrm{MOD}_k(\mathbf{x}, \mathbf{y}) = \|\mathbf{x}\| + \|\mathbf{y}\| \pmod{k}.$$

Try to find a good communication strategy for this function. How many bits of communication are sufficient when following this strategy?

The players' aim is to communicate in such a clever way, that z can be determined without wasting too many bits of communication. For this they agree in advance on a set of rules that govern the communication and the interpretation of sent messages. This set of rules is called the *protocol*. Here is a formal definition following [21]:

Definition 1. *A* communication protocol *P over $X \times Y$ and with range Z is a binary tree where each internal node v is either labeled by a function $a_v : X \to \{0, 1\}$ or a function $b_v : Y \to Z$ and each leaf is labeled by some $z \in Z$. [Inner nodes labeled by some a_v "belong" to Alice, others "belong" to Bob.] Further, the two edges leaving a node are labeled 0 and 1, respectively.*

Execution of the protocol on input (x, y) consists of walking down the tree from the root to one of its leafs. If an internal node v is reached, the next node is the one that is reached by following the edge labeled $a_v(x)$ or $b_v(y)$ depending on which function v is labeled with. The output *of the protocol is the label of the leaf that is finally reached.*

In this definition the nodes in the tree represent the knowledge about the input pair that is common to both players. The functions a_v, b_v determine the next bit to be send by a player depending on her or his input part. It may happen that, in executing the protocol we walk from node v to node v' that both "belong to Alice". This corresponds to the fact that in this case Alice sends two consecutive bits. In the sequel we will describe protocols more conveniently by combining maximum sequences of consecutive bits sent by one and the same player to binary strings, called *messages.* Here is the corresponding terminology (giving also another definition of protocols equivalent to the one above):

1. The players take turns in communication. We consider all sent messages to be binary strings of non-zero length.
2. Each sending of a message is considered a "round" in executing the protocol. The message sent in round i is denoted m_i.
3. Alice sends the first message. Message m_i is sent by Bob, if i is even and by Alice otherwise.
4. The protocol determines in each step i the message to be sent. It depends on the sequence $m_1, \ldots, m_{i-1}$ of previous messages (which is empty in case $i = 1$) and the part of the input known to the player whose turn it is.
   The protocol also specifies when it is finished and the *output* of the protocol.

Remark 1. The sequence $m_1, \ldots, m_{i-1}$ *does not* contain more information than the sequence of edge labels from the root-to-leaf-path from Definition 1. This is due to the fact, that the labeled tree is fixed in advance and known to both players. Given such a path each player can infer from the node labels who sent which bit.

Remark 2. In general it is required that after execution of the protocol both players know the output. If the output cannot be inferred from the messages sent so far and the known input part, the last message will be the output value. This is the case in most of our examples.

Example 1. The trivial protocol for functions $f : X \times Y \to Z$ is given by:

```
$\mathsf{TRIVIAL}(f)$ :
$m_1(\mathbf{x}) := \mathbf{x}$
output $:= m_2(m_1; \mathbf{y}) := f(m_1, \mathbf{y})$
```

The “clever parity protocol” described above is given by

```
$\mathsf{PARITY}_n$ :
$m_1(\mathbf{x}) := \|\mathbf{x}\| \pmod 2$
output $:= m_2(m_1; \mathbf{y}) := m_1 + \|\mathbf{y}\| \pmod 2$
```

Let P be a certain protocol. The sequence $(m_1, m_2, \ldots, m_r)$ of messages sent while executing P on input pair $(\mathbf{x}, \mathbf{y})$ is called *transcript of P on* $(\mathbf{x}, \mathbf{y})$ and denoted $s_p(\mathbf{x}, \mathbf{y})$. Let $|m_i|$ denote the length of m_i, i.e., the number of bits in it and let $|s_P(\mathbf{x}, \mathbf{y})| = \sum_{i=1}^{r} |m_i|$ denote the total length of the transcript.

Definition 2. *The* communication complexity of the protocol P *is the number of bits exchanged by the protocol in the worst case:*

$$CC(P) := \max_{(\mathbf{x},\mathbf{y}) \in X \times Y} |s_P(\mathbf{x}, \mathbf{y})|.$$

Exercise 3. Specify the rules of the strategy from Exercise 2 as a protocol MOD_k and determine its complexity.

Let $f_P(\mathbf{x}, \mathbf{y})$ denote the output generated by following P on input pair $(\mathbf{x}, \mathbf{y})$. $f_P : X \times Y \to Z$ is the *function computed by P*.

Definition 3. *The* communication complexity of a function f *is the communication complexity of the best protocol for f:*

$$CC(f) := \min_{f_P = f} CC(P)\,.$$

Proposition 1. *For any Boolean function $\{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ holds*

$$CC(f) \le n + 1$$

Proof. $CC(\mathsf{TRIVIAL}(f)) = n + 1$.

Exercise 4. Clearly for every $f : X \times Y \to Z$ holds $CC(f) \ge 0$. Give a characterization for the set of functions, for which $CC(f) = 0$.

4.1.3 Some “Benchmark Functions” for Communication Complexity

When studying papers on communication complexity one very often meets special functions whose communication complexity is investigated. A simple function of shape $\{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ that is important in communication complexity is the following:

equality

$$\mathrm{EQ}_n(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 & \text{, if } \mathbf{x} = \mathbf{y} \\ 0 & \text{, otherwise.} \end{cases}$$

Proposition 2. $CC(\mathrm{EQ}_n) = n + 1$.

Proof. We show that protocol $\mathsf{TRIVIAL}(\mathrm{EQ}_n)$ is optimal. First observe, that for every input (x, y) both players need to send at least one bit — otherwise the output would not depend on the silent players input part contradicting the definition of the function. Suppose Alice (who starts the communication) sends less than n bits in total. Then there are two input parts $\mathbf{x}_1, \mathbf{x}_2 \in \{0, 1\}^n$, on which Alice sends the same messages. Then for all possible input parts y of Bob (whose messages depend only on $\mathbf{y}$ and received messages) the sequence of messages on (x_1, y) and (x_2, y) are the same. This means, by traveling down the protocol tree (x_1, y) and (x_2, y) reach the same leaf and in particular, the computed value will be the same. Now, suppose $\mathbf{y} = \mathbf{x}_1$. Then $\mathrm{EQ}_n(\mathbf{x}_1, \mathbf{y}) = 1$ and $\mathrm{EQ}_n(\mathbf{x}_2, \mathbf{y}) = 0$, hence the protocol cannot compute the function EQ_n.

Other functions often met are:

greater-than

$$\mathrm{GT}_n(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 & \text{, if } \mathbf{x} > \mathbf{y} \\ 0 & \text{, otherwise.} \end{cases}$$

Here, $\mathbf{x}, \mathbf{y}$ are considered as binary representations of numbers in $\{0, \ldots, 2^n - 1\}$ — likewise in the next example.

inner-product

$$\mathrm{IP}_n(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} x_i \cdot y_i \pmod 2.$$

disjointness

$$\mathrm{DISJ}_n(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 & \text{, if there is no index } i \text{ such that } x_i = y_i = 1 \\ 0 & \text{, otherwise.} \end{cases}$$

Think of $\mathbf{x}$, $\mathbf{y}$ as descriptions of sets $A, B \subseteq \{1, \ldots, n\}$ — x_i (y_i, respectively) is 1 iff i is contained in A (B, respectively). Then $\mathrm{DISJ}_n(\mathbf{x}, \mathbf{y})$ indicates, whether A and B are disjoint sets.

Sometimes, when the length n of the input is understood or is unimportant, or we speak about *all* functions of the sequence, we omit the index n and write simply EQ, GT etc.

These functions perhaps don't look too practical, and knowing about their communication complexities does not seem to be of great value, but it is! Consider the problem of computing some function value $F(\mathbf{a})$, where $\mathbf{a}$ is an N-bit input. To have something specific in mind, let $\mathbf{a}$ be n groups of n bits (interpreted as n numbers in the range 0 to $2^n - 1$) and let $F(\mathbf{a}) = \mathit{MAX}(\mathbf{a})$

be the index of the maximum of the numbers (the smallest such index in case of a draw). Then every computational device that computes MAX can also be used to compute the function GE.

Exercise 5. “Reduce” the computation of GT to the computation of MAX: Given $\mathbf{x}$ and $\mathbf{y}$, construct $\mathbf{a}$, such that the value of $\mathrm{GT}(\mathbf{x}, \mathbf{y})$ can be inferred from $MAX(a)$.

In other words, the computation of GT is a by-product of computing MAX. Thus knowing $CC(\mathrm{GT})$ gives us a truly unbeatable lower bound met by every computational device for MAX, that is “somehow charged” for communication (we’ll see an example in the next section). The point is, that MAX is more complicated and more special than GT. Therefore when deriving such lower bound for *MAX directly* there is good chance that essentials are hidden behind a bunch of peculiarities. On the other hand the bound would be a less general statement — it would apply only to exactly this function.

Further “benchmark functions” are symmetric functions:

Definition 4. *A function is called* symmetric*, if its value does not change when input positions are permuted.*

Exercise 6. Prove that the following is an equivalent definition for symmetric boolean functions:

Definition 5. *A function defined on binary inputs is called* symmetric*, if its value does only depend on the number of input bits that are equal to 1.*

Example 2. MOD_k is a symmetric function, but IP and the other examples above are not.

Exercise 7. Prove that if $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ is symmetric, then

$$CC(f) \leq \lceil \log(n+1) \rceil + 1.$$

4.1.4 An Application

This section serves to illustrate the idea from the last section, that real-life computations can be “charged for communication” in non-obvious ways. It shows also that, as mentioned in the introduction, communication can take place implicitly within a system. The exposition is taken from the beautiful survey [24].

Consider the following problem in VLSI-design. A “microchip” is to be layed-out. Its purpose is to compute $\mathrm{EQ}_n(\mathbf{x}, \mathbf{y})$. Two groups of n bits each $\mathbf{x} = (x_1, x_2, \ldots, x_n), \mathbf{y} = (y_1, y_2, \ldots, y_n)$, arrive simultaneously at $2n$ places (the “input processors”) arranged in exactly the given bit order along one edge of the layout. There is also an “output processor” — some place at the border that produces the required output bit 1. There may be more processors as well as connecting wires between them on the chip.

Considering the processors and wires as nodes and edges of a graph, a VLSI-design can be considered a special embedding of that graph into the plane with some geometric restrictions. It is embedded into a grid of squares each of which is either empty or contains exactly one of the following as shown in Figure 4.1: a processor, a straight line connecting two opposite sides of the square (in two orientations), a sharp bend connecting to adjacent sides of the square (in 4 orientations) or a crossing of two lines, that each connect two opposite sides of the square.

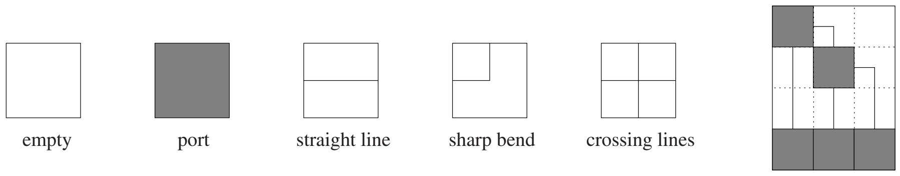


Fig. 4.1. Squares in an VLSI-design and an example layout

The chip works in distinct time steps. At each step, each processor reads the bits sent to it, computes some bit for some of its connecting wires and sends it to the other end of this wire. It is essential to note, that we assume "fast wires", i.e., the time that a signal needs to travel from one to the other end of a wire is considered to be a non-zero constant independent from its length.

The smallest possible layout contains only the input processors lined up in a row, one of them serving also as output processor. Let $p_1, \ldots\ p_{2n}$ be the nodes, each containing a processor and the have wires to read in the input (see figure 4.2). We describe the functioning of this layout. In the first time step all nodes except p_{n+1} "sleep". In this step p_{n+1} reads y_1, sends it to the left and sends an "alarm bit" to the right. The alarm bit wakes up p_{n+2}, who sends y_2 to the left and alarms its right neighbor, and so on. Each processor, that receives some bit from the right, will transmit it to the left in the next step.

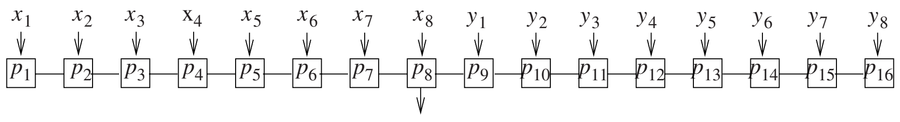


Fig. 4.2. Linear VLSI layout

When p_1 eventually receives y_1 it compares it with x_1 and sends bit 1 to the right if they are equal and bit 0 if they are not. p_2 receives this bit simultaneously with y_2. It sends bit 1 to the right, if it received 1 from the left and x_2 and y_2 are the same, otherwise it sends bit 0. Finally, p_n receives one bit from the left (the resulting bit from all previous comparisons) and the bit y_n from the right. If the bit from the left is 1 and x_n and y_n are equal, then p_n gives the overall output bit 1, else 0.

This design is small, but computation takes n time steps, since each wire can only carry one bit at a time. With a more generous design, the chip would finish work earlier. The idea is to provide extra wires that pass x_1 and y_1, x_2 and y_2, ..., x_n and y_n *at the same time* to special comparator nodes and to collect the outcomes of comparison in a binary tree design to give the final output value (see Figure 4.3).

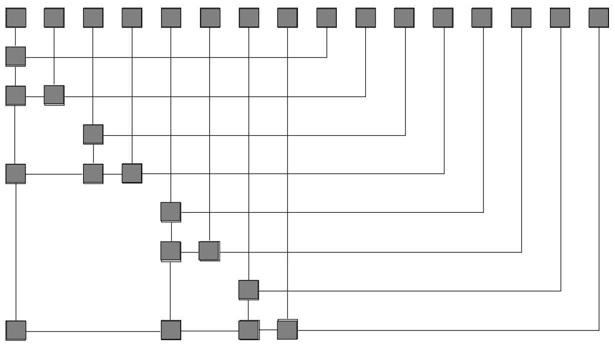


Fig. 4.3. Tree-like VLSI layout

In this design $\mathcal{O}(\log n)$ time steps are sufficient to complete the task. By using bends in the tree part of the layout it is possible to use a little less of the chip's area. However, this would not reduce the total length of the wires. Since every unit of the wire occupies some place on the chip (a square), it makes sense to consider the total length of wires as occupied chip area. Adopting this point of view our layout uses more than an amount of n^2 of area (the right half of the design). Combining ideas from the small-but-slow linear and the fast-but-big tree-like design, it is possible to make a design that still works in time $\mathcal{O}(\log n)$ but uses area only $\mathcal{O}(n^2/\log n)$. However, still, the product of area and time is at least n^2. And this is true for *any* VLSI-design! This is a special case of results of [31], which build on communication complexity ideas:

Proposition 3. *If a VLSI-chip computes* EQ_n*, then for the number* T *of time steps needed and the total number* A *of wire units satisfies*

$$AT \geq n^2.$$

Proof. Divide the design into two pieces by a vertical line, so that on the left hand side we have inputs $x_1, \ldots, x_n$, on the right hand side we have $y_1, \ldots, y_n$. We concentrate on the communication that crosses the line. Since by Proposition 2 we have $CC(\mathrm{EQ}_n) = n+1$ at least n bits must cross the line until the result is known. Why not $n+1$? Since only the output node (sitting on one of the sides) needs the final result, it is not necessary to "inform" the other. Because there are only T time steps, in one of the steps at least n/T bits cross the line. But as the wires carry only one bit at a time, this means at least n/T wires cross this line.

How many wires do cross the line, if its position were one place to the right? In this case x_1 and y_1 were on the same side, but still communication is needed to compute $\mathrm{EQ}_{n-1}(x_2, \ldots, x_n), (y_2, \ldots, y_n))$. By the same argument we conclude, that at least $(n-1)/T$ wires cross this line. Now we shift one position further to the right, and as before we can conclude that at least $(n-2)/T$ wires cross the line, and so on. Similarly we can also shift the line by 1, 2, ... places to the left and the line will cross at least $(n-1)/T, (n-2)/T, \ldots$ wires.

In summing up, our lines crossed at least

$$\frac{n}{T} + 2\frac{n-1}{T} + \frac{n-2}{T} \ldots + 2\frac{1}{T} = \frac{n^2}{T}$$

wires, which proves the statement.

4.1.5 Some History and Some References

Communication complexity arguments where first applied in the late 1970s [1, 32]. The result from the last section is from [31]. As the technique became known widely, more and more results in a variety of areas where based on it or techniques were formulated in the language of communication complexity. In particular, also the model was extended to more sophisticated situations: more players, non-determinism, randomization, different charging, other input partitions etc.

The primary motivation for communication complexity study comes from applications in the field of distributed and parallel computing (including VLSI-computing). However, communication complexity is a neat field of study on its own. There are even analogies to NP-completeness theory and structural complexity. There is one difference however to "classical" complexity theory: in communication complexity we can *prove* the separations, that we only *conjecture* in the classical setting. Unfortunately, there seems no way to carry things over... This line of research was begun by [29] and continued by, e.g., [3, 13, 18, 23, 8] (a random selection). One of the latest developments is quantum communication complexity [9]. The achievements of communication complexity theory feed back into other fields of application and theory.

Nowadays, communication complexity is an established technique that is characterized by strict pinpointing combinatorial situations that appear in computations of all kind, beautiful mathematics, appealing puzzles, and surprising results. This little tutorial can only give a glimpse on some of the basic ideas in this nice toolbox. There are some very good surveys on communication complexity around, that cover more or different material: [28, 24, 14]. There are also books devoted to the subject of communication complexity: [21, 15].

4.2 Some Lower Bound Methods and Results

4.2.1 The Range Bound and the Tiling Method

Definition 6. *Let $f : X \times Y \to Z$ be a function. The* range *of f is the set of all values, that can be taken by the function:*

$$\mathrm{Range}(f) := \{z \in Z | \exists(\mathbf{x}, \mathbf{y}) \in X \times Y : f(\mathbf{x}, \mathbf{y}) = z\}.$$

Proposition 4.

$$CC(f) \geq \log |\mathrm{Range}(f)|.$$

Proof. If $f(\mathbf{x}_1, \mathbf{y}_1) \neq f(\mathbf{x}_2, \mathbf{y}_2)$, then the players must use different transcripts on $(\mathbf{x}_1, \mathbf{y}_1)$ and $(\mathbf{x}_2, \mathbf{y}_2)$ — otherwise the protocol would make an error. Hence, we must have at least $|\mathrm{Range}(f)|$ many different transcripts, which are binary strings of length at least $CC(f)$. By Exercise 1 $CC(f)$ must be at least $\log |\mathrm{Range}(f)|$ to give these many strings.

Exercise 8. Apply Proposition 4 to the function MOD_k.

For functions $f : X \times Y \to \{0, 1\}$ with boolean output Proposition 4 is not very helpful, so we try to refine the argument.

Before going into details it is useful to have the following picture in mind: We regard f as a matrix of order $|X| \times |Y|$ with rows indexed by inputs $\mathbf{x}$ (Alice's part) and columns indexed by inputs $\mathbf{y}$ (Bob's part). Consequently, the entry in row $\mathbf{x}$ and column $\mathbf{y}$ is $f(\mathbf{x}, \mathbf{y})$. We refer to this matrix as the *communication matrix* of f and denote it by M_f — however, as said, it is nothing else than f itself, written down in a special manner.

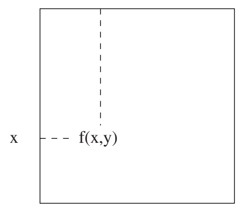


Fig. 4.4. The communication matrix of f

Exercise 9. Write down the communication matrix of PARITY : $\{0,1\}^3 \times \{0,1\}^3 \to \{0,1\}$. (After writing the first few bits in a row and in a column you will quickly see the structure and can stop writing ...)

Definition 7. *A subset $R \subseteq X \times Y$ is called a* rectangle *in $X \times Y$, if $R = A \times B$ for some subsets $A \subseteq X$ and $B \subseteq Y$.*

Please note, that it is *not* required, that A and B are consecutive in any sense! This requirement would even be meaningless, without specifying the order of entries in the communication matrix.

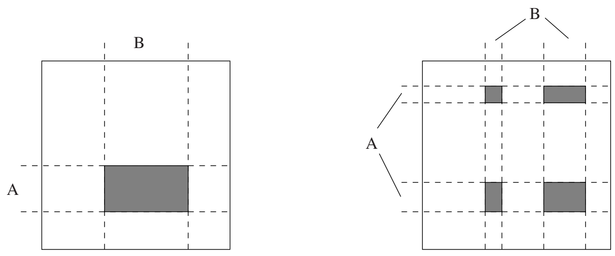


Fig. 4.5. Two examples for rectangles

Lemma 1 (Combinatorial Characterization of Rectangles). *Let $R \subseteq X \times Y$. The following are equivalent:*

1. *R is a rectangle.*
2. *For any two points $(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2) \in R$ holds $(\mathbf{x}_1, \mathbf{y}_2) \in R$.*
3. *$R = R_X \times R_Y$, where $R_X = \{\mathbf{x} | \exists \mathbf{y} : (\mathbf{x}, \mathbf{y}) \in R\}$ and $R_Y = \{\mathbf{y} | \exists \mathbf{x} : (\mathbf{x}, \mathbf{y}) \in R\}$.*

Proof. 3. $\Rightarrow$ 1. is obvious.

1. $\Rightarrow$ 2. Let $R = A \times B$. Since $(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2) \in R$ we know on the one hand side, that $\mathbf{x}_1 \in A$ and on the other hand side, that $\mathbf{y}_2 \in B$. Hence, $(\mathbf{x}_1, \mathbf{y}_2) \in A \times B = R$.

2. $\Rightarrow$ 3. We show (1) $R \subseteq R_X \times R_Y$ and (2) $R_X \times R_Y \subseteq R$.
(1) Let $(\mathbf{x}, \mathbf{y}) \in R$. Clearly, $\mathbf{x} \in R_X$ (since there is an $\mathbf{y}$, such that $(\mathbf{x}, \mathbf{y}) \in R$). Similarly we have $\mathbf{y} \in R_Y$. Together we obtain $(\mathbf{x}, \mathbf{y}) \in R_X \times R_Y$.
(2) Let $\mathbf{x}_1 \in R_X$ and $\mathbf{y}_2 \in R_Y$. By construction of R_X, R_Y this means, there are $\mathbf{y}_1 \in Y$ and $\mathbf{x}_2 \in X$, such that $(\mathbf{x}_1, \mathbf{y}_1) \in R$ and $(\mathbf{x}_2, \mathbf{y}_2) \in R$. By the assumption we can conclude $(\mathbf{x}_1, \mathbf{y}_2) \in R$.

Let P be a protocol computing f and let $\alpha = (m_1, \ldots, m_r)$ be a transcript between Alice and Bob following this protocol. (Reminder: r depends on the input pair.) We denote the set of input pairs on which the protocols transcript is α as follows:

$$R_\alpha = \{(\mathbf{x}, \mathbf{y}) | s_P(\mathbf{x}, \mathbf{y}) = \alpha\}.$$

Fact 1. *$R_\alpha \subseteq X \times Y$ is a rectangle.*

Proof. We use induction and seemingly show a little bit more: The set R_{α_i} of input pairs whose transcript *only starts* with $\alpha_i = (m_1, \ldots, m_i)$ is a rectangle.

For $i = 0$ (no message sent) $R_{\alpha_0} = X \times Y$, which is a rectangle. For $i = 1$ (Alice sent her first message) $R_{\alpha_1} = A \times Y$ for some $A \subseteq X$, since m_1 depends only on Alice's input $\mathbf{x}$ (A is simply the set of those $\mathbf{x}$, for which Alice would send m_1).

Suppose R_{α_i} is a rectangle $A \times B$ for some $i \geq 1$. Without loss of generality we assume, that the next message is to be sent by Alice. By definition of the protocol this message depends only on $\mathbf{x}$ and the previous messages. Let A' be the subset of A, on which Alice, given previous messages $m_1, \ldots, m_i$, sends m_{i+1}. Then $R_{\alpha_{i+1}} = A' \times B$ which again is a rectangle.

Let's think about the induction argument for a second. It says that after every round the input pairs, that are still "in the game" form a set of rectangular shape. That's interesting, isn't it?

Fact 2. *$f(\mathbf{x}, \mathbf{y})$ is the same for every input pair $(\mathbf{x}, \mathbf{y}) \in R_\alpha$.*

Proof. Follows directly from the definition of the output.

Fact 3. *The family of sets $R_P = \{R_\alpha | \alpha$ is a transcript of P on some input pair$\}$ is a partition of $X \times Y$. This partition is called the* protocol partition *of P.*

Proof. Input pair $(\mathbf{x}, \mathbf{y}) \in X \times Y$ belongs to $R_{s_P(\mathbf{x},\mathbf{y})}$ but to no other of the sets R_α.

Definition 8. *Let $f : X \times Y \to Z$. We consider f like a* coloring *of the entries of $X \times Y$. A rectangle $R \subseteq X \times Y$ is called f-*monochromatic *if f is constant on R. If f is understood, we sometimes simply speak of* monochromatic *rectangles. Especially, for $z \in Z$ an f-monochromatic rectangle $R \subseteq X \times Y$ is called a z-*rectangle *if $f(\mathbf{x}, \mathbf{y}) = z$ for all $(\mathbf{x}, \mathbf{y}) \in R$.*

We consider the partition number of f, *which is the smallest number of f-monochromatic rectangles in a partition of $X \times Y$:*

$$Cov^D(f) := \min\{T | \exists \text{ partition of } X \times Y \text{ into } T \text{ monochromatic rectangles}\}.$$

Remark 3. The superscript D in $Cov^D(f)$ reminds to "disjoint" — we want a partition, not just a covering by monochromatic rectangles.

Example 3. Recall how communication matrices of PARITY-functions look like (see Exercise 9). What is the partition number of such functions?

Let's first look at our PARITY$_3$-example from the exercise. Since order is not an issue for the rectangles we have in mind, we order the rows and columns such that first we have all n-vectors with even parity, and then we have all n-vectors with odd parity. Then we fill in the matrix entries. As the example shows, this gives 4 rectangles:

$$\begin{matrix}0&0&0&0&1&1&1&1\\0&0&0&0&1&1&1&1\\0&0&0&0&1&1&1&1\\0&0&0&0&1&1&1&1\\1&1&1&1&0&0&0&0\\1&1&1&1&0&0&0&0\\1&1&1&1&0&0&0&0\\1&1&1&1&0&0&0&0\end{matrix}.$$

It is obvious, that this holds in general: Entries of type even-even are 0-entries, as well as odd-odd-type entries. The 1-entries are covered by an even-odd and an odd-even rectangle. Hence the partition number of a PARITY-function is at most 4. Can it be smaller?

If it was smaller, then we could combine, say, the 0-rectangles into one (the argument for 1-rectangles is similar). So lets take an even-even entry $(\mathbf{x}_1, \mathbf{y}_1)$ and an odd-odd entry $(\mathbf{x}_2, \mathbf{y}_2)$ and assume they are in the same monochromatic rectangle. But then, by the Characterization Lemma 1 also $(\mathbf{x}_1, \mathbf{y}_2)$ belongs to this rectangle. But this entry is of even-odd type, which is a contradiction.

Hence $Cov^D(f) = 4$ for all n.

Proposition 5. *For every function $f : X \times Y \to Z$ holds*

$$CC(f) \geq \log Cov^D(f).$$

Proof. By the above mentioned facts the protocol partition of P is a partition of $X \times Y$ into monochromatic rectangles. If P is an optimal protocol, on every input pair at most $CC(f)$ bits are exchanged. Therefore the number of possible transcripts (and therefore the number of rectangles in the particular partition induced by P) is at most $2^{CC(f)}$. Hence we obtain $2^{CC(f)} \geq Cov^D(f)$.

For later referencing we call this lower bound argument the *tiling method*. Since the partition number of PARITY-functions is 4, we can conclude, that at least 2 bits have to be exchanged to jointly determine $\text{PARITY}_n(\mathbf{x}, \mathbf{y})$, hence the protocol PARITY_n is optimal. Well, no big surprise ...

Example 4. Also no big surprise, but hopefully instructive: We reprove Proposition 2 $CC(\text{EQ}_n) = n + 1$. But now we use partition numbers.

An EQ_n-monochromatic partition clearly needs to have 2^n 1-rectangles to cover all 1-entries (by an argument as in Example 3 we see, that no two of the 1's can live in the same monochromatic rectangle.).

On the other hand, we need at least one 0-rectangle to cover the 0-entries of the communication matrix. This means $C(f) > 2^n$, from which we conclude $CC(\text{EQ}_n) > n$ by the help of Proposition 5. But since $CC(\text{EQ}_n)$ is an integer number, it is at least $n + 1$.

Exercise 10. As the examples might suggest, it is tempting to believe, that each partition of the communication matrix of f into monochromatic rectangles there is indeed already a protocol partition. However, this is not the case. Can you find an example function of shape $\{1, 2, 3\} \times \{1, 2, 3\} \to \{0, 1\}$ and partition into monochromatic rectangles, that is *not* a protocol partition?

Remark 4. It is still open, whether for all $f : \{0, 1\}^n \times \{0, 1\}^n \to \{0, 1\}$ holds $CC(f) = \mathcal{O}(Cov^D(f))$. This question was posed in [25].

4.2.2 The Fooling Set Method

It is sometimes hard to prove lower bounds on monochromatic partition numbers. The following method is often easier to apply. It was first used in [32] and in a more elaborated form in [22].

Let $f : X \times Y \to Z$.

Definition 9. *A set of input pairs* $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_\ell, \mathbf{y}_\ell)\}$ *is called a* fooling set *for* f*, if there exists some* $z \in Z$*, such that*

1. *for all* i*,* $f(\mathbf{x}_i, \mathbf{y}_i) = z$*,*
2. *for all* $i \neq j$*, either* $f(\mathbf{x}_i, \mathbf{y}_j) \neq z$ *or* $f(\mathbf{x}_j, \mathbf{y}_i) \neq z$*.*

For fixed z *the set is called* z-fooling set*.*

Proposition 6. *If* f *has a fooling set of size* ℓ*, then*

$$CC(f) \geq \log \ell.$$

Proof. By Proposition 5 it is sufficient to prove $Cov^D(f) \leq \ell$.

Suppose the opposite is true, i.e., suppose there is a partition of $X \times Y$ into less than ℓ monochromatic rectangles. Then there exist two pairs $(\mathbf{x}_i, \mathbf{y}_i), (\mathbf{x}_j, \mathbf{y}_j)$ in the fooling set, that are in the same rectangle $A \times B$. By definition z is the "color" that f gives to this rectangle. By the Characterization Lemma 1 we know that also $(\mathbf{x}_i, \mathbf{y}_j)$ and $(\mathbf{x}_j, \mathbf{y}_i)$ belong to this rectangle. But by definition of the fooling set, one of those pairs has a different color that z, which is a contradiction. Hence, there must be at least ℓ rectangles in any f-monochromatic partition of $X \times Y$.

Remark 5. The proof considers only rectangles colored z and therefore in fact we get a lower bound on the number of z-rectangles. By lower-bounding the number of rectangles of any color $z \in Z$ and summing these bounds up, we can improve the bound: Let $Z = \{z_1, \ldots, z_t\}$ and for $i = 1, \ldots, t$ let s_i be the size of some z_i-fooling set for f, then $CC(f) \leq \lceil \log(s_1 + \ldots + s_t) \rceil$. We make use of this argument in the following example.

Example 5. We apply the bound to the disjointness function. Recall the interpretation: n-bit vectors $\mathbf{x}$ are considered as "encoding" subsets A from $\{1, \ldots, n\}$: $x_i = 1$ iff $i \in A$.

We claim, that the following is a 1-fooling set for DISJ_n:

$$\{(A, A^c) \mid A \subseteq \{1, \ldots, n\}\},$$

where A^c denotes the complement of A in $\{1, 2, \ldots, n\}$. Indeed, $\mathrm{DISJ}_n(A, A^c) = 1$ for all A and on the other hand for $A \neq B$ either $A \cap B \neq \emptyset$ or $A^c \cap B^c \neq \emptyset$. Hence we conclude $CC(\mathrm{DISJ}_n) \geq n$.

But this concerns only 1-rectangles. Since the function is non-constant, there must be at least one 0-rectangle. Hence, $CC(\mathrm{DISJ}_n) \geq \lceil \log(2^n + 1) \rceil = n + 1$.

Exercise 11. Use the fooling-set bound to show that $CC(\mathrm{GT}_n) = n + 1$.

Remark 6. The communication argument from Section 4.1.4 on the area-time-product of VLSI-circuits is easily seen to extend to functions like GE, GT and so on.

4.2.3 The Rank Method

Recall that M_f denotes the communication matrix of f, i.e., the matrix with rows and columns indexed with inputs $\mathbf{x} \in X$ and $\mathbf{y} \in Y$ and entries $f(\mathbf{x}, \mathbf{y})$.

The following bound is due to [26]

Proposition 7. *Let* $f : X \times Y \to \{0, 1\}$ *and let* $\mathrm{rank}(M)$ *denote the rank of a matrix over the rationals. Then*

$$CC(f) \geq \log \mathrm{rank}(M_f).$$

Proof. Consider a partition of $X \times Y$ into $Cov^D(f)$ rectangles that are f-monochromatic. Let $R_1, \ldots, R_t \subseteq X \times Y$ be the 1-rectangles in this partition. By the tiling bound it is sufficient to prove $\mathrm{rank}(M_f) \geq t$. We associate the following matrices of order $|X| \times |Y|$ to them:

$$M_i(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 & \text{, if } (\mathbf{x}, \mathbf{y}) \in R_i. \\ 0 & \text{, else.} \end{cases}$$

Observe, that all non-zero rows of M_i are the same, hence $\mathrm{rank}(M_i) = 1$ for all i. Since the R_i do not intersect, we have $M_f = \sum_{i=1}^{t} M_i$. By the properties of the rank we conclude

$$\mathrm{rank}(M_f) \leq \sum_{i=1}^{t} \mathrm{rank}(M_i) = t.$$

There is good chance to successfully apply the rank lower bound in cases, where the communication matrix features some algebraic property.

Example 6. Now we can prove a lower bound on the inner product function IP:

$$CC(\mathrm{IP}_n) \geq n.$$

To this end, let's have a look on the communication matrices of $\mathrm{IP}_1, \mathrm{IP}_2, \mathrm{IP}_3, \ldots$. We use the natural binary ordering on rows and columns (e.g., in case $n = 2$ the order is 00, 01, 10, 11). Then the resulting matrices are:

$$M_{\mathrm{IP}_1} = \begin{pmatrix} 0\,0 \\ 0\,1 \end{pmatrix}, \; M_{\mathrm{IP}_2} = \begin{pmatrix} 0\,0\;0\,0 \\ 0\,1\;0\,1 \\ 0\,0\;1\,1 \\ 0\,1\;1\,0 \end{pmatrix}, \; M_{\mathrm{IP}_3} = \begin{pmatrix} 0\,0\,0\,0\;0\,0\,0\,0 \\ 0\,1\,0\,1\;0\,1\,0\,1 \\ 0\,0\,1\,1\;0\,0\,1\,1 \\ 0\,1\,1\,0\;0\,1\,1\,0 \\ 0\,0\,0\,0\;1\,1\,1\,1 \\ 0\,1\,0\,1\;1\,0\,1\,0 \\ 0\,0\,1\,1\;1\,1\,0\,0 \\ 0\,1\,1\,0\;1\,0\,0\,1 \end{pmatrix}, \ldots$$

Observe that except the first one, each matrix contains copies of its predecessor matrix in the left-upper, left-lower, and in the right-upper corner and a copy of the complement (0–1 exchanged) in the right-lower corner. These matrices are known as *Sylvester-matrices* and, their rank, e.g., over the rationals is one less than full-rank. To see this, consider entry $(\mathbf{x}, \mathbf{z})$ of $(M_{\mathrm{IP}_n})^2$. By definition this entry equals $\sum_{\mathbf{z} \in \{0,1\}^n} \mathrm{IP}_n(\mathbf{x}, \mathbf{z})\mathrm{IP}_n(\mathbf{z}, \mathbf{y})$. This is the number of $\mathbf{z} \in \{0,1\}^n$ for which $\mathrm{IP}_n(\mathbf{x}, \mathbf{z}) = \mathrm{IP}_n(\mathbf{z}, \mathbf{y}) = 1$. This is an inhomogeneous linear system of equations modulo 2 in n indeterminates $(z_1, \ldots, z_n)$. In case $\mathbf{x} \neq \mathbf{0} \neq \mathbf{y}$ the number of solutions is 2^{n-1} if $\mathbf{x} = \mathbf{y}$ and 2^{n-2} else. If one of $\mathbf{x}, \mathbf{y}$ is identically zero the number of solutions is zero. We can conclude that $\mathrm{rank}(M_{\mathrm{IP}_n}) = 2^n - 1$ and Proposition 7 gives $CC(\mathrm{IP}_n) \geq n$.

Exercise 12. Apply Proposition 7 to the function GT_n.

4.2.4 Comparison of Lower Bound Methods

The fooling-set method and the rank-method rely on the tiling bound, therefore it is the potentially strongest longer bound method for communication complexity. However, direct application of the tiling-bound is difficult.

It is interesting, to compare the relative power of lower bound methods on communication complexity. This study has been started in [2] and was continued in [13] and [10]. We report here some of the results from such study. For ease of exposition we adopt the following notations from [10], that all refer to a function of shape $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$:

- $\mathrm{r}(f) = \log \mathrm{rank}(M_f)$,
- $\mathrm{fs}(f) = \log \ell$, where ℓ is the maximum size of a 1-fooling set for f,
- $\mathrm{t}(f) = \log Cov^D(f)$.

Remark 7. Since in any case $\operatorname{rank}(M_f) \leq 2^n$, the rank lower bound can give no larger bound than n. On the other hand, the fooling set lower bound accounting for also 0-rectangles can give the optimal bound $n+1$ (see Example 5). The reason for this somehow annoying difference is simple: An all-0-row is a 0-rectangle and would contribute to the fooling-set bound. However, its rank is 0 — so it would not contribute to the rank lower bound.

Hence, it is only fair in a comparison, to restrict consideration to 1-fooling sets only.

The following general inequalities are known:

- $\mathrm{t}(f) \leq CC(f) \leq (\mathrm{t}(f)+1)^2$
- $\mathrm{r}(f) \leq \mathrm{t}(f)$, $\mathrm{fs}(f) \leq \mathrm{t}(f)$ (see proofs above)
- If n is sufficiently large, there are example functions f, such that $CC(f) = n$ but $\mathrm{fs}(f) = \mathcal{O}(\log n)$ — the fooling-set bound may be very weak.

Concerning the comparison of fs and r the following more detailed facts were shown

1. For almost all $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ holds $\mathrm{r}(f) = n$ and simultaneously $\mathrm{fs}(f) = \mathcal{O}(\log n)$. Explicit constructions of such functions are known.
2. For all $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ holds $\mathrm{fs}(f) \leq 2\mathrm{r}(f)$ and explicit constructions for f are known, for which $\mathrm{fs}(f) = n$ but $\mathrm{r}(f) < 0.8n$.

Let us comment on the first of these results: What does "for almost all" mean? It says, that if a function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ is taken at random from the uniform distribution on the set of all such functions, then the probability that is has the mentioned properties tends to 1 as n grows to infinity.

Exercise 13. For each $k \in \{0,1,\ldots,2n\}$ let $\mathrm{EXACT}_k : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ be defined by

$$\mathrm{EXACT}_k(\mathbf{x},\mathbf{y}) = \begin{cases} 1 & \text{, if } \|\mathbf{x}\| + \|\mathbf{y}\| = k \\ 0 & \text{, else.} \end{cases}$$

Show that $CC(\mathrm{EXACT}_k) \geq \lceil \log(k+1) \rceil$. For the proof you can use any of the mentioned lower bound methods.

Exercise 14. We know that symmetric functions $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ have communication complexity bounded by $\mathcal{O}(\log n)$. Give an example of such a function with $CC(f) \geq \lceil \log n \rceil$.

Remark 8. Taking into account that there must exist at least one 0-rectangle one can improve the result of Exercise 13 to

$$CC(\mathrm{EXACT}_k) \geq \lceil \log(k+2) \rceil = \lceil \log((k+1)+1) \rceil.$$

Exercise 15. If the range of the function under consideration is greater than $\{0, 1\}$, the simple idea from the last remark can be extended. E.g., it can be shown that $CC(\text{MOD}_k) \geq \lceil \log(2k-1) \rceil = \lceil \log(k + (k-1)) \rceil$, by presenting a 0-fooling set for and by taking into account, that there must be at least one 1-rectangle, at least one 2-rectangle, ..., at least one $(k-1)$-rectangle. Try this!

But for this function it is easy to give better lower bounds on the number of those z-rectangles. This way one can prove $CC(\text{MOD}_k) \geq \lceil 2 \log k \rceil$, which is your exercise.

4.3 Communication Complexity and Chomsky Hierarchy

It is natural to ask, how communication complexity relates to other complexity measures or to restricting features of computational devices. We saw already one example concerning complexity measures in the field of VLSI-design. Next we concentrate on the Chomsky hierarchy. We take some ideas from the exposition in [15].

Recall the Chomsky hierarchy:

$$\text{REG} \subset \text{CFL} \subset \text{CS} \subset \text{RE} \subset \text{ALL},$$

where the notations in this order denote the classes of regular, context-free, context-sensitive, recursively enumerable, and all languages. (As usual, by a *language* we mean a subset of Σ^* for some fixed finite alphabet Σ.)

To bring these two concepts in touch we first need to translate functions (studied in communication complexity) into languages (studied in formal language theory) and vice versa. We confine to Boolean functions and to languages over the alphabet $\{0, 1\}$. The key idea is to bring *sequences of functions* in correspondence to languages. Here is how:

Definition 10. *For every natural number N let $f_N : \{0,1\}^N \to \{0,1\}$ be a Boolean function. We denote by f the sequence $f_1, f_2, \ldots$.* The language defined by f *is the set*

$$L_f = \{\mathbf{w} \in \{0,1\}^* | f_{|\mathbf{w}|}(\mathbf{w}) = 1\}$$

($|\mathbf{w}|$, as usual, denotes the length of the string $\mathbf{w}$).

If $L \subseteq \{0,1\}^n$, then the Boolean function sequence f_L defined by L *is the sequence $f_{L,N} : \{0,1\}^N :\to \{0,1\}, N = 1, 2, \ldots$ with*

$$\forall \mathbf{w} \in \{0,1\}^N : f_{L,N}(\mathbf{w}) = 1 \Leftrightarrow \mathbf{w} \in L.$$

We want to apply communication complexity ideas to member functions from sequences f_L for arbitrary L. In the Boolean function examples studied so far, the input was always partitioned in *equal sized* parts and distributed to Alice and Bob. This is not possible for functions $f_{L,N}$, if N is odd. But for these cases we simply consider the input space as product of $X = \{0,1\}^{\lfloor N/2 \rfloor}$ and $Y = \{0,1\}^{\lfloor N/2 \rfloor + 1}$ — so we give the first $\lfloor N/2 \rfloor$ bits to Alice (this input part is denoted $\mathbf{x}$) and the remaining bits to Bob (this input part is denoted $\mathbf{y}$).

This said, we can now speak about the *communication complexity of the language L* by considering communication complexities of the member functions in the corresponding Boolean function sequence: $CC(f_{L,N})$.

For any $g : \mathbb{N} \to \mathbb{N}$ let $\mathrm{CC}(g(N))$ denote the set of languages $L \subseteq \{0,1\}^*$, such that $\forall N : CC(f_{L,N}) \leq g(N)$.

We start with the top of the Chomsky hierarchy. The result of the following exercise seems disappointing at first glance, but at least it is instructive. We'll comment on it afterwards.

Exercise 16. First recall the solution to Exercise 4. Then use a diagonalization argument to prove the following statement:

Proposition 8. *There is a language $L \subseteq \{0,1\}^*$ which is not recursively enumerable but has zero communication complexity:*

$$L \in \mathrm{CC}(0) \setminus \mathrm{RE}.$$

Remark 9. The result is not really surprising. Each Turing machine is a finite object, processing every input *uniformly* by the same algorithm. Proposition 8 shows, that it is inadequate to compare a uniform computational mechanism (acceptors for languages in the Chomsky hierarchy) to *non-uniform* ones, like infinite sequences of communication protocols, that provide for each input length an own algorithm.

There is a standard way to translate uniform devices to corresponding non-uniform ones (see [20]), that we introduce shortly: A computational device is called *non-uniform* if there is some infinite sequence $\alpha_1, \alpha_2, \ldots$ called *advice* that is used like an oracle mechanism: Instead of $\mathbf{x}$ the device processes the combination $\mathbf{x}\#\alpha_{|\mathbf{x}|}$. If $\mathcal{C}$ is a certain complexity class, defined by a resource bounded uniform computational device, then $\mathcal{C}/g(N)$ denotes the class of languages accepted by the same resources but with advice of length at most $g(N)$ for inputs of length N. To define this formally we introduce the following notation:

$$\mathcal{C}/g(N) = \{L : \alpha | L \in \mathcal{C}, \alpha = (\alpha_N)_{N \in \mathbb{N}} : \forall N \, |\alpha_N| \leq g(N)\}$$

where

$$L : \alpha = \{w | w\#\alpha_{|w|} \in L\}\}.$$

The proposition shows, that in order to accept the language L', a Turing machine acceptor needed the information *which* constant function is computed at inputs of length 1, length 2, ... This is only one bit of advice, or — in the terminology of [20] — $\mathrm{CC}(0) \subseteq \mathrm{RE}/1$. In fact, already a finite automaton equipped with this advice could accept L'. It is straight-forward to extend the advice mechanism to finite automata (see [6]). Hence, the following is true:

$$\mathrm{CC}(0) \subseteq \mathrm{REG}/1.$$

Proposition 9. *For each regular language $L \subseteq \{0,1\}^*$ there exists a constant k, such that $L \in \mathrm{CC}(k)$.*

Proof. See Chapter 4.4.

So, for every regular language a constant number of communicated bits is sufficient to decide membership in that language. We express this fact as

$$\mathrm{REG} \subseteq \mathrm{CC}(\mathcal{O}(1)).$$

On the other hand in [15] it is shown, that no single constant number of communicated bits is sufficient to decide about *any* regular language:

Proposition 10. *For every natural number k, there is a regular language L_k outside* $\mathrm{CC}(k)$.

Proof. Given k consider $L_k = \{\mathbf{w} \in \{0,1\}^* : \|\mathbf{w}\| = 2^{k+1}\} \in \mathrm{REG}$. By Exercise 13 any deterministic communication protocol needs at least $k+1$ bits of communication to decide $\mathbf{w} \in L_k$.

Remark 10. Let *const* (*poly*, respectively) denote the class of advice sequences $\alpha_1, \alpha_2, \ldots \in \{0,1\}^*$ such that $\forall n : |\alpha_n| \leq k$ ($\forall n : |\alpha_n| \leq n^k$, respectively) for some constant k. Using notation and ideas from Remark 9, the proof of Proposition 9 and [6] one can show

$$\mathrm{CC}(\mathcal{O}(1)) \supset \mathrm{REG}/\mathit{const}.$$

However, providing more than constant length advice does not help (see [6]):

$$\mathrm{REG}/\mathit{const} = \mathrm{REG}/\mathit{poly}.$$

How about context-free languages? Already in this language class there are languages requiring the highest possible — namely linear — communication complexity.

Proposition 11. *There is a context-free language $L \subseteq \{0,1\}^*$, such that $L \notin$* $\mathrm{CC}(o(N))$.

Proof (sketch). It is easy to show, that the set of palindromes, which is a context free language, has maximum communication complexity.

Remark 11. Palindromes are hard to recognize in our basic two party communication model since the partition of input bits among Alice and Bob is worst-case. For other partitions (in the case $n = 4$, e.g., $(\{x_1, x_2, y_1, y_2\}, \{x_3, x_4, y_3, y_4\})$) communication complexity 2 would be sufficient. This is related to "best partition communication complexity", discussed in Section 4.5.2. In [15] an example of a context-free set is presented that has communication complexity $\Omega(n)$ regardless how the (balanced!) partition of input bits among Alice and Bob looks like.

4.4 Communication Complexity Applications for Finite Automata and Turing Machines

The following can be found in [21].

Lemma 2. *Let $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ and let there be a deterministic finite automaton with accepted language L, such that $f_{L,2n} = f$ (i.e., the "length $2n$ slice of L" is exactly the set of inputs $\mathbf{xy}$ with $f(\mathbf{x},\mathbf{y}) = 1$). Then for the number s of its states holds*

$$CC(f) \leq \lceil \log s \rceil + 1.$$

Proof. Consider a deterministic finite automaton accepting L. Let $\{q_0, \ldots, q_{s-1}\}$ be its states and let $k = \lceil \log s \rceil + 1$. Then the following is a deterministic communication protocol, that accepts the input pair $(\mathbf{x},\mathbf{y})$ if and only if the concatenation $\mathbf{xy}$ belongs to L. Alice simulates the automaton on the input $\mathbf{x}$ and passes the state q_r in which the automaton ends in a binary encoding (no more than k bits) to Bob. Bob then starts simulating the automaton on $\mathbf{y}$ but with q_r as starting state. He sends to Alice bit 1, if this computation ends accepting and 0 otherwise. Hence $\lceil \log s \rceil + 1$ bits of communication are sufficient.

Proof (Proposition 9). If L is regular, then it is recognized by some deterministic finite automaton with s states. From the lemma we can conclude, that $L \subseteq \mathrm{CC}(\lceil \log s \rceil + 1)$.

Here is another application:

Exercise 17. Prove by means of communication complexity that the languages

$$L = \{\mathbf{xy} | |\mathbf{x}| = |\mathbf{y}| \text{ and } \mathrm{EQ}(\mathbf{x},\mathbf{y}) = 1\}$$

and

$$L' = \{\mathbf{xy} | |\mathbf{x}| = |\mathbf{y}| \text{ and } \mathrm{EQ}(\mathbf{x},\mathbf{y}) = 0\}$$

are not regular.

The third application concerns the following well-known simulation: For every s-state non-deterministic automaton there is a 2^s state deterministic finite automaton that accepts the same language. We can show, that this construction is essentially optimal:

Proof. We know from Exercise 17 that a deterministic finite automaton that accepts the language

$$L'_n = \{\mathbf{xy} | |\mathbf{x}| = |\mathbf{y}| = n \text{ and } \mathrm{EQ}_n(\mathbf{x},\mathbf{y}) = 0\}$$

needs at least 2^n states. On the other hand, it is easy to design a non-deterministic finite automaton with $\mathcal{O}(n)$ states for L'_n: It simply guesses two input positions $i, 1 \leq i \leq n$ and $j, n+1 \leq j \leq 2n$ and accepts, if and only if the input differs in these positions.

Communication complexity finds also application for Turing machines. The idea of the below lower bound is quite similar to the situation with deterministic finite automata (and it reminds also to the VLSI-example from Section 4.1.4). We concentrate on the following example language:

$$\text{PALINDROME} = \{\mathbf{w}\mathbf{w}^R | \mathbf{w} \in \{0,1\}^*\},$$

where $\mathbf{x}^R$ denotes the reversed string $\mathbf{x}$.

Exercise 18. Prove that PALINDROME can be recognized by a Turing machine (1) in linear time using linear space or (2) in quadratic time using logarithmic space.

Proposition 12. *If a Turing machine accepts* PALINDROME *in time* $T(N)$ *using space* $S(N)$ *then*

$$T(N) \cdot S(N) = \Omega(N^2).$$

Proof. Consider a Turing machine M that decides this language. Alice and Bob compute $\text{EQ}(\mathbf{x}, \mathbf{y})$ for $(\mathbf{x}, \mathbf{y}) \in \{0,1\}^n \times \{0,1\}^n$ by simulating the work of M on $\mathbf{x}0^n\mathbf{y}^R \in \{0,1\}^N$ where $N = 3n$. Clearly M accepts if and only if $\text{EQ}(\mathbf{x}, \mathbf{y}) = 1$. Whenever the read-only input head is located in the "$\mathbf{x}$-region" it is Alice who simulates and if it is in the "$\mathbf{y}$-region" it is Bob who simulates. When the head enters the "0-region" the current player can continue to simulate until the head enters the region corresponding to the other player. It takes at least time n to move from the $\mathbf{x}$- to the $\mathbf{y}$-region, thus this happens at most $T(N)/n$ times. Further, when the head crosses the responsibility border (i.e., when it walks out of the 0-region), the current player passes the necessary information to the other: the state ($\mathcal{O}(1)$ bits) and the contents of the work tape(s) ($\mathcal{O}(S(N))$ bits). In total they exchange $\mathcal{O}(S(N) \cdot T(N)/n)$ bits while finding $\text{EQ}(\mathbf{x}, \mathbf{y})$. Because of Proposition 2 we can conclude $T(N) \cdot S(N) = \Omega(n^2)$.

The idea works obviously also for other functions with linear communication complexity.

Remark 12. In principle the same idea should also work for pushdown automata (divide the input in two parts and let Alice simulate on the left and Bob on the right). However, passing information between players is more difficult, because of the large amount of information that can be stored in the stack. In [16] a more sophisticated communication model was defined, that is applicable for lower bound proofs in this situation. The details are beyond the scope of this tutorial.

4.5 A Survey on Communication Problems and Applications

4.5.1 Different Modes of Communication

The model of communication that was introduced in Section 4.1.2 is the basic deterministic model. However, the concept of non-determinism known from computational complexity theory turns out to be fruitful also in communication complexity. We survey some of these and related notions and results.

A non-deterministic communication protocol is a protocol, that may allow players to choose one of several messages to send. We define the computed value to be 1 if at least one transcript gives output 1. The length of the protocol transcript for the worst-case input pair is the complexity of the non-deterministic protocol. The complexity of the best nondeterministic protocol for a function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ is called the *nondeterministic communication complexity* of f and is denoted

$$CC^N(f)\,.$$

Still, it holds that the set R_α of all input pairs with transcript α is a monochromatic rectangle. It is not hard to see, that every nondeterministic protocol for f defines a (not necessarily disjoint) *covering* of $\{0,1\}^n \times \{0,1\}^n$ by monochromatic rectangles. Let $Cov^1(f)$ denote the minimum number of 1-rectangles required to cover M_f.

Exercise 19. Prove that

$$CC^N(f) = \lceil \log Cov^1(f) \rceil.$$

Let $Cov^0(f)$ denote the minimum number of 0-rectangles needed to cover M_f. The following is proved in [13]:

Proposition 13. $CC(f) = \mathcal{O}(Cov^1(f) \cdot Cov^0(f))$.

$Cov^1(f)$ and $Cov^0(f)$ are called *cover numbers*. Let $ms(f)$ denote the maximum size of an f-monochromatic 1-rectangle. The following is obvious:

Lemma 3.

$$CC(f) \geq Cov^1(f) + Cov^0(f).$$

$$CC^N(f) \geq \#f^{-1}(1)/ms(f).$$

Exercise 20. Use properties of Sylvester-matrices (see Example 6) to prove $\#\mathrm{IP}_n^{-1}(1) > 3^n$.

It can be shown, that 1-rectangles of IP_n have size at most 2^n: $ms(\mathrm{IP}_n) \leq 2^n$. Using the lemma and the exercise this gives:

Proposition 14.

$$CC^N(\mathrm{IP}_n) = \Omega(n).$$

The rectangle size argument was extended to *randomized communication protocols* in [17]. In randomized communication protocols players make use of random bits and the input is to be accepted with a certain probability. There is a "private coins model" (each player uses a separate source of randomness) and a "public coin model" (players share the random bits). In the latter model the random bits are not taken into account for communication.

Example 7. There is a private coin random communication protocol that computes EQ_n within $\mathcal{O}(\log n)$ bits of communication and with error at most $1/n$.

Proof (sketch). Let $p, n^2 < p < 2n^2$ be a prime and consider the input vectors as coefficients of degree $n-1$ polynomials. So every player has a polynomial. Alice picks some $r, 0 \leq r < p$ at random and evaluates her polynomial at r. She sends r as well as the value modulo p to Bob. Bob evaluates his polynomial at the point r. If his result equals hers modulo p, then he accepts. Otherwise he rejects.

No more than $\mathcal{O}(\log n)$ bits are communicated and the probability of error is small: If the input parts are the same, the polynomials are the same and no error occurs. If the inputs are different, the polynomials differ and there is good chance that this will be detected by random evaluation.

Another type of "communication mode" concerns rounds. The protocols mentioned in previous sections are all 1-round protocols: Alice sends some bits and Bob can compute the function value. The question is, whether it is sufficient to restrict attention to protocols bounded to a certain number of rounds or how much round restrictions affect length of protocols. This study was started in [29]. We mention some results:

- The protocol used in the proof of Proposition 13 uses binary search in a number of rounds that grows as $\log \mathit{Cov}^0(f)$. No deterministic protocol with a constant number of rounds is known, that gives the same upper bound.
- For each fixed k functions are known that need exponentially more communicated bits in $k-1$-round games compared to k-round games (see [11] for the deterministic version and [27] for the randomized).
- There is a connection between round-restricted protocols and depth/size-tradeoffs for Boolean circuits [27].

4.5.2 Different Partitions

Recall the VLSI-example from Section 4.1.4. Processors compute EQ_n but bits to be compared are at a large distance from each other. This is, what makes the computation costly. Without this restriction, the lower bound $AT \geq n^2$ would no longer be applicable.

Exercise 21. Describe a layout for a VLSI-circuit with arbitrary arrangement of inputs that achieves an area-time product $o(n^2)$.

In view of this it seems meaningful to consider communication complexity of functions on base of arbitrary input partitions. This model was introduced in [29] and is heavily used for VLSI lower bound proofs:

Definition 11. *Let (S, T) be a partition of the set of inputs of a boolean function with $\#S = \#T$. An (S, T)-*communication protocol *is a communication protocol that enables Alice and Bob (given the bits with index $i \in S$ and with index $i \in T$, respectively) to jointly compute the value of f on the given input. Its complexity is defined as usual. The* communication complexity of f with respect to (S, T) *(denoted $CC_{S,T}(f)$) is the complexity of the best protocol with respect to S, T. The* best partition communication complexity of f *is the minimum of $CC_{S,T}(f)$ over all input set partitions (S, T):*

$$CC^{\text{best}}(f) := \min_{(S,T)} CC_{S,T}(f)\,.$$

The "benchmark function" for best partition communication complexity is:

shifted equality

$$\text{SEQ}_n(\mathbf{x}, \mathbf{y}, i) = \begin{cases} 1 & \text{, if } \mathbf{x}(i) = \mathbf{y} \\ 0 & \text{, otherwise,} \end{cases}$$

where $\mathbf{x}(i)$ denotes the cyclic shift of $\mathbf{x}$ by i places.

Proposition 15.

$$CC^{\text{best}}(\text{SEQ}_n) = \Omega(m),$$

where $m = 2n + \log n$ is the size of the input.

Proposition 16. *Every VLSI-chip that computes f satisfies*

$$AT^2 \geq (CC^{\text{best}}(f))^2.$$

For proofs see [21].

4.5.3 Different Games

Depending on the situation to be modeled different versions of "communica tion games" are in use. The basic model is the two party communication game as introduced. To give an impression we mention only two of them:

- Multiparty communication: The input consists of k parts $\mathbf{x}_1, \ldots, \mathbf{x}_k$. An obvious and hopefully useful generalization of the two-party case is the following version (1): The share of player i consists of $\mathbf{x}_i$. Less obvious but also useful is version (2): The share of player i consists of all $x_j, 1 \leq i \leq k$ *except* x_i. Version (2) is dubbed "number-on-the-forehead-model" (NOF) while version (1) is the "number-in-the-hand-model". The NOF version was introduced by [5]. There are surprising connections between this model and Boolean circuit complexity. And there are also surprising protocols (see, e.g., [12, 7]).

- Communication complexity of relations, which was introduced in [19]: A relation is a subset $R \subseteq X \times Y \times Z$. Alice is given $x \in X$, Bob is given $y \in Y$ and their task is to find some $z \in Z$, such that $(x, y, z) \in R$. For an example consider the so-called *universal relation*: $U_n \subseteq \{0,1\}^n \times \{0,1\}^n \times \{1, \ldots, n\}$ is defined to be the set of all triples $(\mathbf{x}, \mathbf{y}, i)$ such that $\mathbf{x} \neq \mathbf{y}$ and $x_i \neq y_i$. The corresponding communication game is the following: Alice is given $\mathbf{x}$ and Bob is given $\mathbf{y}$. They know in advance, that $\mathbf{x} \neq \mathbf{y}$. Their task is to find some bit i, in which their inputs differ.

The communication complexity of such protocols is defined in the usual way — it is the length of the best protocol on the worst case input.

Let us elaborate somewhat more on communication complexity of relations.

Let $f : \{0,1\}^n \to \{0,1\}$ be a non-constant Boolean function. Suppose Alice is given some $\mathbf{x} \in f^{-1}(0)$ and Bob is given some $\mathbf{y} \in f^{-1}(1)$. Then they know, that their input parts differ in at least one bit and they can communicate to find one such position. This is the communication game on the following relation associated to f:

$$R_f := \{(\mathbf{x}, \mathbf{y}, i) | \mathbf{x} \in f^{-1}(0), \mathbf{y} \in f^{-1}(1), x_i \neq y_i\}.$$

Obviously, every protocol for U_n gives a protocol for R_f — this is where the name "universal relation" comes from. The trivial protocol for U_n gives an upper bound of $n + \lceil \log n \rceil$ communicated bits. In [30] several protocols are given that achieve an upper bound of $n + 2$ (while the lower bound — proved in the same paper — is $n + 1$).

We denote by $CC(R_f)$ the communication complexity of the relation R_f. Further let $d(f)$ denote the minimum depth of a Boolean circuit for f that uses only $\wedge$-, $\vee$-, and $\neg$-gates. The following interesting connection was proved in [19]:

Lemma 4.

$$d(f) = CC(R_f)\,.$$

This lemma was used in [4] for a new proof of the following fact, that is known from 1950s:

Proposition 17. *For every symmetric Boolean function f holds*

$$d(f) = \mathcal{O}(\log n).$$

Remark 13. The proof relies on a the construction of an $\mathcal{O}(\log n)$ protocol for R_f. Please note, that this is not related to the result of Exercise 7.

References

1. H. Abelson. Lower bounds on information transfer in distributed computations. In *Proceedings of the 19th IEEE FOCS*. IEEE Computer Society, Ann Arbor, pp. 151-158, 1978.

2. A. Aho, J. Ullman, and M. Yannakakis. On notions of information transfer in VLSI-circuits. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pages 133–139, 1983.
3. L. Babai, P. Frankl, and J. Simon. Complexity classes in communication complexity theory. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science, FOCS 86*. IEEE Computer Society, Toronto, pp. 337-347, 1986.
4. G.S. Brodal and T. Husfeldt. A communication complexity proof that symmetric functions have logarithmic depth. Brics report series. http://www.brics.dk/RS/96/1/BRICS-RS-96-1.ps.gz.
5. A.K. Chandra, M.L. Furst, and R.J. Lipton. Multi-party protocols. In *STOC.*, Boston, ACM, pages 94–99, 1983.
6. C. Damm and M. Holzer. Automata that take advice. In *Mathematical Foundations of Computer Science*, LNCS, vol 969, pp. 149-158, 1995.
7. C. Damm, S. Jukna, and J. Sgall. Some bounds on multiparty communication complexity of pointer jumping. *Computational Complexity*, 7(2):109–127, 1998.
8. C. Damm, M. Krause, C. Meinel, and S. Waack. On relations between counting communication complexity classes. *Journal of Computer and System Sciences*, 69:259–280, 2004.
9. R. de Wolf. Quantum communication and complexity. *Theoretical Computer Science*, 287:337–353, 2002.
10. M. Dietzfelbinger, J. Hromkovič, and G. Schnitger. A comparison of two lower-bound methods for communication complexity. *Theoretical Computer Science*, 168(1):39–51, 1996.
11. P. Duriš, Z. Galil, and G. Schnitger. Lower bounds on communication complexity. *IC*, 73:1–22, 1987.
12. V. Grolmusz. The bns lower bound for multi-party protocols is nearly optimal. *Information and Computation*, 112:51–54, 1994.
13. B. Halstenberg and R. Reischuk. Different modes of communication. *SIAM Journal on Computing*, 22, 1993.
14. J. Hromkovič. Randomized communicating protocols (a survey). In *Proceedings of the International Symposium on Stochastic Algorithms: Foundations and Applications*. LNCS, vol. 2264, Springer, pp. 1-32.
15. J. Hromkovič. *Communication Complexity and Parallel Computing.* Springer, Berlin, 1997.
16. J. Hromkovič and G. Schnitger. Pushdown automata and multicounter machines, a comparison of computation modes. In *Automata, Languages and Programming: 30th International Colloquium*, pages 66–80, 2003.
17. M. Karchmer, E. Kushilevitz, and N. Nisan. Fractional covers and communication complexity. *SIAM Journal on Discrete Mathematics*, 8:76–92, 1995.
18. M. Karchmer, I. Newman, M. Saks, and A. Wigderson. Non-deterministic communication complexity with few witnesses. *Journal of Computer and System Sciences*, 49:247–257, 1994.
19. M. Karchmer and A. Wigderson. Monotone circuits for connectivity require super-logarithmic depth. In *STOC.*, pages 539–550, 1988.
20. R.M. Karp and R.J. Lipton. Turing machines that take advice. *L'Enseignement Mathématique*, 28:191–209, 1982.
21. E. Kushilevitz and N. Nisan. *Communication Complexity.* Cambridge University Press, 1996.

22. R.J. Lipton and R. Sedgewick. Lower bounds for VLSI. In *13th Annual ACM Symposium on the Theory of Computing*, pages 300–307, 1981.
23. S.V. Lokam. Spectral methods for matrix rigidity with applications to size-depth tradeoffs and communication complexity. In *FOCS*, pages 6–15, 1995.
24. L. Lovász. Communication complexity: A survey. In B. Korte, L. Lovász, H.J. Prömel, and A. Schrijver, editors, *Paths, Flows, and VLSI Layout. Algorithms and Combinatorics 9*, pages 235–265. 1990.
25. L. Lovász and M. Saks. Lattices, Möbius functions and communication complexity. *Journal of Computer and System Sciences*, 47:322–349, 1993.
26. K. Mehlhorn and E.M. Schmidt. Las vegas is better than determinsm in VLSI and distributed computing. In *14th Annual ACM Symposium on Theory of Computing*, pages 330–337, 1982.
27. N. Nisan and A. Wigderson. Rounds in communication complexity revisited. *SIAM Journal on Computing*, 22, 1993.
28. A. Orlitsky and A. Gamal. Communication complexity. In *Complexity in Information Theory*, pages 16–61. 1988.
29. C. Papadimitriou and M. Sipser. Communication complexity. *JCSS*, 28:260–269, 1984.
30. G. Tardos and U. Zwick. The communication complexity of the universal relation. In *Proceedings of the 12th IEEE Conference on Computational Complexity*, pages 247–259, 1997.
31. C. Thompson. Area-time complexity for VLSI. In *Proceedings of the 11th ACM STOC*. ACM, Atlanta, pp. 81-88.
32. A.C.C. Yao. Some complexity questions related to distributive computing. In *Proceedings of the 11th Annual ACM Symposium on the Theory of Computing*, Atalanta, pp. 209-213, 1979.

5

Formal Languages and Concurrent Behaviours

Jetty Kleijn[1] and Maciej Koutny[2]

[1] LIACS, Leiden University
P.O.Box 9512, NL-2300 RA Leiden, The Netherlands
kleijn@liacs.nl
[2] School of Computing Science, Newcastle University
Newcastle upon Tyne, NE1 7RU, United Kingdom
maciej.koutny@ncl.ac.uk

Summary. This is a tutorial based on a course delivered as part of the International PhD School in Formal Languages and Applications located at the Rovira i Virgili University in Tarragona, Spain. It is focused on an application of formal language theory to represent behaviours of concurrent systems necessitating a generalisation of language theory to traces, which originates with the work of Mazurkiewicz in 1977. The tutorial uses Petri nets as an underlying system model which allows one to clearly distinguish between causality and independence between executions of actions, a major feature of concurrent behaviour.

Keywords: formal languages, traces, concurrency, causality, independence, partial orders, Petri nets, inhibitor arcs, processes.

5.1 Introduction

The dynamic behaviours of concurrent systems are not always adequately captured with the help of purely functional input-output descriptions. Often one is more interested in the modelling of ongoing evolutions of such systems at the interface with the environment, e.g., when communicating or reacting to external stimuli. One way of representing such evolutions is to use sequences of executed actions, which in a natural way leads to a formal language semantics of dynamic systems. A successful example of this approach are finite state machines and their languages which have numerous applications in almost every branch of Computer Science. Another example are Turing machines and their languages which delimit the effectiveness of computational behaviour. Both classes of machines are of a sequential nature which makes languages a suitable semantical domain. However, plain words and languages are only of limited usefulness when it comes to faithful representation of concurrent behaviours. For example, a sequential description of behaviour cannot be used to describe the result of action refinement for which the information about concurrency or independence, as opposed to causality, is of crucial importance. The problem of insufficient expressibility of sequential descriptions was recognised in the 1970s when concurrent systems became a focus of intensive research activity carried out by various groups, and when it was realised that additional information should be added to the sequential descriptions of system behaviours. An example coming from the process algebra world is Milner's observational equivalence [39] — replacing language equivalence as a means of comparing different systems — which allows one to identify the exact points when choices between alternative actions were made during system executions. Another such example are traces — introduced by Mazurkiewicz [35] — providing explicit information on the causal dependencies between executed actions. It is this latter approach which is the subject of this tutorial.

A key concept behind traces is that a given concurrent run can be observed in different ways depending on the viewpoint of the observer and on the particular way of recording this behaviour. Trace theory then provides a tool which allows one to identify in a precise way different observations of a concurrent behaviour. In its most basic form, traces equate sequences of executed actions on basis of given independencies between such actions. The original idea of Mazurkiewicz was to use the well-developed tools of formal language theory for the analysis of concurrent systems, understood as Petri nets [44, 45, 46].

Petri nets are an operational model which directly generalises state machines (labelled transition systems) through the notions of a state and state change. Both models allow a graphical representation. What makes Petri nets radically different from state machines is their ability to represent states as distributed entities, and state changes as affecting only local parts of these distributed states in a way prescribed by the underlying graphical structure. Whereas the executions or runs of a sequential system consist of actions executed one-by-one in a totally ordered fashion, the actions of a run of a concurrent system are not necessarily executed one after the other. Having distributed states and local effects makes concurrency aspects explicit since actions may be independent in the sense that they involve disjoint parts of distributed states. Hence words and languages (sets of words) which are appropriate models for the behaviours of sequential systems are no longer sufficient since actions executed in a concurrent system are only partially ordered. The concern of trace theory is how to add information to observations in order to convey the essence of causality between executed actions (i.e., the necessary ordering in the sense that cause must precede effect).

Trace theory has as its starting point an alphabet of action names enriched with information about which (occurrences of) actions are independent (or non-interfering). A key assumption is that the order of observation of two independent actions is accidental. Hence two sequential observations (words) which differ only w.r.t. the order of occurrence of independent actions are in fact observations of *the same* concurrent run and consequently may be identified. A trace is then simply the resulting equivalence class of all sequential observations of the same underlying concurrent run. In this way, formal language theory is lifted to quotient structures consisting of equivalent observations, and a number of standard language theoretic tools can therefore be adapted and applied in the analysis of concurrent systems. Moreover, by explicitly recording the dependencies between executed actions a unique (causal) partial order can be associated to each trace. In other words, traces can be seen as *partial orders* in the same way as words can be seen as total orders.

The tutorial is organised in the following way. After a preliminary section on sets, graphs and languages, we introduce traces and recall their main properties, including the underlying dependence graphs. We then consider Elementary Net systems [47] which are generally regarded as the most fundamental class of Petri nets, and were indeed the model which inspired the introduction of traces. We investigate both sequential and non-sequential ways of executing them. The trace-based behaviour is obtained by taking sequential executions and combining them with the structural information about the dependencies between executed actions obtained from the graph structure of a net. That this approach is sound follows from the fact that the partial orders defined by traces coincide with the partial order semantics of nets represented by the non-sequential observations captured by operationally derived processes. This treatment is then repeated for two significant, and practically relevant, extensions of Elementary Net systems (note that 'Petri net' is actually a generic name rather than a single model). The first extension consists in adding inhibitor arcs to the net, and the other in extending the notion of a global state. In each case we demonstrate the necessary generalisations of the concept of action independence, leading to comtraces and local traces, respectively. The first is based on the enhanced structure of the net whereas the other is history dependent.

The tutorial is based on existing work and contains no proofs. The main text presents key definitions and results as numbered items, whereas supporting observations (and suggested exercises for the reader) are marked with the ✓symbol. For further background, proofs and references the reader is provided with bibliographical remarks at the end of each technical section.

5.1.1 A Running Example

Throughout this tutorial, we will discuss various aspects of simple yet practically relevant concurrent systems consisting of producers, buffers and consumers.

We start with a model consisting of three (sequential) components: a producer `Prod`, buffer of capacity one `Buff` and consumer `Cons`. Each component is characterised by its *language* which is the set of finite sequences of atomic actions it can execute. `Prod` can execute three actions: `m` representing $\underline{\text{m}}$aking of an item, `a` representing $\underline{\text{a}}$dding of a newly produced item to the buffer, and `r` representing the $\underline{\text{r}}$etirement of the producer. `Cons` can execute two actions:

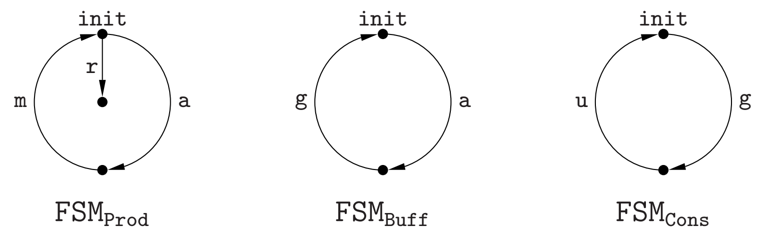

init
r
m
a
$\mathtt{FSM}_{\mathtt{Prod}}$
init
g
a
$\mathtt{FSM}_{\mathtt{Buff}}$
init
u
g
$\mathtt{FSM}_{\mathtt{Cons}}$

Fig. 5.1. Finite state machines for the running example.

`g` representing getting an item from the buffer, and `u` representing using the newly acquired item. `Buff` can execute the `a` and `g` actions. The sequences of allowed action executions are given by the regular languages generated by the three finite state machines shown in Figure 5.1 (all states in these machines are considered final). The three components together form a concurrent system in which they operate independently except for the requirement that the actions shared by two processes are executed if both of the processes (can and) do so.

For example, `ama` is a valid action sequence for the producer, but it is not a valid behaviour of the combined system, because the buffer does not allow two `a` actions without an intermediate `g`. On the other hand, the sequence `amgau` can be executed by the whole system, and so it is a valid history. Note that `amgru` is a history leading to a deadlock, i.e., a global state in which none of the five actions can be executed.

5.2 Preliminaries

A *relational tuple* is a tuple *reltuple* $\overset{\mathrm{df}}{=} (X_1, \dots, X_m, Q_1, \dots, Q_n)$ where the X_i's are disjoint sets forming the *domain*, and the Q_i's are relations involving the elements of the domain and perhaps some other elements.[1] For example, directed graphs and finite state machines can be regarded as relational tuples. In fact, in all cases considered later on, a relational tuple can be viewed as a graph of some sort and we will use the usual graphical conventions to represent its nodes (i.e., the elements of its domain), various relationships between these nodes, and some particular characteristics of these nodes (e.g., the initial state of a finite state machine, or a labelling of the elements).

A particular issue which links together various kinds of relational tuples is the idea that what really matters is the *structures* they represent rather than the identities of the elements of their domains. A technical device which can be used to capture such a view is as follows: two relational tuples, *reltuple* and *reltuple*$'$, are *isomorphic* if there is a bijection ψ from the domain of *reltuple* to the domain of *reltuple*$'$ such that if we replace throughout *reltuple*

[1]In this tutorial, $m \leq 2$ and $n \leq 4$. Note that suitable Q_i's can represent functions on the domain as well as subsets and individual elements of the domain.

each element x in its domain by $\psi(x)$ then the result is *reltuple*$'$.[2] It is then standard to consider isomorphic relational tuples as undistinguishable.

5.2.1 Set Theoretic Notations

$\mathbb{N}$ denotes the set of natural numbers including zero. The powerset of a set X is denoted by $\mathbb{P}(X)$, and the cardinality of a finite set X by $|X|$. Sets $X_1, \ldots X_n$ form a partition of a set X if they are non-empty disjoint subsets of X such that $X = X_1 \cup \ldots \cup X_n$.

A *labelling* ℓ for a set X is a function from X to a set of labels. The labelling can be applied to finite sequences of elements of X, $\ell(x_1 \ldots x_n) \overset{\text{df}}{=} \ell(x_1) \ldots \ell(x_n)$, and to finite sequences of subsets of X, $\ell(X_1 \ldots X_n) \overset{\text{df}}{=} \ell(X_1) \ldots \ell(X_n)$. The composition $R \circ Q$ of two relations $R \subseteq X \times Y$ and $Q \subseteq Y \times Z$ comprises all pairs (x, z) in $X \times Z$ for which there is y in Y such that $(x, y) \in R$ and $(y, z) \in Q$.

Definition 1 : relations

Let R be a binary relation on a set X.

- $R^{-1} \overset{\text{df}}{=} \{(y, x) \mid (x, y) \in R\}$ denotes the inverse of R.
- $R^0 = id_X \overset{\text{df}}{=} \{(x, x) \mid x \in X\}$ is the identity relation on X.
- $R^n \overset{\text{df}}{=} R^{n-1} \circ R$ is the n-th power of R, for $n \geq 1$.
- $R^+ \overset{\text{df}}{=} R^1 \cup R^2 \cup \ldots$ is the transitive closure of R.
- $R^* \overset{\text{df}}{=} R^0 \cup R^+$ is the transitive and reflexive closure of R.
- R is symmetric / reflexive / irreflexive / transitive if, respectively, $R = R^{-1}$ / $id_X \subseteq R$ / $id_X \cap R = \varnothing$ / $R \circ R \subseteq R$.
- R is acyclic if R^+ is irreflexive.

The restriction of a function $f : X \to Y$ to a subset Z of X is denoted by $f|_Z$, and of a relation $R \subseteq X \times Y$ to a subset Z of $X \times Y$ by $R|_Z$. The *domain* of R is $dom_R \overset{\text{df}}{=} \{x \mid (x, y) \in R\}$ and its *codomain* is given by $codom_R \overset{\text{df}}{=} \{y \mid (x, y) \in R\}$. We will often use the infix notation $x \, R \, y$ to denote $(x, y) \in R$.

Definition 2 : equivalence relations

A binary relation R on a set X is an equivalence relation if it is reflexive, symmetric and transitive. An equivalence class of R is any maximal subset of equivalent elements.

[2]This definition is not strictly formal, but it should convey sufficient meaning to make the presentation clear.

In other words, R is an equivalence relation *iff* $R = (R \cup R^{-1})^*$ holds ✓. If R is an equivalence relation on X, then X/R denotes the set of all equivalence classes of R.

Given an equivalence relation R on X and a function f defined for n-tuples of elements of X, it is often useful to lift f to n-tuples of equivalence classes of R by setting $f(R_{x_1}, \ldots, R_{x_n}) \overset{\text{df}}{=} f(x_1, \ldots, x_n)$ where each R_{x_i} is the equivalence class of R containing x_i. We say that f is *well-defined* on X/R if the value returned does not depend on the choice of the representing element x_i from R_{x_i}.

Definition 3 : partial orders

A binary relation R on a set X is a partial order if it is irreflexive and transitive.

In other words, R is a partial order *iff* $R = R^+ \setminus id_X$ holds ✓.

Definition 4 : partially ordered sets

A labelled partially ordered set (or poset) $po \overset{\text{df}}{=} (X, \prec, \ell)$ is a relational tuple consisting of a finite[a] set X, a partial order $\prec$ on X, and a labelling ℓ of X. The poset is total (or linear) if, in addition, all distinct elements of X are ordered.

[a]For simplicity we restrict ourselves to finite posets.

More precisely, po is total if $x \prec y$ or $y \prec x$ for all $x \neq y$ in X. Two elements $x \neq y$ of X are *unordered* if neither $x \prec y$ nor $y \prec x$; we denote this by $x \frown y$. Moreover, we write $x \preceq y$ if $x \prec y$ or $x = y$.

A total poset tpo is a *linearisation* of a poset po if they have the same domain and labelling, and the partial order relation of the former extends (includes) the partial order relation of the former. The set of all linearisations of po is denoted by $lin(po)$.

The *intersection* $\bigcap \mathcal{TPO}$ of a non-empty set of total posets $tpo = (X, \prec_{tpo}, \ell)$ with the same domain X and labelling ℓ is $(X, \prec, \ell)$ where $\prec$ is the relation comprising all pairs (x, y) of elements of X such that $x \prec_{tpo} y$ for each tpo in $\mathcal{TPO}$. The set of all linearisations of a poset po is non-empty and $po = \bigcap lin(po)$ which means that any poset can be identified with its set of linearisations ✓.

5.2.2 Directed Acyclic Graphs

As usual, we define a labelled directed graph (or simply graph) G as a relational tuple (V, A, ℓ) consisting of a set of nodes V, a set of arcs $A \subseteq V \times V$, and a labelling of V. (For simplicity we restrict ourselves to finite graphs, i.e., with a finite set of nodes).

Definition 5 : dags

A graph $G = (V, A, \ell)$ is acyclic (transitive) if A is an acyclic (transitive) relation. A dag is a (directed) acyclic graph.

Figure 5.2 shows an example of a dag $\mathtt{G}$ with six nodes. Any poset is a dag and, conversely, any dag defines a (unique) poset after adding all arcs implied by transitivity: for a dag $G = (V, A, \ell)$ the relation A^+ is a partial order on V, and we refer to the graph $G^+ \overset{\text{df}}{=} (V, A^+, \ell)$ as the *transitive closure* of G. Deleting all transitive arcs from G yields its *Hasse diagram*, i.e., the graph $hasse(G) \overset{\text{df}}{=} (V, A \setminus (A \circ A^+), \ell)$. Figure 5.2 shows examples of both these notions.

The Hasse diagram of a poset *po* is the minimal (w.r.t. the number of arcs) dag G such that $G^+ = po$. Moreover, a dag is a poset *iff* it is its own transitive closure, and the transitive closures of two dags coincide *iff* their Hasse diagrams coincide ✓.

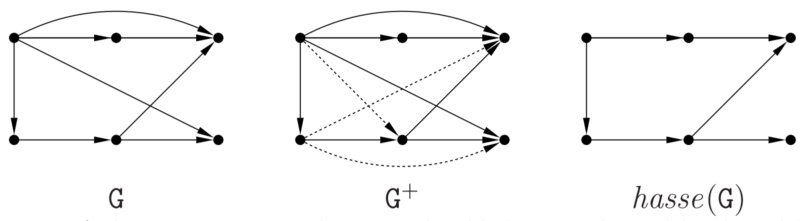


Fig. 5.2. A dag, its transitive closure with added arcs indicated by dotted lines, and its Hasse diagram. Note that node labels are omitted as they are irrelevant.

5.2.3 Words and Languages

Alphabets, words, and languages are the main notions for recording the sequential view of a system's behaviour.

Definition 6 : alphabets

An alphabet Σ is a finite non-empty set of symbols. A word (over Σ) is any finite sequence $a_1 \cdots a_n$ of symbols a_i (from Σ), and a language (over Σ) is any set of words (over Σ).

In the case that $n = 0$ in the above definition, one is dealing with the empty sequence or *empty word*, denoted by λ. The set of all words over Σ including λ, is denoted by Σ^*.

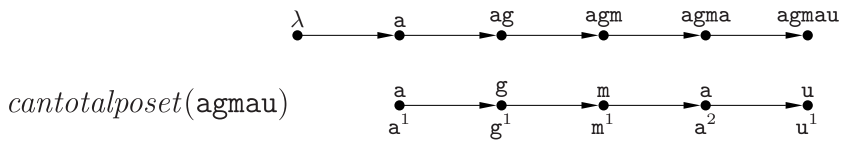


Fig. 5.3. Hasse diagram of the prefix ordering for the word `agmau`, and Hasse diagram of its canonical total poset showing the identities of its nodes (bottom) and labels (top).

Definition 7 : words

Let $u = a_1 \dots a_n$ and $v = b_1 \dots b_m$ be two words over an alphabet Σ, $(m, n \geq 0)$.

- $uv \overset{\mathrm{df}}{=} a_1 \dots a_n b_1 \dots b_m$ is the concatenation of u and v.
- $\mathit{length}(u) \overset{\mathrm{df}}{=} n$ is the length of u.
- $\mathit{alphabet}(u)$ comprises all symbols occurring within u.
- $\#_a(u)$ is the number of occurrences of a symbol a within u.
- $\mathit{occ}(u)$ is the set of symbol occurrences of u comprising all indexed symbols a^i with $a \in \mathit{alphabet}(u)$ and $1 \leq i \leq \#_a(u)$.

For the running example in this tutorial, we will use the alphabet $\mathtt{\Sigma} = \{\mathtt{a}, \mathtt{g}, \mathtt{m}, \mathtt{r}, \mathtt{u}\}$. For the string $u = \mathtt{agmau}$ over $\mathtt{\Sigma}$, we have: $\mathit{length}(u) = 5$, $\mathit{alphabet}(u) = \{\mathtt{a}, \mathtt{g}, \mathtt{m}, \mathtt{u}\}$, $\#_{\mathtt{a}}(u) = 2$ and $\mathit{occ}(u) = \{\mathtt{a}^1, \mathtt{a}^2, \mathtt{g}^1, \mathtt{m}^1, \mathtt{u}^1\}$.

Fact 8 : The set of all words over an alphabet Σ with concatenation and the empty word forms a monoid. That is, concatenation is associative[a] and λ is its unit.[b]

[a] $(uv)w = u(vw)$ for all u, v, w in Σ^*.
[b] $\lambda u = u\lambda = u$ for each u in Σ^*.

A word u is a *prefix of* a word v if $v = uw$ for some word w. We denote this by $u \trianglelefteq v$. Moreover, if $u \trianglelefteq v$ and $u \neq v$ then we write $u \triangleleft v$. The prefix relation $\triangleleft$ on words is a partial order ✓. For example, $\mathtt{ag} \triangleleft \mathtt{agma}$.

For a given word, $\triangleleft$ is a total order on its prefixes which can be interpreted as saying that every word has a unique history (see Figure 5.3). Moreover, every word, being a sequence of symbols, corresponds directly to a total poset. This is an important relationship now made precise.

To start with, the elements of a total poset $\mathit{tpo} = (X, \prec, \ell)$ can be listed as a (unique) sequence $x_1 \dots x_n$ such that $x_i \prec x_j$ *iff* $i < j$. The word *generated* by *tpo* is then defined as $\mathit{word}(\mathit{tpo}) \overset{\mathrm{df}}{=} \ell(x_1 \dots x_n)$. Total orders are isomorphic *iff* the words they generate are the same ✓.

Now, given a word u it is clearly possible to see it as corresponding to any total poset *tpo* such that $\mathit{word}(\mathit{tpo}) = u$. Since all such total posets are isomorphic it does not really matter which one is chosen and for our purposes it is convenient to single out one such poset. It is called the *canonical* total poset of u and is defined as $\mathit{cantotalposet}(u) \overset{\mathrm{df}}{=} (\mathit{occ}(u), \prec, \ell)$ where $a^i \prec b^j$ if the i-th occurrence of a precedes the j-th occurrence of b within u, and $\ell(a^i) \overset{\mathrm{df}}{=} a$, for all symbol occurrences a^i and b^j in $\mathit{occ}(u)$. Distinct words have distinct canonical total posets, and the word generated by the canonical total poset of a word is that word itself ✓.

5.2.4 Bibliographical Remarks

We have recalled the necessary notions and results used later in this tutorial. Most of them are standard. We only mention that the fact that any poset can be identified with its set of linearisations is usually known as 'Szpilrajn's Theorem' and has been first given in [49] in a fully general setting.

5.3 Traces

Words represent a sequential view of the actions executed by a system. As such, no further information is provided on the intrinsic dependencies among the actions and the resulting necessary ordering of their occurrences. The introduction of traces starts from the definition of a concurrency alphabet, which simply states which symbols are considered as representing independent actions (not interfering with each other) and thus should be treated as concurrent.

Definition 9 : concurrency alphabets

A concurrency alphabet is a pair $CA \overset{\mathrm{df}}{=} (\Sigma, \mathit{Ind})$ where Σ is an alphabet and Ind is an irreflexive and symmetric binary relation over Σ called an independence relation.

When two symbols are not independent (i.e., they do not appear as a pair in the given independence relation), they are said to be *dependent.* Note that this dependence relation is reflexive.

Let (Σ, Ind) be a concurrency alphabet and let $u, v \in \Sigma^*$. We write $u \sim_{\mathit{Ind}} v$ if there are words w and z and independent symbols $(a, b) \in \mathit{Ind}$ such that $u = wabz$ and $v = wbaz$. Thus $u \sim_{\mathit{Ind}} v$ means that they are the same word except for a change of order of two adjacent occurrences of independent symbols. *Trace equivalence* (with respect to (Σ, Ind)) is the equivalence relation $\equiv_{\mathit{Ind}}$ over Σ^* obtained as the reflexive and transitive closure of $\sim_{\mathit{Ind}}$. Thus, two words are trace equivalent if one can be obtained from the other by changing (repeatedly) the order of adjacent occurrences of independent symbols. This means ✓ that $u \equiv_{\mathit{Ind}} v$ *iff* $\#_a(u) = \#_a(v)$ for all $a \in \Sigma$ and the order of occurrences a^i and b^j is the same within u and v for all pairs a, b of dependent symbols and for all $1 \leq i \leq \#_a(u)$ and $1 \leq j \leq \#_b(u)$.

Continuing our running example with $\mathtt{\Sigma} = \{\mathtt{a}, \mathtt{g}, \mathtt{m}, \mathtt{r}, \mathtt{u}\}$, we assume $\mathtt{CA} = (\mathtt{\Sigma}, \mathtt{Ind})$ is the concurrency alphabet with independence relation $\mathtt{Ind}$ given by:

$$\mathtt{Ind} = \{(\mathtt{r},\mathtt{g}), (\mathtt{g},\mathtt{r}), (\mathtt{r},\mathtt{u}), (\mathtt{u},\mathtt{r}), (\mathtt{m},\mathtt{g}), (\mathtt{g},\mathtt{m}), (\mathtt{m},\mathtt{u}), (\mathtt{u},\mathtt{m}), (\mathtt{a},\mathtt{u}), (\mathtt{u},\mathtt{a})\}.$$

Then $\mathtt{agmr} \sim_{\mathtt{Ind}} \mathtt{amgr} \sim_{\mathtt{Ind}} \mathtt{amrg}$ and so $\mathtt{agmr} \equiv_{\mathtt{Ind}} \mathtt{amrg}$.

Definition 10 : traces

A trace over a concurrency alphabet (Σ, Ind) is any equivalence class of the trace equivalence relation $\equiv_{\mathit{Ind}}$, and a trace language is any set of traces over (Σ, Ind).

The trace containing a given word u is denoted by $[u]_{\mathit{Ind}}$, and the set of all traces over (Σ, Ind) by $\Sigma^*/_{\equiv_{\mathit{Ind}}}$. Whenever the independence relation *Ind* is understood, we may drop the subscript *Ind*. Note that the *empty* trace $[\lambda]$ is $\{\lambda\}$ rather than the empty set. For the running example, $[\mathtt{amgr}] = \{\mathtt{amgr}, \mathtt{agmr}, \mathtt{amrg}\}$.

If two words are trace equivalent, then both their lengths and alphabets are the same ✓. Hence, the alphabet of a trace and its length, defined in the natural way as $\mathit{alphabet}(\alpha) \overset{\text{df}}{=} \mathit{alphabet}(u)$ and $\mathit{length}(\alpha) \overset{\text{df}}{=} \mathit{length}(u)$, where u is any word belonging to the trace α, are both well-defined notions. Also, trace concatenation (or sequential composition, in operational terms) defined as $[u] \circ [v] \overset{\text{df}}{=} [uv]$ is a well-defined operation. This follows from the observation ✓ that uv and $u'v'$ are trace equivalent whenever $[u] = [u']$ and $[v] = [v']$.

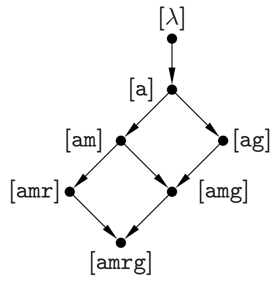
Fact 11 : The set $\Sigma^*/_{\equiv_{\mathit{Ind}}}$ of all traces over a concurrency alphabet (Σ, Ind) is a monoid. That is, concatenation of traces is an associative[a] operation with the empty trace $[\lambda]$ as its unit.[b]

[a] $(\alpha \circ \beta) \circ \gamma = \alpha \circ (\beta \circ \gamma)$ for all α, β, γ in $\Sigma^*/_{\equiv}$.
[b] $\alpha \circ [\lambda] = [\lambda] \circ \alpha = \alpha$ for each α in $\Sigma^*/_{\equiv}$.

The last result can be pushed a little bit further.

Fact 12 : The trace monoid $\Sigma^*/_{\equiv_{\mathit{Ind}}}$ is partially commutative in the sense that, for any pair of traces α and β, $\mathit{alphabet}(\alpha) \times \mathit{alphabet}(\beta) \subseteq \mathit{Ind}$ implies $\alpha \circ \beta = \beta \circ \alpha$ and, moreover, the converse holds whenever the alphabets of α and β are disjoint.

We lift the prefix relationship on words to the level of traces, by stating that a trace α is a *prefix of* a trace β if $\beta = \alpha \circ \gamma$ for some trace γ. We denote this by $\alpha \trianglelefteq \beta$. Moreover, if $\alpha \trianglelefteq \beta$ and $\alpha \neq \beta$ then we write $\alpha \vartriangleleft \beta$. For



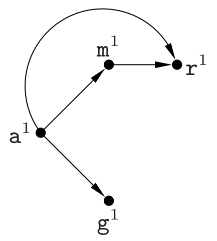


Fig. 5.4. Hasse diagram of the prefix ordering for the trace [amrg] and its dependence graph (labelling is obvious and therefore omitted).

example, [a] $\lhd$ [amr]. It then follows that $u \lhd v$ and $v \equiv w$ implies $[u] \lhd [w]$ where u, v, and w are words ✓. Consequently, $u \lhd v$ implies $[u] \lhd [v]$, but $[u] \lhd [v]$ does not necessarily imply that $u \lhd v$ holds ✓.

Unlike words which have a unique history that is captured through the prefix relation (see Figure 5.3), the prefix relation for traces can be interpreted as associating with a trace several histories which are sequential observations (represented by the words in the trace) of possibly concurrent behaviour. One could say that the concurrency between occurrences has been 'flattened' to choosing an order of occurrence. In Figure 5.4 the trace [amrg] is depicted with the ordering of its prefixes. That trace has three histories in correspondence with the three directed paths defining its elements amgr, agmr, and amrg.

To extract information on the dependencies between the symbol occurrences in a trace, dependence graphs are used. In these graphs, the relationship between dependence and order is made explicit.

Definition 13 : dependence graphs

A dependence graph over a concurrency alphabet (Σ, Ind) is a dag in which two nodes are connected iff they are labelled with dependent symbols.

In other words, a dag $G = (V, A, \ell)$ is a dependence graph over (Σ, Ind) if $\ell : V \to \Sigma$ is a labelling such that for all distinct nodes $x, y \in V$, there is an arc $(x, y) \in A \cup A^{-1}$ if and only if $(\ell(x), \ell(y)) \notin Ind$.

Every dag defines a language, consisting of all words that can be read from its labels without violating the order implied by its arcs (e.g., by using a topological sorting procedure). Formally, the *language* of a dag G comprises all words associated with the total extensions of its transitive closure, i.e., $language(G) \overset{\text{df}}{=} word(lin(G^+))$. Their languages provide a fairly precise characterisation of dependence graphs, as two dependence graphs over the same concurrency alphabet are isomorphic *iff* their languages are the same ✓.

Conversely, assuming a given concurrency alphabet (Σ, Ind), with every word $u \in \Sigma^*$ a dependence graph can be associated. The *canonical dependence graph* of u is the dag $candepgraph_{Ind}(u)$ given by $(occ(u), \prec, \ell)$ where $\ell(a^i) \overset{\text{df}}{=} a$,

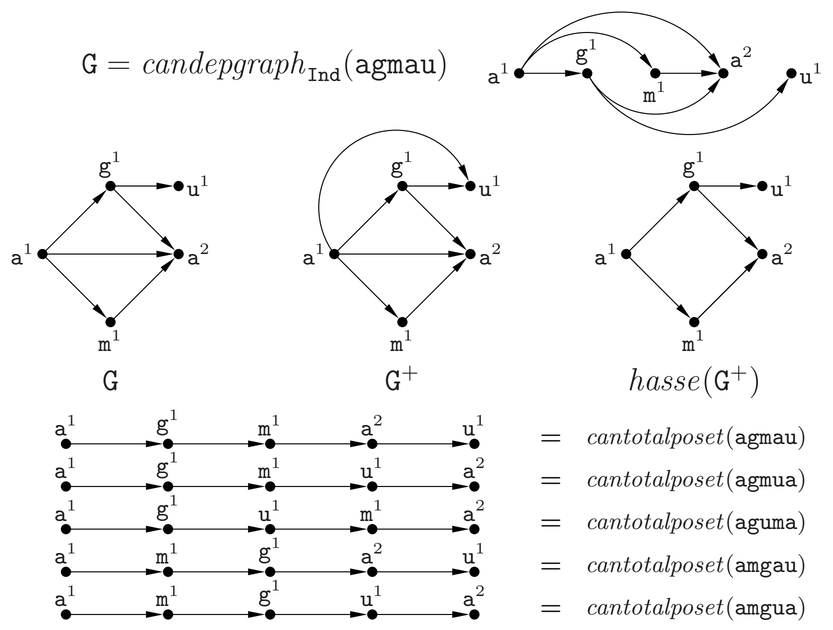


Fig. 5.5. The canonical dependence graph (twice) of agmau, its transitive closure, the Hasse diagram of its transitive closure, and Hasse diagrams of its total extensions. (Labelling of the nodes is obvious and therefore omitted.)

for all $a \in \mathit{alphabet}(u)$ and a^i in $\mathit{occ}(u)$, and $a^i \prec b^j$ if the i-th occurrence of a (strictly) precedes the j-th occurrence of b within u and $(a, b) \notin \mathit{Ind}$. Note that the canonical dependence graph of the empty word is $\mathit{candepgraph}_{\mathit{Ind}}(\lambda) = (\varnothing, \varnothing, \varnothing)$, and that $\mathit{candepgraph}_{\varnothing}(u) = \mathit{cantotalposet}(u)$. Figure 5.5 shows an example of a canonical dependence graph, and its total extensions.

>From the definition of $u \sim_{\mathit{Ind}} v$, it follows that the canonical dependence graphs of two words are *equal* whenever they are trace equivalent $\checkmark$. Hence the canonical dependence graph of a trace α can be defined as $\mathit{candepgraph}(\alpha) \overset{\text{df}}{=} \mathit{candepgraph}(u)$ where u is any word in α. What is more, the language defined by the canonical dependence graph of a trace consists exactly of the words comprising that trace $\checkmark$. It therefore follows that distinct traces have distinct canonical dependence graphs $\checkmark$. Hence there is a one-to-one correspondence between dependence graphs and traces.

On basis of the canonical dependence graph of a trace, we define the *canonical poset* of a trace α as $\mathit{canposet}(\alpha) \overset{\text{df}}{=} \mathit{candepgraph}(\alpha)^+$. >From the above observations, it follows that the canonical poset of a trace properly captures the behaviours represented by the words in that trace.

Fact 14 : Let α be a trace.

- $lin(canposet(\alpha)) = cantotalposet(\alpha)$.[a]
- $word(lin(canposet(\alpha))) = \alpha$.

[a]Note that in $cantotalposet(\alpha)$ the trace α is treated as a set of words.

All information on the dependencies between the occurrences in a trace is represented in its uniquely associated poset.

5.3.1 Bibliographical Remarks

Main independent sources of trace theory are [7] (in the context of combinatorial problems) and [35, 29] (in the context of concurrency theory). An extensive account of trace theory is provided by [11] which, in particular, contains a chapter on dependence graphs [20]. For a bibliography on traces see [15].

5.4 Elementary Net Systems

In this section we first briefly discuss Petri nets as a system model, or rather as a framework for the modelling of concurrent systems. Then we introduce in more detail Elementary Net systems, the most basic Petri net model. In this model the key primitive notions underlying the operation of concurrent systems are explicitly represented and as such it has been the inspiration for the development of trace theory. In later sections, we will discuss more expressive net classes and how they lead to generalizations of traces.

The description of a Petri net comes in two parts, giving its static and dynamic aspects. The (static) structure of a Petri net is a graph specifying the local states (called places) of the system being modelled and its possible actions (called transitions). Global (system) states consist of combinations of the local states and it is the role of transitions to change those states in accordance with the given (dynamic) rules. Each transition has a neighbourhood of places with which it is linked and there are specific rules when transitions can occur (concurrently) and the effect of such occurrence. Both notions are fully determined by the transition's neighbourhood, i.e., every transition occurrence depends on neighbouring local states and also its effect when it occurs is completely local. A net system is fully specified when also an initial state is supplied from which possible behavioural scenarios are initiated. By varying the kind and nature of the relationships between places and transitions, as well as the precise notions of global state, and the enabling and occurrence rules, one obtains different classes of Petri nets.

First we introduce the basic structure underlying every Petri net. The definition below captures what presumably is the most fundamental class of nets.

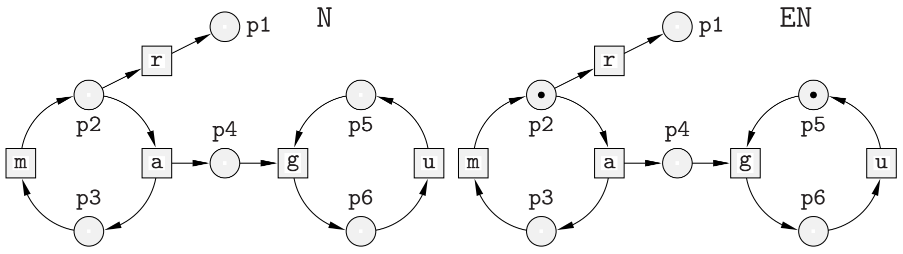


Fig. 5.6. A net without and with configuration (an EN-system) for the running example with a producer and a consumer subnet connected by a (buffer) place p4.

Definition 15 : nets

A net N is a relational tuple (P, T, F) with P and T disjoint finite sets of nodes, called respectively places and transitions, and $F \subseteq (T \times P) \cup (P \times T)$ the flow relation.

In diagrams, places are drawn as circles, and transitions as rectangles. The flow relation is represented by directed arcs between them. Hence nets are drawn as bipartite graphs.

Figure 5.6 shows the net $\mathtt{N} = (\mathtt{P}, \mathtt{T}, \mathtt{F})$, where $\mathtt{P} = \{\mathtt{p1}, \mathtt{p2}, \mathtt{p3}, \mathtt{p4}, \mathtt{p5}, \mathtt{p6}\}$ is the set of places, $\mathtt{T} = \{\mathtt{a}, \mathtt{g}, \mathtt{m}, \mathtt{r}, \mathtt{u}\}$ is the set of transitions, and the flow relation F comprises the following twelve arcs:

$$\begin{array}{cccccc} (\mathtt{r}, \mathtt{p1}) & (\mathtt{p2}, \mathtt{r}) & (\mathtt{p3}, \mathtt{m}) & (\mathtt{m}, \mathtt{p2}) & (\mathtt{p2}, \mathtt{a}) & (\mathtt{a}, \mathtt{p3}) \\ (\mathtt{a}, \mathtt{p4}) & (\mathtt{p4}, \mathtt{g}) & (\mathtt{p5}, \mathtt{g}) & (\mathtt{g}, \mathtt{p6}) & (\mathtt{p6}, \mathtt{u}) & (\mathtt{u}, \mathtt{p5})\ . \end{array}$$

Let (P, T, F) be a net. The *inputs* and *outputs* of a node $x \in P \cup T$ are the sets ${}^\bullet x$ and $x^\bullet$, respectively comprising all y such that yFx and xFy, and the *neighbourhood* ${}^\bullet x^\bullet$ of x is the union of its inputs and outputs. The dot-notations readily extend to sets of nodes, e.g., ${}^\bullet X$ comprises all inputs of the nodes in X. It is assumed here that each net is *T-restricted* which means that every transition has at least one input (cause) and at least one output (effect). For the net N in Figure 5.6, ${}^\bullet\mathtt{g} = \{\mathtt{p4}, \mathtt{p5}\}$ and $\mathtt{p3}^\bullet = \{\mathtt{m}\}$.

5.4.1 Configurations and Transition Occurrence

In this and the next section, the states of a net $N \overset{\text{df}}{=} (P, T, F)$ are given by subsets of places representing the conditions that hold at a given global situation.

Definition 16 : configurations

A configuration of a net is a subset of its places.

In diagrams, a configuration C is represented by drawing in each place p in C a token (a small black dot). A possible configuration for the net in Figure 5.6 is $\mathtt{C} = \{\mathtt{p2}, \mathtt{p5}\}$, as illustrated on the right of Figure 5.6.

Transitions represent actions which may occur at a given configuration and then lead to a new configuration.

Definition 17 : transition occurrences

A transition t can occur (or is enabled) at a configuration C if ${}^{\bullet}t \subseteq C$ and $t^{\bullet} \cap C = \varnothing$. Its occurrence then leads to a new configuration $(C \setminus {}^{\bullet}t) \cup t^{\bullet}$.

Thus a (potential) occurrence of a transition depends only on its neighbours. If t can occur at C then we write $C[t\rangle$, and if its occurrence leads to C' we write $C[t\rangle C'$. Note that through such an occurrence, all inputs of t cease to hold, and all outputs start to hold. Hence the change caused by the occurrence of a transition is always the same and does not depend on the current global state. For the configuration $\mathtt{C}$ shown in Figure 5.6, the enabled transitions are $\mathtt{r}$ and $\mathtt{a}$. Moreover, we have $\mathtt{C}[\mathtt{a}\rangle\{\mathtt{p3}, \mathtt{p4}, \mathtt{p5}\}$ and $\mathtt{C}[\mathtt{r}\rangle\{\mathtt{p1}, \mathtt{p5}\}$. Figure 5.7 provides further intuition about the enabling and occurrence rules for net transitions.

We now lift the execution of transitions to a concurrent context by allowing the simultaneous occurrence of transitions provided that they do not interfere with one another, i.e., their neighbourhoods are mutually disjoint.

Definition 18 : steps

A step of a net is a subset of its transitions. A step can occur (or is enabled) at a configuration C if the neighbourhoods of its transitions do not overlap, and each transition is enabled. The effect of its occurrence is the cumulative effect of the occurrences of the transitions it comprises.

In other words, a step U is enabled at C if ${}^{\bullet}t^{\bullet} \cap {}^{\bullet}t'^{\bullet} = \varnothing$ for all distinct transitions t and t' in U, and $C[t\rangle$ for each transition t in U. We denote this by $C[U\rangle$. The occurrence of an enabled step leads to a new configuration C' given by $(C \setminus {}^{\bullet}U) \cup U^{\bullet}$, and we denote this by $C[U\rangle C'$. Note that $C[U\rangle C$ *iff* the step U is empty $\checkmark$. For the configuration $\mathtt{C}$ shown in Figure 5.6, we have $\mathtt{C}[\{\mathtt{a}\}\rangle\mathtt{C}'$ where $\mathtt{C}' = \{\mathtt{p3}, \mathtt{p4}, \mathtt{p5}\}$; moreover, we further have:

$$\mathtt{C}'[\{\mathtt{m}, \mathtt{g}\}\rangle\{\mathtt{p2}, \mathtt{p6}\} \qquad \mathtt{C}'[\{\mathtt{m}\}\rangle\{\mathtt{p2}, \mathtt{p4}, \mathtt{p5}\} \qquad \mathtt{C}'[\{\mathtt{g}\}\rangle\{\mathtt{p3}, \mathtt{p6}\} \ .$$

We are now ready to introduce sequences of transitions and step occurrences.

Definition 19 : step sequences

A step sequence of a net is a finite sequence of non-empty steps occurring one after another from a given configuration.

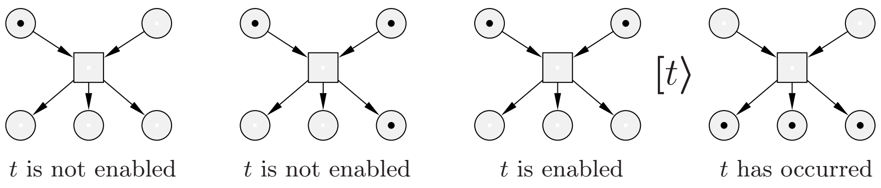


Fig. 5.7. Local change-of-state produced by the occurrence of a transition.

In other words, a step sequence from a configuration C to a configuration C' is a possibly empty sequence $\sigma = U_1 \dots U_n$ of non-empty steps U_i such that $C[U_1\rangle C_1, \dots, C_{n-1}[U_n\rangle C'$, for some configurations $C_1, \dots, C_{n-1}$. We also write $C[\sigma\rangle C'$ or $C[\sigma\rangle$, and say that C' is a configuration *reachable* from C. The set of all configurations reachable from C will be denoted by $[C\rangle$. Note that we always have $C \in [C\rangle$. If $n = 0$, thus $\sigma = \lambda$ the empty (step) sequence, then $C = C'$. The converse implication however does not hold $\checkmark$. For the configuration $\mathtt{C}$ shown in Figure 5.6, $\mathtt{C}[\{\mathtt{a}\}\{\mathtt{m},\mathtt{g}\}\{\mathtt{u},\mathtt{r}\}\rangle\{\mathtt{p1},\mathtt{p5}\}$, and, as we will see later on, the set $[\mathtt{C}\rangle$ comprises twelve reachable configurations.

To improve the readability of the notations when discussing examples, we will often drop the curly brackets when writing a singleton step, e.g., we can write $\mathtt{a}\{\mathtt{m},\mathtt{g}\}\mathtt{ur}$ instead of $\{\mathtt{a}\}\{\mathtt{m},\mathtt{g}\}\{\mathtt{u}\}\{\mathtt{r}\}$.

A special kind of step sequences are those that consist of singleton steps only. Such sequences (of transitions) are referred to as *firing sequences*. For example, $\mathtt{amgur}$ is a firing sequence from $\mathtt{C}$ to $\{\mathtt{p1},\mathtt{p5}\}$. Reachability of configurations does not depend on whether one uses step sequences or firing sequences. If, however, the structure of a net is enriched with inhibitor arcs as we will do it in the next section, then reachability may be affected by the restriction to firing sequences.

5.4.2 Concurrency and Causality

The definition of concurrent behaviour on basis of non-interference, as introduced above, allows one to investigate some intricate relationships in the way transitions can occur. As a first observation we have that transitions which can be executed simultaneously (at some configuration) do not have to occur together. They can still occur one after another. Moreover, whenever transitions can occur in any order, they must be concurrently enabled and non-interfering.

Fact 20 : Let C, C' be configurations and U, U' be steps of a net.

- $C[U \cup U'\rangle C'$ and $U \cap U' = \varnothing$ implies $C[UU'\rangle C'$.
- $C[UU'\rangle C'$ and $C[U'\rangle$ implies $U \cap U' = \varnothing$ and $C[U \cup U'\rangle C'$.

This fact is often referred to as a 'diamond property'. The reason is that if we have, say, $C[\{a,b\}\rangle C'$, it then follows that we also have $C[\{a\}\rangle C''[\{b\}\rangle C'$

and $C[\{b\}\rangle C'''[\{a\}\rangle C'$ where C'' and C''' are distinct configurations $\checkmark$. In drawing this yields a diamond shape. Note that the two statements together show that for the dynamics of nets defined sofar, diamonds imply concurrency and vice versa. For the configurations $\texttt{C}' = \{\texttt{p3}, \texttt{p4}, \texttt{p5}\}$ and $\texttt{C}'' = \{\texttt{p2}, \texttt{p6}\}$ of the net shown in Figure 5.6, we have $\texttt{C}'[\{\texttt{m}, \texttt{g}\}\rangle \texttt{C}''$ as well as $\texttt{C}'[\texttt{mg}\rangle \texttt{C}''$ and $\texttt{C}'[\texttt{gm}\rangle \texttt{C}''$, and the resulting 'diamond' can be seen with a little bit of effort at the centre of the upper state graph in Figure 5.8.

The first part of Fact 20 implies that every step of simultaneously occurring transitions can be split into any partition of subsets occurring in sequence, with the same effect as the original step. As a consequence, every step sequence eventually gives rise to a valid (but not necessarily unique) firing sequence. And so the configurations reachable from a given one are the same for step sequences and firing sequences.

Fundamental relationships between transitions can be classified in a way which reflects their causal dependence (occurrence of one enables the other), competition for shared resources (both can occur, but they cannot occur together), or concurrency (they can occur together).

Definition 21 : fundamental situations - behavioural

Let t and t' be distinct transitions, and C be a configuration of a net.

- t causally depends on t' at C if $\neg C[t\rangle$ and $C[t't\rangle$.
- t and t' are in conflict at C if $C[t\rangle$, $C[t'\rangle$ and $\neg C[\{t, t'\}\rangle$.
- t and t' are concurrent at C if $C[\{t, t'\}\rangle$.

For the configuration C shown in Figure 5.6, we have that g causally depends on a, and the latter is in conflict with r. Moreover, m and g are concurrent at the configuration $\texttt{C}' = \{\texttt{p3}, \texttt{p4}, \texttt{p5}\}$.

It is interesting to note the difference between conflict and concurrency in terms of firing sequences: in case of conflict at a configuration, both are enabled to occur, but the occurrence of one disables the other, whereas in case of concurrency, the two transitions can occur in either order.

Fact 22 : Let t and t' be transitions, and C be a configuration of a net.

- If t causally depends on t' at C then $\neg C[tt'\rangle$ and $C[t't\rangle$.
- If t and t' are in conflict at C then $\neg C[tt'\rangle$ and $\neg C[t't\rangle$.
- If t and t' are concurrent at C then $C[tt'\rangle$ and $C[t't\rangle$.

These fundamental relationships between transitions are defined dynamically by referring to a global state. However, if two transitions are in one of these three relationships at some configuration, then none of the other relationships will ever hold for them (at whatever configuration) $\checkmark$. In fact, the (potential) relationships between transitions are determined by the graph structure.

Definition 23 : fundamental situations - structural

Let t and t' be two distinct transitions of a net N.

- t and t' are structurally causally related if ${}^{\bullet}t \cap t'^{\bullet} \neq \varnothing$ or $t^{\bullet} \cap {}^{\bullet}t' \neq \varnothing$.
- t and t' are in structural backward conflict if ${}^{\bullet}t \cap {}^{\bullet}t' \neq \varnothing$.
- t and t' are in structural forward conflict if $t^{\bullet} \cap t'^{\bullet} \neq \varnothing$.
- t and t' are structurally independent if ${}^{\bullet}t^{\bullet} \cap {}^{\bullet}t'^{\bullet} = \varnothing$.

For the net shown in Figure 5.6, `a` and `g` are structurally causally related, `a` and `r` are in structural forward conflict, and `r` and `u` are structurally independent.

5.4.3 EN-Systems and Their State Spaces

Having defined nets with states and dynamics, it is now time to study them as systems which start their operation from an initial state.

Definition 24 : EN-systems

An elementary net system (or EN-system) consists of an underlying net and an initial configuration. Its state space consists of all configurations reachable from the initial configuration.

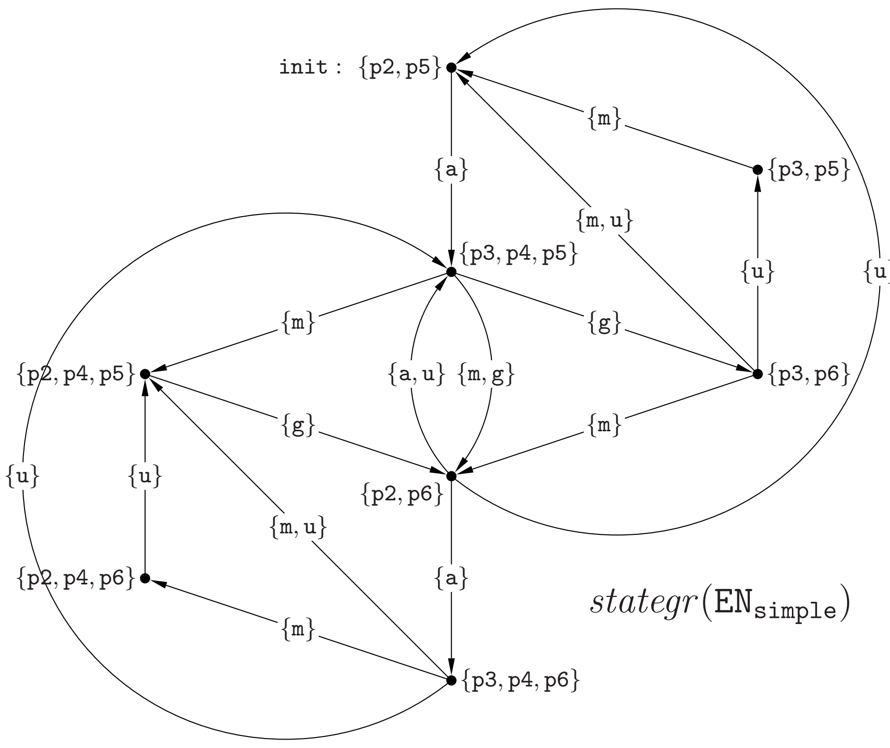
In other words, an elementary net system EN is a relational tuple (P, T, F, C_{init}) such that the first three components form its underlying net and $C_{init} \subseteq P$ is the initial configuration. Figure 5.6 shows on the right an EN-system $\mathtt{EN} = (\mathtt{P}, \mathtt{T}, \mathtt{F}, \mathtt{C_{init}})$, where $\mathtt{C_{init}} = \mathtt{C} = \{\mathtt{p2}, \mathtt{p5}\}$, modelling our running example. Its state space consists of twelve configurations:

$$[\mathtt{C_{init}}\rangle = \{\{\mathtt{pi}, \mathtt{pj}\} \mid \mathtt{i} = 1, 2, 3 \wedge \mathtt{j} = 5, 6\} \cup \{\{\mathtt{pi}, \mathtt{p4}, \mathtt{pj}\} \mid \mathtt{i} = 1, 2, 3 \wedge \mathtt{j} = 5, 6\} .$$

The *state graph* of EN is a relational tuple $stategr(EN) \overset{\text{df}}{=} ([C_{init}\rangle, LA, C_{init})$ with node set $[C_{init}\rangle$, set of labelled arcs $LA \overset{\text{df}}{=} \{(C, U, C') \mid C \in [C_{init}\rangle \wedge C[U\rangle C'\}$, and initial node C_{init}. Restricting the arcs of the state graph to those labelled by singletons steps yields the *sequential state graph* of EN, denoted by $seqstategr(EN)$. Figure 5.8 gives examples of each kind of state graph for the EN-system $\mathtt{EN_{simple}}$ in Figure 5.9.

Since every configuration reachable from the initial configuration by a step sequence is also reachable by a firing sequence, all nodes in $seqstategr(EN)$ are reachable from the initial node. Interestingly, also $stategr(EN)$ can be recovered from the sequential state graph $seqstategr(EN)$ by saturating the latter with non-singleton step labelled edges using the diamond property (Fact 20) ✓.

To illustrate the above idea, let us consider the state graphs in Figure 5.8, and two nodes, $\mathtt{C} = \{\mathtt{p3}, \mathtt{p4}, \mathtt{p6}\}$ and $\mathtt{C}' = \{\mathtt{p2}, \mathtt{p4}, \mathtt{p5}\}$. Looking at the sequential state graph, we can deduce that $\mathtt{C}[\{\mathtt{m}\}\{\mathtt{u}\}\rangle\mathtt{C}'$ and $\mathtt{C}[\{\mathtt{u}\}\rangle$. Hence, by the



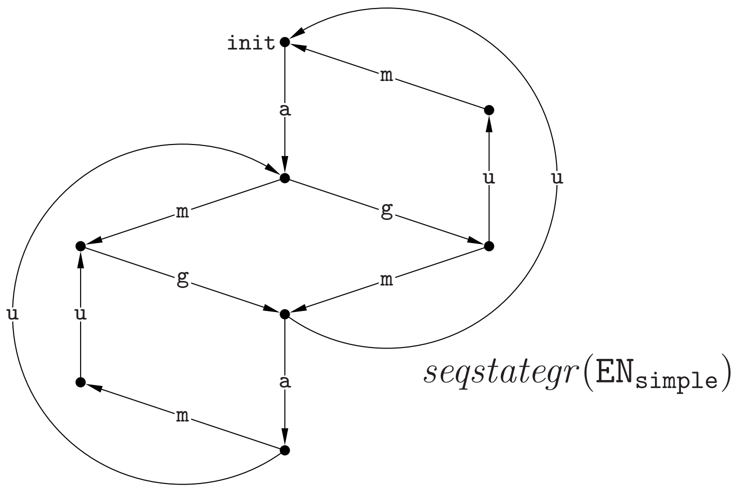


Fig. 5.8. The state graph of $\texttt{EN}_{\texttt{simple}}$ from Figure 5.9 and its sequential state graph.

second part of Fact 20, we have $\mathtt{C}[\{\mathtt{m},\mathtt{u}\}\rangle\mathtt{C}'$ and so the concurrent step $\{\mathtt{m},\mathtt{u}\}$ from $\mathtt{C}$ to $\mathtt{C}'$ in the state graph has been deduced from purely sequential information.

For a behavioural comparison of EN-systems, isomorphism is too discriminating, because then there would be essentially only one structure defining the behaviour under consideration. Therefore, in EN-system theory it is the state graph which provides the main reference point for any behaviour related analysis. However, all information on (the relevant, active, part of) the net underlying the EN-system can still be recovered from its state graph; the places belonging to reachable configurations, transitions which actually occur and thus appear in the steps labelling the arcs, and their neighbourhood relations, are all explicitly represented in the state graph. Using the state graph itself would thus lead to a similar identification of net structure and behaviour. To abstract from the concrete information on places and transitions, state graph isomorphism is used as an equivalence notion for the comparison of concurrent behaviours. Already the structure of its state graph provides a complete and faithful representation of the behaviour of an EN-system. In particular, causality, conflict, and concurrency among (possibly renamed) transitions can be determined from it. Note that two EN-systems have isomorphic state graphs *iff* also their sequential state graphs are isomorphic $\checkmark$. After isomorphism of EN-systems, state graph isomorphism is the second strongest notion of equivalence employed in the behavioural analysis of EN-systems. With this equivalence it is possible to transform EN-systems in order to realise a desired property or feature (a normal form) without affecting their dynamic properties in an essential way, i.e., the state graph remains the same up to isomorphism and the resulting system is considered behaviourally equivalent. An important application of this idea is the following.
The enabling relation for transitions checks explicitly for the emptiness of their output places. This may be regarded as somewhat unsatisfactory. It would be more efficient and intuitively more appealing if it would be sufficient to check only whether all input conditions are fulfilled.

Definition 25 : contact-freeness

An EN-system is contact-free if for every reachable configuration C and every transition t, it is the case that ${}^\bullet t \subseteq C$ implies $t^\bullet \cap C = \varnothing$.

In other words, a contact-free system is one where the test for transition enabledness can simply be ${}^\bullet t \subseteq C$ without changing anything. The EN-system shown in Figure 5.6 is <u>not</u> contact-free $\checkmark$. Not all EN-systems are contact-free, but the simple transformation described next turns any EN-system into a behaviourally equivalent contact-free version.

Two places, p and q, are *complements* of one another if ${}^\bullet p = q^\bullet$, $p^\bullet = {}^\bullet q$ and exactly one of them belongs to the initial configuration C_{init}. The *complementation* $\widetilde{EN}$ of *EN* is obtained by adding, for each place p without a

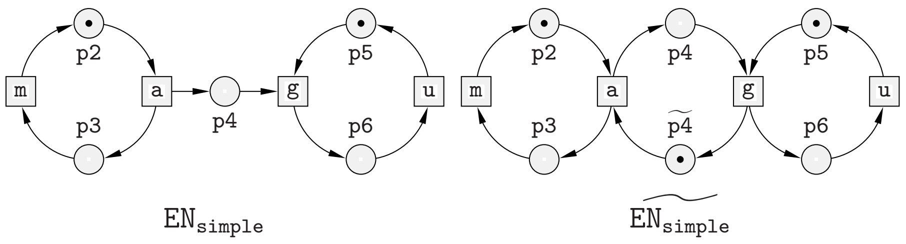


Fig. 5.9. A simplified version of the EN-system from Figure 5.6 and its complementation.

complement, a fresh complement place $\widetilde{p}$; moreover, if the initial configuration does not contain p then $\widetilde{p}$ is added there as well. The result is clearly an EN-system and the two systems have isomorphic state spaces. In fact, only the reachable configurations have to be renamed in the case that new complement places have been added; the arc labels between corresponding states however are the same ✓ .

Fact 26 : $\widetilde{EN}$ is contact-free and its state space is isomorphic to that of EN.

The construction is illustrated by the non-contact-free EN-system $\mathtt{EN_{simple}}$ in Figure 5.9 and its contact-free complementation $\widetilde{\mathtt{EN_{simple}}}$. The state spaces of the two EN-systems are respectively:

$$\begin{array}{ll} \mathtt{Conf} \cup \{\{\mathtt{pi}, \mathtt{pj}\} \mid \mathtt{i} = 2,3 \wedge \mathtt{j} = 5,6\} & \text{(left)} \\ \mathtt{Conf} \cup \{\{\mathtt{pi}, \widetilde{\mathtt{p4}}, \mathtt{pj}\} \mid \mathtt{i} = 2,3 \wedge \mathtt{j} = 5,6\} & \text{(right)} \end{array}$$

where $\mathtt{Conf} = \{\{\mathtt{pi}, \mathtt{p4}, \mathtt{pj}\} \mid \mathtt{i} = 2,3 \wedge \mathtt{j} = 5,6\}$. It is can be seen that a suitable isomorphism for their state graphs maps each $\{\mathtt{pi}, \mathtt{pj}\}$ to $\{\mathtt{pi}, \widetilde{\mathtt{p4}}, \mathtt{pj}\}$, and is the identity for the configurations in $\mathtt{Conf}$.

Fact 26 assumes that one adds complements for all non-complemented places. But it is also possible to add complementation selectively and, in general, we have that any EN-system with an arbitrary, added set of new complement places has a state space which is isomorphic to that of the original EN-system ✓ . For the EN-system EN modelling the running example we can add a complement of the buffer place which results in the equivalent EN-system shown in Figure 5.10. In this case already the selective complementation yields a contact-free EN-system ✓ .

Since it is always possible to ensure contact-freeness without changing the behaviour represented in the state-graph, we now make a simplifying assumption.

In the rest of this tutorial all EN-systems are **contact-free**.

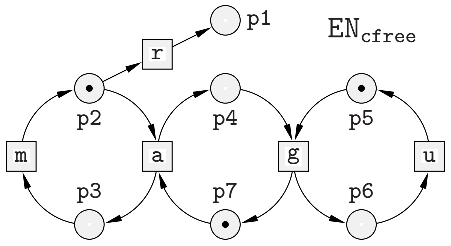


Fig. 5.10. A contact-free version of the EN-system from Figure 5.6 where the place $\texttt{p4}$ has been complemented, i.e., $\texttt{p7} = \widetilde{\texttt{p4}}$.

5.4.4 Behaviour of EN-Systems

Let $EN = (P, T, F, C_{init})$ be a fixed EN-system for the rest of this section.

In addition to the state graph, we can also associate firing sequences and step sequences as behavioural notions to EN-systems. The set of all firing sequences $\mathit{firseq}(EN)$ of EN consists of those sequences $u \in T^*$ such that $C_{init}[u\rangle$ and, similarly, the set of all step sequences $\mathit{stepseq}(EN)$ of EN comprises all step sequences of EN from C_{init}. Each firing sequence corresponds to a finite labelled path through the sequential state graph from the initial node. Since the set of reachable configurations of an EN-system is finite, the sequential state graph is a finite state machine. Hence the set of firing sequences of an EN-system is a prefix-closed regular language. However, it consists of purely sequential observations of the EN-system's behaviour without any reference to the possible independence of transitions. Yet such causality information is often of high importance for system analysis and design.

Let us first demonstrate how the theory of traces can be applied to extract partial orders from firing sequences as representations of the *necessary* causal ordering of transition occurrences within these sequences.

Definition 27 : concurrency alphabets of EN-systems

The concurrency alphabet of EN is $CA_{EN} \overset{\text{df}}{=} (T, Ind_{EN})$ where the structural independence relation Ind_{EN} comprises all pairs of distinct transitions with disjoint neighbourhoods.

Defined in this way, $Ind_{EN} = \{(t, t') \mid t, t' \in T \wedge {}^\bullet t^\bullet \cap {}^\bullet t'^\bullet = \varnothing\}$ is a symmetric and irreflexive relation and so it is indeed an independence relation. For the EN-system $\texttt{EN}_{\texttt{cfree}}$ in Figure 5.10, $\texttt{Ind}_{\texttt{EN}_{\texttt{cfree}}} = \texttt{Ind}$ where $\texttt{Ind}$ was defined at the beginning of Section 5.3. An important observation is now that in a firing sequence adjacent occurrences of independent transitions could have occurred also in the other order (see the diamond property, Fact 20). Hence, for every firing sequence of EN, all its trace equivalent words from T^* are also firing sequences of EN.

Fact 28 : $\mathit{firseq}(EN) = \bigcup_{u \in \mathit{firseq}(EN)} [u]$.

Taking, for example, $\mathtt{EN_{cfree}}$ in Figure 5.10, we have $\mathtt{agm} \in \mathit{firseq}(\mathtt{EN_{cfree}})$ and $[\mathtt{agm}] = \{\mathtt{agm}, \mathtt{amg}\}$. Clearly, $\mathtt{amg}$ is also a firing sequence of $\mathtt{EN_{cfree}}$.

The step sequences of an EN-system obviously provide important insights into concurrency aspects of its behaviour. They are nevertheless still sequential rather than concurrent in nature in the sense that the sequential ordering of the steps obscures the true causal dependencies between the occurrences of transitions. Petri net models can however easily support a formal approach where this information is readily available by unfolding behaviours into structures allowing an explicit representation of causality and concurrency.

5.4.5 Non-Sequential Observations

Rather than describing the behaviour of the system in terms of sequential observations, like firing sequences and step sequences, we now present a semantics based on a class of acyclic Petri nets, called *occurrence nets*. What one essentially tries to achieve here is to record the changes of configurations due to transitions being executed along some legal behaviour of the EN-system, and in doing so record which places were emptied (served as inputs) and which filled (as outputs). The resulting occurrence nets may be viewed as partial net unfoldings, with each transition representing an occurrence of a transition in the original net (thus occurrence nets are acyclic), and each place corresponding to the occurrence of a token on a place of the original net. Conflicts between transitions are resolved and thus the places in an occurrence net do not branch.

Definition 29 : occurrence nets

An occurrence net is a relational tuple $ON \overset{\mathrm{df}}{=} (B, E, R, \ell)$ such that (B, E, R) is an underlying net,[a] ℓ is a labelling for $B \cup E$, R is an acyclic flow relation, and $|{}^\bullet b| \leq 1$ and $|b^\bullet| \leq 1$, for every $b \in B$.

[a] The dot-notations, configurations, firing rule, etc, for ON are as those defined for the underlying net.

The places of an occurrence net are usually called *conditions* (‘Bedingungen’ in German) and its transitions *events* (‘Ereignisse’ in German). The default *initial configuration* of ON consists of all conditions without incoming arcs, i.e., C^{ON}_{init} comprises all conditions $b \in B$ such that ${}^\bullet b = \varnothing$, and the default

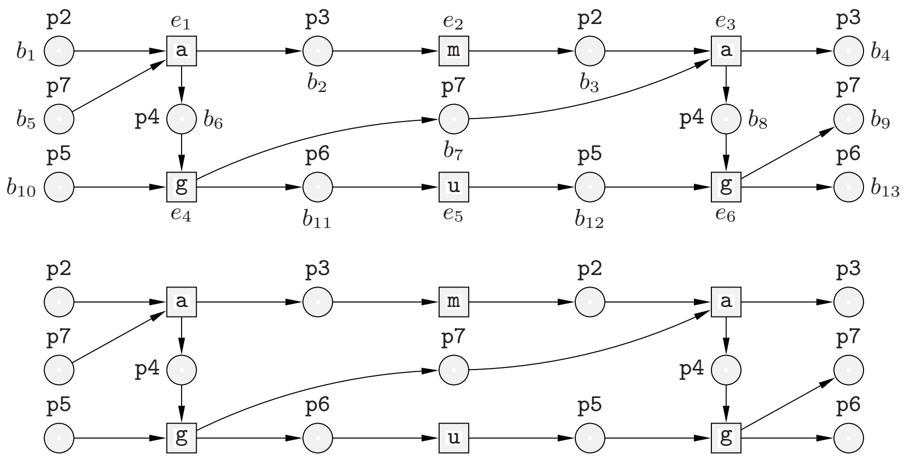


Fig. 5.11. An occurrence net ON with nodes labelled by places and transitions of the EN-system $\mathtt{EN_{cfree}}$ in Figure 5.10 (top), and the same occurrence net with identities of the nodes omitted (bottom).

final configuration C^{ON}_{fin} consists of all conditions without outgoing arcs. The default initial configuration of the occurrence net in Figure 5.11 is $C^{\mathtt{ON}}_{init} = \{b_1, b_5, b_{10}\}$ and the default final configuration is $C^{\mathtt{ON}}_{fin} = \{b_4, b_9, b_{13}\}$.

The sets of firing and step sequences of ON are defined w.r.t. the default initial configuration. However, since an occurrence net is meant to represent a record of a concurrent run of an EN-system, what really counts is not the identities of its events, but their labels which are linked to the occurrences of transitions in the EN-system. The *language* of ON is the set $language(ON)$ of all sequences $\ell(u)$ such that u is a firing sequence from the default initial configuration of ON to the default final configuration.

By abstracting from the conditions we associate with the occurrence net $ON = (B, E, R, \ell)$ a directed acyclic graph with E as its set of nodes. This dag $dag(ON) \overset{\text{df}}{=} (E, R \circ R|_{E \times E}, \ell|_E)$ represents the direct causal relationships between the events. Its transitive closure $dag(ON)^+$, see Figure 5.12, then gives all, direct and indirect, causal dependencies. For example, e_4 directly causes e_5, but there is only an indirect causal link from e_4 to e_6.

ON with its default initial configuration is basically a contact-free EN-system ✓. Interestingly, all the sets occurring in any step sequence σ from the initial configuration to another configuration C, are mutually disjoint ✓. Moreover, C is the default final configuration *iff* the steps in σ use all the events of the occurrence net ✓.

A *slice* of ON is a maximal (w.r.t. set inclusion) subset S of events from ON which are causally unrelated, i.e., $(S \times S) \cap R^+ = \varnothing$. The set of all slices of ON is denoted by $slices(ON)$. Clearly, both default configurations are slices and, in general, $[C^{ON}_{init}\rangle = slices(ON)$, i.e., slices are exactly those configurations which are reachable from the initial configuration ✓. Moreover, the

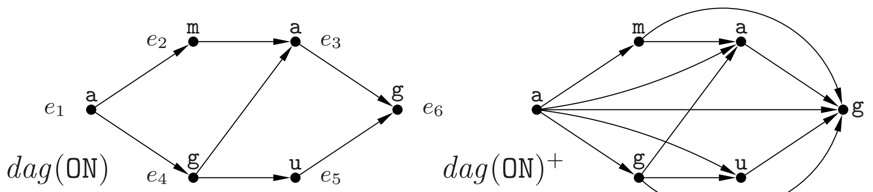


Fig. 5.12. Direct causality among the events in the occurrence net in Figure 5.11, and full causality (node identities omitted).

final configuration of ON is always reachable from any configuration reachable from the initial one $\checkmark$. Essentially, this means that ON is deadlock-free until its final configuration has been reached.

The processes of an EN-system are occurrence nets reflecting its structure and possible behaviour through their labelling and initial configuration.

Definition 30 : processes of EN-systems

A process of EN is an occurrence net $ON = (B, E, R, \ell)$ such that:

- ℓ labels conditions with places and events with transitions.
- ℓ is injective on the default initial configuration of ON, as well as on the sets of input and output conditions of each event.
- $\ell(C^{ON}_{init}) = C_{init}$ and, for every $e \in E$, $\ell({}^\bullet e) = {}^\bullet\ell(e)$ and $\ell(e^\bullet) = \ell(e)^\bullet$.

The occurrence net ON in Figure 5.11 is a process of the EN-system in Figure 5.10.

Processes can be used to investigate the behaviours of EN-systems. Due to the second and third conditions in Definition 30, we can relate the firing sequences, step sequences and configurations of EN to their labelled versions in ON. More precisely, if we take a step sequence $C^{ON}_{init}[\sigma\rangle C$ then $C_{init}[\ell(\sigma)\rangle\ell(C)$ holds. This can be proved by an inductive argument from which it also follows that labelling of ON is injective on all its slices and hence also on the sets occurring in any step sequence of ON $\checkmark$. If σ is a step sequence from the default initial configuration of ON, then $\ell(\sigma)$ is referred to as a *labelled step sequence* of ON. Similar to the language of ON, the *step language* of ON is defined as the set *steplanguage*(ON) of all sequences $\ell(\sigma)$ such that σ is a step sequence from the default initial configuration of ON to the default final configuration.

In general, it follows that all firing and step sequences of EN-systems can be derived from their processes.

Fact 31 : Let $\mathcal{ON}$ be the set of all processes of EN.

- $\mathit{firseq}(EN) = \bigcup_{ON \in \mathcal{ON}} \mathit{language}(ON)$.
- $\mathit{stepseq}(EN) = \bigcup_{ON \in \mathcal{ON}} \mathit{steplanguage}(ON)$.

Definition 30 does not provide any clues as to how to derive a process of an EN-system. This is rectified in the next definition which shows how to construct a process corresponding to a given step sequence.

Definition 32 : processes construction

The occurrence net ON_σ generated by a step sequence $\sigma = U_1 \dots U_n$ of EN is the last element in the sequence $N_0, \dots, N_n$ where each N_k is an occurrence net (B_k, E_k, R_k, ℓ_k) constructed thus.
Step 0: $B_0 \overset{\text{df}}{=} \{p^1 \mid p \in C_{init}\}$ and $E_0 = R_0 \overset{\text{df}}{=} \varnothing$.
Step k: Given N_{k-1} we extend the sets of nodes and arcs as follows:

$$\begin{aligned} B_k &\overset{\text{df}}{=} B_{k-1} \cup \{p^{1+\triangle p} \mid p \in U_k^\bullet\} \\ E_k &\overset{\text{df}}{=} E_{k-1} \cup \{t^{1+\triangle t} \mid t \in U_k\} \\ R_k &\overset{\text{df}}{=} R_{k-1} \cup \{(p^{\triangle p}, t^{1+\triangle t}) \mid t \in U_k \wedge p \in {}^\bullet t\} \\ &\qquad\quad \cup \{(t^{1+\triangle t}, p^{1+\triangle p}) \mid t \in U_k \wedge p \in t^\bullet\} \,. \end{aligned}$$

In the above, the label of each node x^i is set to be x, and $\triangle x$ denotes the number of nodes of N_{k-1} labelled by x.

The construction is illustrated in Figure 5.13 for the $\mathtt{EN_{cfree}}$ in Figure 5.10 and its step sequence $\sigma = \mathtt{a}\{\mathtt{m},\mathtt{g}\}\{\mathtt{a},\mathtt{u}\}\mathtt{g}$. The resulting occurrence net is isomorphic to the occurrence net $\mathtt{ON}$ in Figure 5.11 which is a process of $\mathtt{EN_{cfree}}$.

Fact 33 : Each occurrence net constructed as in Definition 32 is a process of EN and, for each process of EN, there is a run of the construction from Definition 32 generating an isomorphic occurrence net.

Thus the operationally defined processes and the axiomatically defined processes of an EN-system are essentially the same.

Finally, we return to the trace semantics of EN-systems in relation to processes. First note that each trace gives rise to only one process, since interchanging adjacent occurrences of independent transitions has no effect on the construction of a process. So, $ON_u = ON_w$ whenever u and w are trace equivalent firing sequences $\checkmark$. Hence $ON_{[u]}$ the process associated to a trace is a well-defined notion. Conversely, the language of a process is identical to its defining trace $\checkmark$. Thus we have a one-to-one correspondence between traces and the processes of an EN-system. Moreover, even though the dag

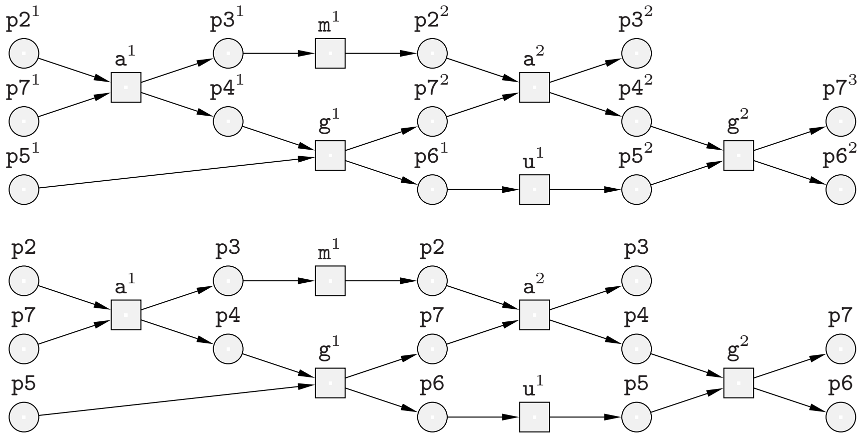


Fig. 5.13. The occurrence net $\mathtt{ON}_{\mathtt{a\{m,g\}\{a,u\}g}}$ generated for the EN-system in Figure 5.10: node-oriented view (top), and label-oriented view (bottom).

defined by a process is not necessarily isomorphic to the dependence graph of its trace ✓, they always define the same partial order on their transition occurrences.

Fact 34 : Let u be a firing sequence of EN.

- $[u] = language(ON_u)$.
- $canposet([u]) = dag(ON_u)^+$.

To conclude, the trace semantics and the process semantics of EN-systems lead to one partial order semantics by providing for each EN-system the same (isomorphic) partial orders modelling the causalities in its concurrent executions. This provides a strong argument in favour of the view that both these approaches capture the essence of causality in the behaviours of EN-systems.

5.4.6 Bibliographical Remarks

Over the past 40 or so years different classes of Petri nets have been introduced by varying the kind of underlying net, notion of local state, or transition relation. An early systematic treatment of basic notions in net theory and EN-systems can be found in [48]. Other extensions of the EN-systems approach adopt notions like priorities, real-time behaviour, or object-orientation. (In fact, we consider two such extensions later in this tutorial.) The general question of the intrinsic or common properties of nets is discussed in [8]. The problem of associating non-sequential semantics with Petri nets is dealt with, in particular, in [35, 40, 36, 37, 41, 42, 19, 47]. There is a systematic way of dealing with process semantics of various classes of Petri nets proposed in [31] which makes it possible to separately discuss behaviour, processes, causality,

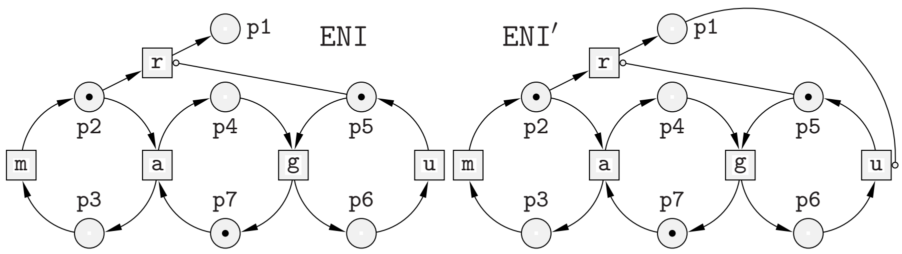


Fig. 5.14. Two ENI-systems modelling two variations of the running example.

and their mutually consistency. General Petri net related resources can be found in the web pages at [26].

5.5 Adding Inhibitor Arcs

This section extends the treatment of concurrency considered so far in EN-systems in order to accommodate the practically relevant case of nets with inhibitor arcs. In particular, we will demonstrate how the original definition of traces may be extended to describe in an adequate way also the additional features of the resulting new kind of concurrent behaviours.

To see why inhibitor arcs can be a convenient modelling device, let us imagine that a designer would like to modify the running example so that the producer cannot retire if the customer is waiting for an item. Such a modification is easily achieved by taking the EN-system of Figure 5.10 and adding to it an inhibitor arc linking the place `p5` and transition `r`. This yields the net system `ENI` shown on the left of Figure 5.14. (Inhibitor arcs are drawn with small open circles as arrowheads.) Adding this arc means that `r` cannot be enabled if `p5` contains a token, and so the producer indeed cannot retire if the consumer is waiting for an item. Elementary net systems with inhibitor arcs, or simply ENI-systems, thus extend EN-systems. The usefulness of inhibitor arcs stems from their ability to detect a lack rather than the presence of specific resources, i.e., tokens in specific places. That such an addition to the EN-system syntax is a true extension of their modelling power follows from the observation that there is no EN-system with exactly the same set of firing sequences as `ENI`. This can be shown by considering two firing sequences of `ENI`, `amgru` and `amgu`. If there was an EN-system generating the same firing sequences as `ENI`, then, due to the second statement in Fact 20, it would also have to generate the firing sequence `amgur`. But such a firing sequence is not generated by `ENI` as executing the last transition would contradict the defining characteristic of the inhibitor arc between `r` and `p5`. We will return to this example after introducing ENI-systems more formally.

Definition 35 : ENI-systems

An elementary net system with inhibitor arcs (or ENI-system) is a relational tuple $ENI \overset{\text{df}}{=} (P, T, F, C_{init}, Inh)$ such that the first four components form an (underlying) EN-system and $Inh \subseteq P \times T$ is a set of inhibitor arcs.

As inhibitor arcs are introduced on top of the model of EN-systems, wherever it is possible notions and notations concerning the structure and configurations of an ENI-system are inherited from its underlying EN-system. Thus, for example, the initial configuration of the ENI-systems in Figure 5.14 is the initial configuration of the EN-system in Figure 5.10. The only new notation is $^\circ t$ denoting the set of all the places p where the presence of a token inhibits the enabling of a transition t, i.e., $(p, t) \in Inh$. For example, we have $^\circ\mathtt{r} = \{\mathtt{p5}\}$ and $^\circ\mathtt{a} = {}^\circ\mathtt{m} = {}^\circ\mathtt{g} = {}^\circ\mathtt{u} = \varnothing$ in the case of `ENI`.

The dynamic aspects of an ENI-system are also derived from the underlying EN-system, with proper attention being paid to the inhibiting features of the new kind of arcs. In fact, all one needs to re-define is the enabling condition for steps, by stating that a step of transitions U of an ENI-system is enabled at a configuration C if it is enabled at C in the underlying EN-system and, in addition, no place in $^\circ U$ belongs to C, where $^\circ U$ consists of all places connected by inhibitor arcs to transitions in U. It is important here to stress that the change of state effected by an executed step is exactly the same as in the underlying EN-system; in other words, inhibitor arcs have only impact on the enabling of steps. In the case of `ENI`, `a` is a singleton step enabled in the initial configuration, but the other singleton step `r` enabled in the initial configuration of the underlying EN-system is not since $\mathtt{p5} \in {}^\circ\mathtt{r} \cap \mathtt{C_{init}}$. Note that it would be quite natural and harmless as it has no effect on the dynamics of an ENI-system to additionally assume that for each of its transitions t, the sets $^\bullet t$, $t^\bullet$ and $^\circ t$ are mutually disjoint ✓. As far as executing step sequences of `ENI` are concerned, we have $\mathtt{C_{init}}[\mathtt{a}\rangle\{\mathtt{p3}, \mathtt{p4}, \mathtt{p5}\}$ and $\mathtt{C_{init}}[\mathtt{a}\{\mathtt{m}, \mathtt{g}\}\rangle\{\mathtt{p2}, \mathtt{p6}, \mathtt{p7}\}$.

Having introduced the step sequence semantics of ENI-systems, we have another look at `ENI`. This ENI-system generates the step sequence $\sigma_{\mathtt{exmpl}} \overset{\text{df}}{=} \mathtt{a}\{\mathtt{m}, \mathtt{g}\}\{\mathtt{u}, \mathtt{r}\}$ Splitting the last step into `ur` leads to the sequence $\sigma_{\mathtt{invalid}} \overset{\text{df}}{=} \mathtt{a}\{\mathtt{m}, \mathtt{g}\}\mathtt{ur}$ which is not a valid behaviour of `ENI` (yet the splitting of $\{\mathtt{u}, \mathtt{r}\}$ into `ru` leads to a valid step sequence). Thus, also the first part of Fact 20 does not, in general, hold for ENI-systems. In fact, not only the diamond property no longer holds for ENI-systems, but even the property that every step sequence of an EN-system can be linearised to yield some valid firing sequence is not true for ENI-systems. Consider, for example, the ENI-system `ENI′` on the right of Figure 5.14 which has been obtained from `ENI` by adding an inhibitor arc between `p1` and `u` ensuring that the consumer can only use an item if there is still a chance that the producer may produce another item in the future. It is easy to see that `ENI′` generates the step sequence $\mathtt{amg}\{\mathtt{u}, \mathtt{r}\}$, but any attempt to linearise its only non-singleton step, `ur`, results in an illegal behaviour.

Note that $\mathtt{ENI}'$ does not even have a firing sequence leading to a configuration including both `p1` and `p5`. Hence the reachability of configurations is affected by the restriction to firing sequences, and so ordinary words are insufficient to capture all potential behaviours of ENI-systems. Consequently, the generalisation of trace theory we will present next will be based on step sequences rather than on words.

5.5.1 Comtraces

We will now show how the notions of independence and causality developed for EN-systems can be lifted to the level of ENI-systems. The concurrency model used for EN-systems is not directly applicable to nets with inhibitor arcs; in particular, the current notion of transition independence needs to be replaced by a device which can be used to disallow some linearisations of executed steps. We therefore start by modifying the notion of concurrency alphabet.

Definition 36 : combined concurrency alphabets

A combined concurrency alphabet is a triple $CCA \overset{\mathrm{df}}{=} (\Sigma, sim, ser)$ where Σ is an alphabet and $ser \subseteq sim$ are binary relations over Σ called respectively simultaneity and serialisability. It is assumed that sim is irreflexive and symmetric.

The two relations in a combined concurrency alphabet serve two distinct purposes, one of which is to define valid steps and the other is to define valid ways of splitting such steps. More precisely, if $(a, b) \in sim$ then a and b may occur together in a step, while $(a, b) \in ser$ means that a and b may occur in a step $\{a, b\}$ and, in addition, such a step can be split into the sequence $\{a\}\{b\}$.

Although it may not be immediately clear, combined concurrency alphabets subsume concurrency alphabets used earlier on. More precisely, the independence relation Ind of concurrency alphabets used in the definition of traces represents the situation that simultaneity and serialisability coincide with concurrency, i.e., $sim = ser = Ind$ (note that this implies $ser = ser^{-1}$ and that the two relations define diamonds). Intuitively, this means that simultaneity of symbols implies that they are totally independent and executing one has no impact on the subsequent executability of the other. In the examples, the combined concurrency alphabet `CCA` corresponds to the running ENI-system example, i.e., `ENI` in Figure 5.14. Thus $\mathtt{CCA} = (\Sigma, \mathtt{sim}, \mathtt{ser})$, where Σ is as before, $\mathtt{sim} = \{(\mathtt{r}, \mathtt{u}), (\mathtt{u}, \mathtt{r}), (\mathtt{g}, \mathtt{m}), (\mathtt{m}, \mathtt{g}), (\mathtt{u}, \mathtt{m}), (\mathtt{m}, \mathtt{u}), (\mathtt{u}, \mathtt{a}), (\mathtt{a}, \mathtt{u})\}$ and $\mathtt{ser} = \mathtt{sim} \setminus \{(\mathtt{u}, \mathtt{r})\}$. As we will later see, these relations can be derived from the structure of `ENI`. Note that the problem with the step sequence $\sigma_{\mathtt{invalid}}$ above is addressed by excluding $(\mathtt{u}, \mathtt{r})$ from `ser`.

Next, the notion of a step is extended so that it does not necessarily depend on a net, but may also be defined relative to a combined concurrency

alphabet $CCA = (\Sigma, sim, ser)$ by stating that any non-empty set $U \subseteq \Sigma$ is a *step* (over CCA) if $(a, b) \in sim$ for all distinct a and b in U. (If sim and ser are not relevant we silently assume that $sim = (\Sigma \times \Sigma) \setminus id_\Sigma$ and $ser = sim$; in that case every finite sequence of non-empty subsets of Σ is a step sequence over Σ.)
We then lift in the obvious way to step sequences the following notions previously defined for words: concatenation, the set of symbol occurrences, and what it means for the i-th occurrence of a to precede the j-th occurrence of b within a step sequence. Note that neither the i-th occurrence of a precedes the j-th occurrence of b, nor the j-th occurrence of b precedes the i-th occurrence of a, if the two occurrences belong to the same step. Taking as an example the combined concurrency alphabet $\mathtt{CCA}$ defined above, we have that $\mathtt{a\{m,g\}am\{u,r\}}$ is a step sequence where $\mathtt{m}^1$ precedes $\mathtt{a}^2$ and the latter symbol occurrence precedes $\mathtt{m}^2$.

We can now introduce *comtraces* generalising traces and based on a combined concurrency alphabet. When defining the trace equivalence relation, we were able to swap any pair of neighbouring occurrences of independent symbols, e.g., $\mathtt{amu} \equiv \mathtt{aum}$. A similar effect can be achieved using the extended concurrency alphabet, but with an additional intermediate phase where the symbols being swapped are put together into a single step which is then linearised. Thus the elementary transformation needed to define comtraces is step splitting and combining rather than symbol swapping. To this end, we introduce $\backsimeq$ which is a relation comprising all pairs (σ, ρ) of step sequences such that $\sigma = \tau U \chi$ and $\rho = \tau U' U'' \chi$ where τ, χ are possibly empty step sequences, and U', U'' form a partition of U such that $U' \times U'' \subseteq ser$. Then we define the *comtrace equivalence* $\approxeq$ to be the reflexive symmetric transitive closure of $\backsimeq$.

Coming back to the trace equivalence $\mathtt{amu} \equiv \mathtt{aum}$ and assuming the combined concurrency alphabet of the running example, we have $\mathtt{amu} \approxeq \mathtt{aum}$ which follows from $\mathtt{amu} \backsimeq^{-1} \mathtt{a\{m,u\}} \backsimeq \mathtt{aum}$. Another example of comtrace equivalence is $\mathtt{a\{m,g\}\{u,r\}} \approxeq \mathtt{a\{m,g\}ru}$, but we also have that $\mathtt{a\{m,g\}\{u,r\}} \not\backsimeq \mathtt{a\{m,g\}ur}$ since $(\mathtt{u},\mathtt{r}) \notin \mathtt{ser}$. In fact, $\mathtt{a\{m,g\}\{u,r\}}$ and $\mathtt{a\{m,g\}ur}$ are not comtrace equivalent step sequences.

Definition 37 : comtraces

A comtrace over a combined concurrency alphabet is any equivalence class of its comtrace equivalence relation.

A comtrace containing a given step sequence σ will be denoted $\langle\sigma\rangle$. Note that $\langle\lambda\rangle = \{\lambda\}$ is the empty comtrace. The comtrace comprising the step sequence $\sigma_{\mathtt{exmpl}}$ is made up of the following six step sequences: $\mathtt{a\{m,g\}\{u,r\}}$, $\mathtt{a\{m,g\}ru}$, $\mathtt{amgru}$, $\mathtt{agmru}$, $\mathtt{amg\{u,r\}}$ and $\mathtt{agm\{u,r\}}$.

Comtraces enjoy a number of the key properties satisfied by traces, and so certain notions introduced for the latter can be re-defined for comtraces.

To start with, $\sigma\rho$ and $\sigma'\rho'$ are comtrace equivalent whenever $\langle\sigma\rangle = \langle\sigma'\rangle$ and $\langle\rho\rangle = \langle\rho'\rangle$ ✓. Hence comtrace concatenation $\langle\sigma\rangle \odot \langle\rho\rangle \overset{\mathrm{df}}{=} \langle\sigma\rho\rangle$ is a well-defined operation, and we can recover the monoidal structure of traces.

Fact 38 : The set of all comtraces over a combined concurrency alphabet with comtrace concatenation and the empty comtrace forms a monoid.

Finally, similarly as it was done for traces, comtraces can be equipped with a prefix relation which reflects their possible histories ✓.

5.5.2 Stratified Posets and Comdags

Traces have posets as their underlying dependency structures. For comtraces however, we will need to provide another notion of causal dependence. We start by providing a characterization of step sequences as a rather specific kind of posets, similar to the way that total posets correspond to words.

Definition 39 : stratified posets

A poset is stratified if being an unordered pair of elements is a transitive relation, and all elements labelled with the same label are linearly ordered.

In other words, a poset $spo = (X, \prec, \ell)$ is stratified if its elements can be partitioned into non-empty sets $X_1, \ldots, X_k$ such that ℓ is injective on each of them and the precedence relation is equal to the union of sets $X_i \times X_j$, for all $i < j$. This further implies that the unorderedness relation $\frown_{spo}$ is equal to the union of sets $X_i \times X_i \setminus id_{X_i}$, for all i, and that $\frown_{spo} \cup id_X$ is an equivalence relation ✓. Since the partitioning of X into these X_i's is unique, one can associate with *spo* the step sequence $steps(spo) \overset{\mathrm{df}}{=} \ell(X_1) \ldots \ell(X_n)$, and thus it is possible to view a stratified poset as a step sequence. The converse move is also possible, and the definition resembles that of the canonical total poset of a word. The *canonical* stratified poset $canstratposet(\sigma)$ of a step sequence σ is defined as $(occ(\sigma), \prec, \ell)$ where $a^i \prec b^j$ if the i-th occurrence of a precedes the j-th occurrence of b within σ, and $\ell(a^i) \overset{\mathrm{df}}{=} a$, for all symbol occurrences a^i and b^j in $occ(u)$. Figure 5.15 shows the canonical stratified poset of the step sequence $\sigma_{\mathtt{exmpl}}$. One can immediately note that $steps(canstratposet(\sigma_{\mathtt{exmpl}})) = \sigma_{\mathtt{exmpl}}$ and since this is a general property holding for any σ, step sequences can be identified with the corresponding stratified order ✓.

We will also need structures generalising dependence graphs. Recall that these dags result from total posets (words) by deleting some of the precedence relationships between elements. Similarly, the new structures can be interpreted as stratified posets from which certain relationships have been deleted, while taking into account that the simultaneity within the steps also defines a weak (mutual) dependency between elements that can be deleted

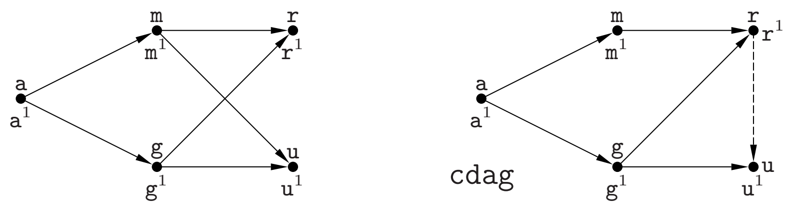


Fig. 5.15. Hasse diagram of the canonical stratified poset for $\sigma_{\mathtt{exmpl}}$, and a comdag.

(in either direction). The main idea here is that standard causal precedence captures the 'happened before' relationship and a new weak causality relation stands for 'happened before or simultaneously'.

Definition 40 : comdags

A labelled directed acyclic combined graph (or simply comdag) is a relational tuple *comdag* $\overset{\text{df}}{=} (X, \prec, \sqsubset, \ell)$ consisting of a finite set X, two irreflexive binary relations over X, $\prec$ and $\sqsubset$, and a labelling of X such that:

- $\prec' \overset{\text{df}}{=} (\prec \cup \sqsubset)^* \circ \prec \circ (\prec \cup \sqsubset)^*$ is an irreflexive relation.
- $\ell(x) = \ell(y)$ implies $x \prec' y$ or $y \prec' x$ for all $x \neq y$ in X.

The irreflexivity of the relation $\prec'$ above has a straightforward interpretation in operational terms, as it means that in a given run of a concurrent system there are no events $x_1, x_2, \ldots, x_k$ such that each x_i 'happened before or simultaneously' with x_{i+1}, while x_k 'happened (strictly) before' x_1. Comdags with an empty relation $\sqsubset$ are nothing but dags, and we adopt similar conventions for their graphical representation: the relation $\prec$ is represented by solid arcs, and $\sqsubset$ by dashed arcs. For example, Figure 5.15 shows a comdag `cdag` such that $\mathtt{a}^1 \prec \mathtt{m}^1 \prec \mathtt{r}^1$, $\mathtt{a}^1 \prec \mathtt{g}^1 \prec \mathtt{u}^1$, $\mathtt{g}^1 \prec \mathtt{u}^1$, and $\mathtt{r}^1 \sqsubset \mathtt{u}^1$.

A full account of the causal dependencies between the nodes of a comdag *comdag* $= (X, \prec, \sqsubset, \ell)$ is conveyed through its transitive closure, defined as the relational structure $comdag^+ \overset{\text{df}}{=} (X, \prec', \sqsubset', \ell)$ where $\prec'$ is the relation introduced in Definition 40 and $\sqsubset' \overset{\text{df}}{=} (\prec \cup \sqsubset)^* \backslash id_X$. The transitive closure of the comdag in Figure 5.15 is shown on the left of Figure 5.16.

5.5.3 Stratified Order Structures

Traces are underpinned by posets which in their turn are transitive dags. Similarly, the structures underlying comtraces are transitive comdags, called stratified order structures.

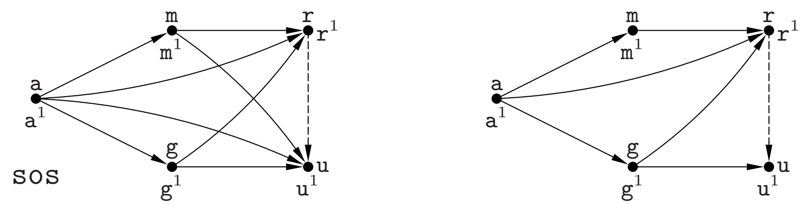


Fig. 5.16. A stratified order structure where dashed arcs between nodes have been omitted if solid arcs are present, and the canonical dependence comdag for $\sigma_{\texttt{exmpl}}$.

Definition 41 : stratified order structures

A comdag $sos \overset{\text{df}}{=} (X, \prec, \sqsubset, \ell)$ is a labelled stratified order structure (or so-structure) if for all x, y, z in X:

(i) $x \prec y$ implies $x \sqsubset y$.
(ii) $x \sqsubset y \sqsubset z$ and $x \neq z$ implies $x \sqsubset z$.
(iii) $x \sqsubset y \prec z$ or $x \prec y \sqsubset z$ implies $x \prec z$.

The first relation in an so-structure should be interpreted as the standard *causality*, and the second relation as *weak causality*.

In Figure 5.16, $\texttt{a}^1$ causally precedes $\texttt{u}^1$, while $\texttt{r}^1$ precedes $\texttt{u}^1$ only in a weakly causal manner. The latter means that $\texttt{r}^1$ may occur before or simultaneously with $\texttt{u}^1$. Observe that the so-structure $\texttt{sos}$ in Figure 5.16 is the transitive closure of the comdag in Figure 5.15, i.e., $\texttt{sos} = \texttt{cdag}^+$.

The transitive closure of a comdag is an so-structure, and the transitive closure of an so-structure is the same so-structure ✓. Moreover, given an so-structure $(X, \prec, \sqsubset, \ell)$ and a pair of elements $x, y \in X$, $x \prec y$ implies $y \not\sqsubset x$, and $(X, \prec, \ell)$ is a poset ✓. That so-structures are conservative extensions of posets follows from the fact that if $(X, \prec, \ell)$ is a poset then $(X, \prec, \prec, \ell)$ is an so-structure ✓. Hence so-structures may be viewed as generalisations of posets.

In the case of posets, we considered their total poset linearisations (corresponding to words). In the current framework, posets have been replaced by so-structures and, accordingly, stratified order structures can be extended to stratified posets (corresponding to step sequences).

A stratified poset *spo* is a *stratification* of an so-structure *sos* if they have the same domain and labelling, $\prec_{sos}$ is included in $\prec_{spo}$, and $\sqsubset_{sos}$ is included in $\preccurlyeq_{spo}$. The set of all stratifications of *sos* is denoted by *strat*(*sos*).

The *intersection* $\bigcap \mathcal{SPO}$ of a non-empty set $\mathcal{SPO}$ of stratified posets with the same domain X and labelling ℓ is $(X, \prec, \sqsubset, \ell)$ where $\prec$ is the relation comprising all pairs (x, y) such that $x \prec_{spo} y$ for each *spo* in $\mathcal{SPO}$, and $\sqsubset$ is a relation comprising all pairs (x, y) such that $x \preccurlyeq_{spo} y$ for each *spo* in $\mathcal{SPO}$. The intersection of stratified posets is always a stratified order structure ✓. Moreover, an so-structure is completely identified by its stratification

$sos = \bigcap strat(sos)$ and $strat(sos)$ is a non-empty set ✓. It is interesting that the result would not hold if we restricted ourselves only to those stratifications which are total posets ✓.

The *step language* of a comdag comprises all step sequences associated with the stratifications of its transitive closure, i.e., we define $steplanguage(comdag) \overset{\text{df}}{=} steps(strat(comdag^+))$. For the comdag in Figure 5.15 and the so-structure in Figure 5.16, $steplanguage(\texttt{cdag}) = steplanguage(\texttt{sos})$ comprises exactly the same six step sequences as the comtrace to which the step sequence $\sigma_{\texttt{exmpl}}$ belongs. This is not a mere coincidence, as we will soon see.

5.5.4 Causality Structures Generated by Comtraces

Comdags can be used to describe the necessary ordering (causality and weak causality) in comtraces. This relationship and the way it is derived strongly resemble what has been done earlier on for traces and their associated dags (dependence graphs). Let us first characterise the comdags which are consistent with a given concurrency alphabet in the sense that nodes are connected appropriately, i.e., reflecting the relation between their labels.

Definition 42 : dependence comdags

A dependence comdag over a combined concurrency alphabet (Σ, sim, ser) is a comdag $(X, \prec, \sqsubset, \ell)$ such that $\ell : X \to \Sigma$ and for all elements $x \neq y$ of X:

- $(\ell(x), \ell(y)) \notin sim$ implies $x \prec y$ or $y \prec x$.
- $(\ell(x), \ell(y)) \notin ser$ implies $x \prec y$ or $y \sqsubset x$.
- $x \sqsubset y$ implies $(\ell(y), \ell(x)) \notin ser$.
- $x \prec y$ implies $(\ell(x), \ell(y)) \notin ser$.

Every comdag has an associated step language consisting of all step sequences that can be read from it as a stratified poset while respecting the indicated ordering. The step language of any dependence comdag over $CCA = (\Sigma, sim, ser)$ thus consists of sequences of sets which are steps relative to CCA, i.e., $(a, b) \in sim$ for every pair of distinct symbols a and b in any step ✓. Moreover, deleting any arc from such comdag changes its step language. Formally, two dependence comdags are isomorphic *iff* their step languages are the same ✓. Moreover, with each step sequence, a dependence comgraph can be associated which has as its nodes the symbol occurrences of the step sequence and arcs implied by their dependencies.

The *canonical dependence comdag* of a step sequence $\sigma = U_1 \dots U_n$ over CCA is $candepcomdag(\sigma) \overset{\text{df}}{=} (occ(\sigma), \prec, \sqsubset, \ell)$ where, for all symbol occurrences a^i and b^j in $occ(\sigma)$ we have $\ell(a^i) \overset{\text{df}}{=} a$ and:

- $a^i \prec b^j$ if $(a, b) \notin ser$ and the i-th occurrence of a precedes the j-th occurrence of b within σ.

- $a^i \sqsubset b^j$ if $(b, a) \notin ser$ and the j-th occurrence of b does not precede the i-th occurrence of a within σ.

Figure 5.16 shows the canonical dependence comdag of the step sequence $\sigma_{\texttt{exmpl}}$.

Canonical dependence comdags capture precisely the essence of the comtrace equivalence relation as $candepcomdag(\sigma) = candepcomdag(\tau)$ *iff* σ and τ are comtrace equivalent step sequences ✓. Hence it is possible to define the canonical dependence comdag of a comtrace α as $candepcomdag(\alpha) \overset{\text{df}}{=} candepcomdag(\sigma)$, where σ is any step sequence in α. Moreover, the step sequences defined by the canonical dependence comdag of a comtrace are exactly the step sequences comprising that comtrace.

Fact 43 : Let α be a comtrace. Then $steplanguage(candepcomdag(\alpha)) = \alpha$.

Hence distinct comtraces have distinct canonical dependence comdags and it follows that comtraces are in one-to-one correspondence with dependence comdags.

Finally, the *canonical so-structure* of a comtrace α is defined as $cansos(\alpha) \overset{\text{df}}{=} candepcomdag(\alpha)^+$. The concluding result states that comtraces and their canonical so-structures capture the same sets of behaviours.

Fact 44 : Let α be a comtrace.

- $strat(cansos(\alpha)) = canstratposet(\alpha)$.[a]
- $steps(strat(cansos(\alpha))) = \alpha$.

[a]Note that in $canstratposet(\alpha)$ the comtrace α is treated as a set of step sequences.

In this way we have obtained the unique causality structure of a comtrace.

5.5.5 Step Sequences, Comtraces and Processes of ENI-Systems

Returning to the starting point of this section, i.e., to ENI-systems, we now aim at capturing the intrinsic causality in their behaviours. Since the treatment follows the same pattern as that provided for the EN-systems in the previous section, we will be fairly brief, and further motivations and discussion can be found there.

Let $ENI = (P, T, F, C_{init}, Inh)$ be henceforth a fixed ENI-system.

First, the basic operational behaviour of *ENI* is captured by its step language $stepseq(ENI)$ defined as the set of all sequences σ of non-empty steps of transitions such that $C_{init}[\sigma\rangle$. Clearly, $stepseq(ENI)$ is a prefix-closed, regular set of step sequences.

To capture the causal ordering of transition occurrences in ENI's behaviour, we will use comtraces and their so-structures. For this reason, we associate with ENI the combined concurrency alphabet CCA with $\Sigma = T$ and its simultaneity and serialisability relations given respectively by:

- $(t, t') \in sim$ if ${}^\bullet t^\bullet \cap {}^\bullet t'^\bullet = {}^\circ t' \cap {}^\bullet t = {}^\circ t \cap {}^\bullet t' = \varnothing$.
- $(t, t') \in ser$ if $(t, t') \in sim$ and $t^\bullet \cap {}^\circ t' = \varnothing$.

It is not difficult to see that the combined concurrency alphabet for the `ENI` in Figure 5.14 is precisely `CCA` defined earlier in this section.

The following key result is a consequence of the observation that all step sequences over CCA which are comtrace equivalent to a step sequence of ENI are steps sequences of ENI as well $\checkmark$.

Fact 45 : $stepseq(ENI) = \bigcup_{\sigma \in stepseq(ENI)} \langle\sigma\rangle$.

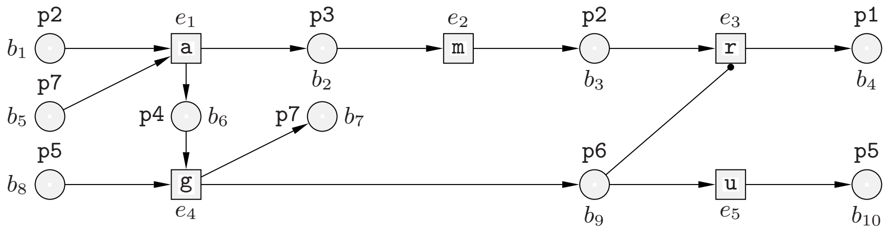
Hence the step language of ENI can be partitioned into comtraces. According to this result, and building on the theory expounded earlier on, we may state that the causal behaviour of ENI-systems can be captured by the so-structures corresponding to the comtraces partitioning their step language. We thus treat the causality issues at hand here similar to the approach presented as in the previous section for EN-systems based on traces and their corresponding posets.

As a conclusion to this section we will present a theory of processes for ENI-systems and demonstrate that the causality and weak causality captured in the comtraces and their so-structures of an ENI-system agree with this process semantics. To define processes, nets similar to the occurrence nets of EN-systems are used to describe the concurrent runs of ENI-systems. This requires an extension of the notion of an occurrence net which has been designed to handle nets with ordinary rather than inhibitor arcs. To deal with such arcs at the level of occurrence nets we introduce so-called activator arcs. Each such arc plays a role dual to that of an inhibitor arc. An activator arc between a place and transition test for the presence of a token in the place, but this token is not affected (removed) by the occurrence of that transition.

Definition 46 : activator occurrence nets

An activator occurrence net (ao-net) is a relational tuple $AON \overset{\text{df}}{=} (B, E, R, \ell, Act)$ such that the first four components form an (underlying) occurrence net and $Act \subseteq B \times E$ is a set of activator arcs.

Similarly as an occurrence net, an ao-net represents a concurrent execution or run of a system and so it has to be acyclic in some sense, to exclude



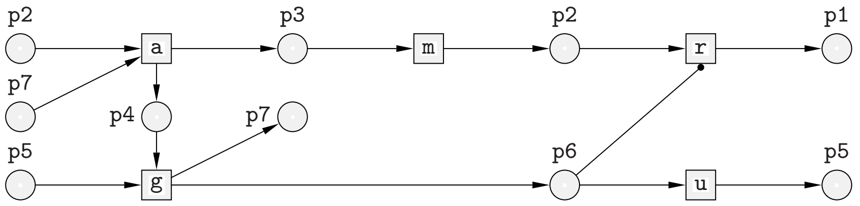


Fig. 5.17. An activator occurrence net AON with nodes labelled by places and transitions of the and ENI-system ENI in Figure 5.14 (top), and the same net with identities of the nodes omitted (bottom).

circularity in the description of the run. It is therefore assumed that the relational structure $comdag(AON) \overset{\text{df}}{=} (E, \prec_{loc}, \sqsubset_{loc}, \ell|_E)$, where $\prec_{loc}$ and $\sqsubset_{loc}$ are relations respectively given by $(R \circ R)|_{E \times E} \cup (R \circ Act)$ and $Act^{-1} \circ R$, is a comdag. Intuitively, these two relations provide local information on the causality between event occurrences based on the dynamics of the ao-net. Thus $\prec_{loc}$ stands for precedence (the first event has to produce a token for consumption or testing by the second event) and $\sqsubset$ for weak precedence (the first transition cannot happen after the second one, since the latter consumes a token for which the former tests).

Activator arcs are drawn with small black circles as arrowheads and, for every transition t, ${}^{\blacklozenge}t$ denotes the set of all places connected by activator arcs with t, i.e., $(p, t) \in Act$. Figure 5.17 shows an ao-net. The step sequences of an ao-net are defined as for its underlying occurrence net, except that a step U is enabled at a configuration C if, in addition, ${}^{\blacklozenge}U \subseteq C$ where ${}^{\blacklozenge}U$ consists of all places connected by activator arcs to transitions in U. Other notions, including the default initial and final configurations, are inherited from the underlying occurrence net.

Every occurrence net defines a dag representing the direct information on the causality between its events, and then through transitive closure also a poset of events. The same approach can be applied to an ao-net, but in this case the resulting causality structure for AON is the so-structure generated through the transitive closure of $comdag(AON)$ defined above. For example, the comdag generated by the ao-net in Figure 5.18 is nothing but the `cdag` shown in Figure 5.15, and so the corresponding so-structure is the `sos` in Figure 5.16.

Processes of ENI-systems are similar to those of EN-systems with the inhibitor arcs of the system represented by activator arcs which rather than testing for the absence of tokens are used to test for the presence of tokens in complement places. Hence, it is tacitly assumed that each place of *ENI* adjacent to an inhibitor arc has a complement place in the underlying EN-system. (Every ENI-system can be transformed into an ENI-system with an isomorphic state graph and satisfying this property ✓.)

Definition 47 : processes of ENI-systems

A process of *ENI* is an ao-net (B, E, R, Act, ℓ) such that the underlying occurrence net of the latter is a process of the underlying EN-system of the former and, in addition, ℓ is injective on ${}^{\blacklozenge}e$ and $\ell({}^{\blacklozenge}e) = \widetilde{{}^{\circ}\ell(e)}$ for every event e in E.

The processes of an ENI-system give information on its behaviour. The *step language* of a process AON of *ENI* is the set $steplanguage(AON)$ of all step sequences $\ell(\sigma)$ such that σ is a step sequence from the default initial configuration of AON to the default final configuration. Observe here that the reachable configurations of AON are also reachable configurations of its underlying occurrence net. Consequently, the labelling of AON is injective on all its reachable configurations and on the steps in its step sequences ✓.

Definition 47 is sound in the sense that the step language of an ENI-system coincides with the step languages of its processes.

Fact 48 : $stepseq(ENI) = \bigcup_{AON \in \mathcal{AON}} steplanguage(AON)$ where $\mathcal{AON}$ is the set of all processes of *ENI*.

The processes of an ENI-system can be described algorithmically as well. This construction is also based on the one given earlier for EN-systems, showing once again that the addition of inhibitor arcs leads to conservative extensions of notions and results presented earlier on.

Definition 49 : processes construction

The activator occurrence net AON_σ generated by a step sequence $\sigma = U_1 \ldots U_n$ of *ENI* is the last element in the sequence $N_0, \ldots, N_n$ where each N_k is an activator occurrence net $(B_k, E_k, R_k, A_k, \ell_k)$ constructed as in Definition 32 with the following additions:
Step 0: $A_0 = \varnothing$.
Step k: $A_k = A_{k-1} \cup \{(\widetilde{p}^{\,\triangle\widetilde{p}}, t^{1+\triangle t}) \mid t \in U \wedge p \in {}^{\circ}t\}$.

Figure 5.18 shows in stages how to construct the ao-net following the execution of the step sequence $\sigma_{\mathtt{exmpl}}$ of `ENI`. The resulting ao-net is a process of `ENI` (it is isomorphic with the net in Figure 5.17).

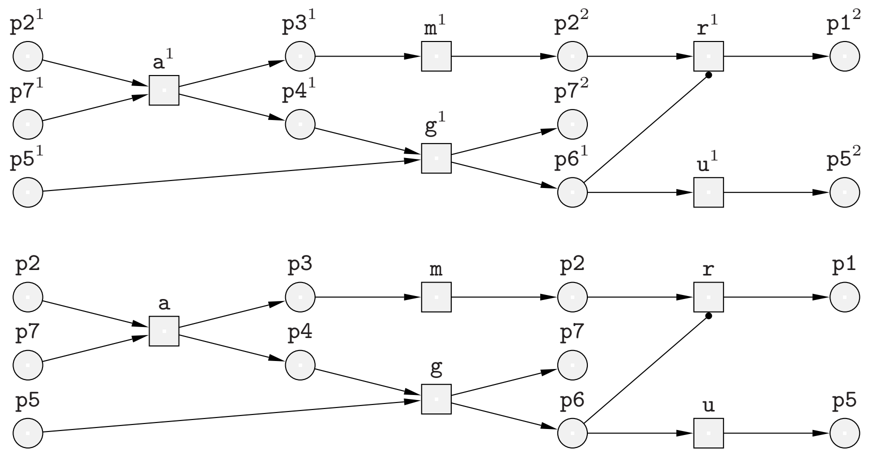


Fig. 5.18. The ao-net $\mathtt{AON}_{\sigma_{\mathtt{exmpl}}}$ constructed for the ENI-system in Figure 5.14: node-oriented view (top), and label-oriented view (bottom).

Fact 50 : Each ao-net constructed in Definition 49 is a process of *ENI* and, for each process of *ENI*, there is a run of the construction from Definition 49 generating an isomorphic ao-net.

As a last point we compare the causality structures of an ENI-system as captured in comtraces through their so-structures with the process semantics and its comdags and related so-structures.

Since splitting and combining steps of transitions according to the simultaneity and serialisability relations defined by the net have no effect on the process construction we know that $AON_\sigma = AON_\tau$ *iff* σ and τ are comtrace equivalent step sequences $\checkmark$. Hence with each comtrace one process (up to isomorphism) is associated. Conversely the step language of a process of *ENI* is identical to its defining comtrace $\checkmark$. We can then relate the comtraces and processes generated by the step sequences of an ENI-system.

Fact 51 : Let σ be a step sequence of *ENI*.

- $\langle\sigma\rangle = steplanguage(AON_\sigma)$.
- $cansos(\langle\sigma\rangle) = comdag(AON_\sigma)^+$.

Hence comtraces and processes give the same views on the causalities in the behaviours of ENI-systems, again providing a justification for the fundamental soundness of the concurrency semantics they both capture.

5.5.6 Bibliographical Remarks

Inhibitor arcs have been found to be particularly useful in areas such as communication protocols (see, e.g., [5]) and performance analysis (see, e.g., [12]) and, indeed, perhaps the most natural extension of the standard net model, e.g., [43] stated that 'Petri nets with inhibitor arcs are intuitively the most direct approach to increasing the modelling power of Petri nets' (note that when added to the PT-system model considered later on, they lead to a strictly more expressive model as now Turing machines can be simulated). This section is based on the work reported in [27] which has been further developed, e.g., in [31] and [28].

The enabledness of transitions in ENI-systems and ao-nets is based on an *a priori* condition: the inhibitor/activator places of transitions occurring in a step should obey the relevant constraints before the step is executed, but not necessarily afterwards. Alternative treatments of this issue are provided in, e.g., [6] and [50].

5.6 Place Transition Nets

In this section we give an impression of how the trace approach to describe net behaviour can be generalised to Place/Transition systems (PT-systems for short), a well-known and prominent class of Petri nets that employ states to describe the availability of local resources in a quantitative way rather than to indicate simply the holding or not-holding of local conditions. PT-systems are of more practical use than EN-systems since certain repetitive features which would lead to unwieldy EN-systems can be collapsed in a PT-system thus allowing more compact representations of systems. Moreover, they are more expressive.

Let us return to the running example. Instead of indicating whether or not the buffer contains an item at all, the buffer place p4 in PT1, the first net in Figure 5.19, gives the number of available (produced and not yet consumed) items. Initially there is one item in the buffer, represented by one token in p4. The producer is allowed to add items to the buffer also when it is not empty. Each such item is represented by an additional token in p4. In diagrams of PT-systems, tokens are used to indicate the current multiplicity of (resources in) a place; thus it is possible to have more than one token in a place. In this example, the number of tokens (items) in p4 (the buffer) is not a priori bounded. The second net PT2 in Figure 5.19 models a producer/consumer system with a buffer (p4) of bounded capacity (two in this case). Its current capacity is given through its complement place p7. The token count in the buffer and the complement together is always exactly 2. Adding an item to the buffer by the producer decreases its remaining capacity and similarly the consumption of an item by the consumer leads to an increase of capacity. The third net PT3 in Figure 5.19 models a producer/consumer system with two consumers. When there are two or more tokens in the buffer and two consumer tokens in the local state p5, then the two consumers can each consume an item without interfering with one another (concurrently). Hence, rather than using a separate subsystem for each consumer the PT-systems model makes it possible to use multiple occurrences of tokens in a net to model identical behaviour.

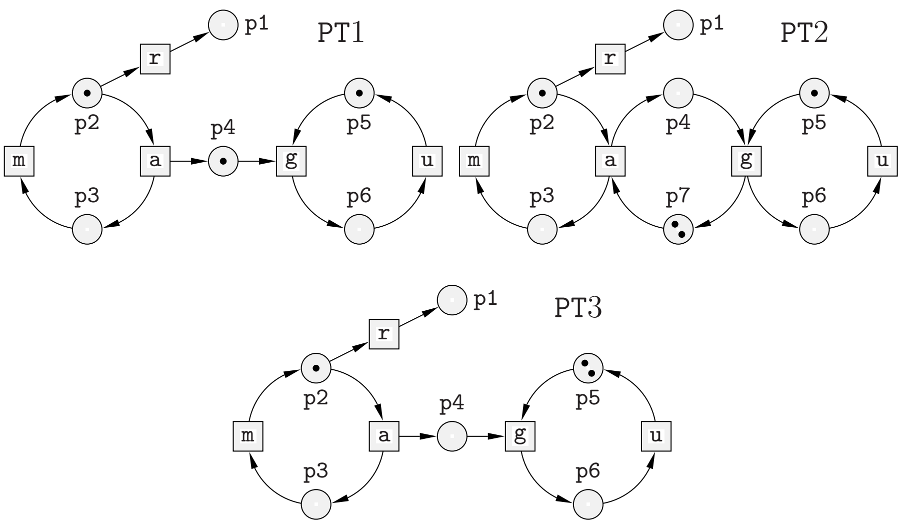


Fig. 5.19. Three PT-systems for the running example: PT1 with an unbounded buffer – containing one item in the initial state – and one consumer; PT2 with a buffer of capacity two; and PT3 with an unbounded buffer and two consumers.

Thus we now have for nets a new notion of state described by multiplicities of places (natural numbers) rather than subsets of places (booleans). Formally, these states, called markings, are multisets of places.[3]

Definition 52 : markings

A marking of a net $N = (P, T, F)$ is a mapping $M : P \to \mathbb{N}$.

The dynamics of nets with markings as global states is based on a new occurrence rule for individual transitions describing their consumption and production of local resources.

[3] A multiset over a set X is a function $\mu : X \to \mathbb{N}$, and any subset of X may be viewed through its characteristic function as a multiset over X. By $\mathbb{M}(X)$ we denote the set of all multisets over X. For two multisets μ, ν with a common domain X, we write $\mu \leq \nu$ if $\mu(x) \leq \nu(x)$ for all $x \in X$.

Definition 53 : transition occurrences

A transition t can occur (or is enabled) at a marking M if $M(p) \geq 1$ for every place $p \in {}^\bullet t$. Its occurrence then leads to a new marking $M'(p) \overset{\text{df}}{=} M(p) - |\{(p,t)\} \cap F| + |\{(t,p)\} \cap F|$ for every place $p \in P$.

Thus t is enabled at a marking M whenever M assigns at least one token to each input place of t. If t occurs, then it consumes one token from each of its input places and produces one token in each of its output places. (Again, the enabledness of a transition and its effect are defined completely locally and do not depend on the global properties of a state.) If t is enabled at M, we write $M[t\rangle$ and if its occurrence at M leads to the marking M', we write $M[t\rangle M'$.

When we consider configurations as (a simple kind of) markings and compare this definition with Definition 17 of transition enabling and occurrence at a configuration, then the following observations are immediate: a transition which is enabled at a configuration is also enabled at the marking represented by the configuration and its occurrence would have the same effect. However, due to the possibility of contact, the reverse does not hold in general. Without output requirements, a transition may have an input place which is also one of its output places (a loop), and still be able to occur at a marking.

Concurrent occurrence of transitions at a marking is possible, provided that enough resources (tokens per place) are available for all transitions together. When multiple transitions occur, the effect of their concurrent occurrence is the accumulated effect of their individual occurrences. As before (Definition 18), a step of a net is a subset of its transitions. A step U can occur at a marking M if $M(p) \geq |p^\bullet \cap U|$ for all places p. Its occurrence then leads to a new marking M' such that $M'(p) \overset{\text{df}}{=} M(p) - |p^\bullet \cap U| + |{}^\bullet p \cap U|$ for every place p. If U is enabled at M, we write $M[U\rangle$ and if its occurrence at M leads to the marking M', we write $M[U\rangle M'$. Note that, the transitions in a step may have overlapping neighbourhoods. In particular, input places can be shared. In that case, however, for the step to be enabled, the marking under consideration should assign to these places at least one token for each of their output transitions in the step (there is a conflict at the marking, if each transition individually is enabled, but they cannot occur as a step). Single transition occurrences are special cases of step occurrences. Furthermore, the case of steps is easily extended to multisets of transitions occurring at a marking: a multiset U of transitions can occur at a marking M if $M(p) \geq \sum_{t \in p^\bullet} U(t)$ for all places p. In such a case, U can be executed leading to the marking M' given by $M'(p) \overset{\text{df}}{=} M(p) - \sum_{t \in p^\bullet} U(t) + \sum_{t \in {}^\bullet p} U(t)$ for every place p. Multisets of transitions model the phenomenon of auto-concurrency. In the producer/consumer system with two consumers (`PT3` in Figure 5.19) transition `g` can occur (twice) concurrent with itself at every marking with two or more tokens in the buffer place `p4` and two consumer tokens in `p5`. For reasons of convenience, we will give emphasis to the explanation of notions based on steps with only occasional reference to multisets.

The occurrence of a step at a marking leads to a next marking. Hence, lifting the terminology introduced for (contact-free) EN-systems and their semantics to the more general level of PT-systems, we can define step sequences (and also firing sequences and multiset sequences) as finite sequences of non-empty steps (single transitions or non-empty multisets, respectively) occurring one after another from a given marking. A step sequence σ from a marking M is a possibly empty sequence $\sigma = U_1 \dots U_n$ of non-empty steps U_i such that $M[U_1\rangle M_1, \dots, M_{n-1}[U_n\rangle M'$, for some markings $M_1, \dots, M_{n-1}$. We write $M[\sigma\rangle M'$ or $M[\sigma\rangle$ and say that M' is *reachable* from M.

When a step is enabled at a marking, sufficient resources are available at that marking for the independent occurrence of each of the transitions in the step. Hence, a diamond property as formulated in the first part of Fact 20 holds.

Fact 54 : Let M, M' be markings and U, U' be steps of a net such that $U \cap U' = \varnothing$. Then $M[U \cup U'\rangle M'$ implies $M[UU'\rangle M'$.

As a consequence, every step of transitions occurring at a marking can be split into any sequence of subsets forming a partition of this set and each such step sequence has the same effect (leads to the same marking) as the original step. In particular, each step in a step sequence can be split into a firing sequence which is an arbitrary permutation of its transitions. For multisets, a similar diamond property can easily be proved and so every multiset sequence can be decomposed into a step sequence ✓. However, due to loops, the second statement in Fact 20 does not hold: it is not the case that a diamond of step sequences at a marking implies that the transitions involved could also occur concurrently. It is possible, e.g., to have step sequences $\{a\}\{b\}$ and $\{b\}\{a\}$ from some marking M of a PT-system, while $M[\{a, b\}\rangle$ does not hold ✓.

5.6.1 PT-Systems and Their State Spaces

Equipping nets with initial markings leads to a new class of net systems.

Definition 55 : PT-systems

A place transition system (or PT-system) consists of an underlying net and an initial marking. Its state space consists of all markings reachable from the initial marking.

That is, a PT-system is a relational tuple $PT \overset{\text{df}}{=} (P, T, F, M_{init})$ such that (P, T, F) is a net and $M_{init} : P \to \mathbb{N}$ is its initial marking. Because of the diamond property (Fact 54) for steps and multisets, reachability of markings is the same whether defined in terms of firing sequences, or step sequences, or multiset sequences ✓. Hence, also the state space is the same for the three semantics. Contact-free EN-systems can be viewed as special PT-systems with the additional property of being *safe*, i.e., no reachable marking will ever assign more than one token to a place ✓. (Note that in a safe PT-system there is no auto-concurrency ✓.) Exactly as for EN-systems we can consider the state graph of PT, with the markings reachable from M_{init} as nodes and with labelled arcs (M, U, M') whenever $M[U\rangle M'$. In addition, there are the sequential state graph of PT and its multiset state graph, both defined in the obvious way.

The most basic behaviour of a PT-system PT is captured by its language $\mathit{firseq}(PT)$ consisting of all firing sequences from its initial marking. Clearly, $\mathit{firseq}(PT)$ is a prefix-closed language, and each firing sequence corresponds to a unique path through the sequential state graph of PT starting from the initial marking. Since the numbers of tokens per place are not necessarily bounded, it is possible that the state space of PT is not finite (even though PT itself is a finite object) and $\mathit{firseq}(PT)$ not regular. Consider PT1, the first net in Figure 5.19 with its initial marking as given there. The number of tokens in the buffer place p4 can be arbitrarily large, but apart from the initial item, the consumer can never consume more items than added to the buffer by the producer. Thus, $\mathit{firseq}(\mathtt{PT1}) \cap \{\mathtt{am}\}^*\{\mathtt{gu}\}^* = \{(\mathtt{am})^k(\mathtt{gu})^n \mid n \leq k+1\}$. Consequently, $\mathit{firseq}(\mathtt{PT1})$ is not regular. Next to $\mathit{firseq}(PT)$, we have $\mathit{stepseq}(PT)$ and $\mathit{multisetseq}(PT)$, the step language and the multiset language PT consisting of all step sequences, multiset sequences respectively, from its initial marking. A PT-system PT such that $\mathit{multisetseq}(PT) = \mathit{stepseq}(PT)$, i.e., it does not exhibit any auto-concurrency at all, is *co-safe*. Note that safe PT-systems are necessarily co-safe, but that the converse implication is not true ✓.

In contrast to the situation for EN-systems, diamonds in the sequential state graphs of PT-systems do not imply nor exclude possible concurrent behaviour and $\mathit{stepseq}(PT)$ cannot be reconstructed from $\mathit{firseq}(PT)$. Similarly, since information on auto-concurrency is missing in the step sequence semantics, $\mathit{multisetseq}(PT)$ cannot, in general, be derived from $\mathit{stepseq}(PT)$. In other words, two PT-systems with isomorphic sequential state graphs may have state graphs which are not isomorphic, and systems with isomorphic state graphs may have multiset state graphs which are not isomorphic. Moreover, for PT-systems, the concurrency, conflict and causality relations between transitions are not merely structural, but may change with the current marking. As an example, consider PT4 in Figure 5.20. This PT-system has exactly all prefixes of *all* words which are permutations of the symbols a, b, and c, as its firing sequences. As step sequences it has in addition $\{\mathtt{a},\mathtt{b}\}$, $\{\mathtt{a},\mathtt{c}\}$, $\{\mathtt{b},\mathtt{c}\}$, $\{\mathtt{a},\mathtt{b}\}\mathtt{c}$, $\{\mathtt{a},\mathtt{c}\}\mathtt{b}$, $\{\mathtt{b},\mathtt{c}\}\mathtt{a}$, $\mathtt{a}\{\mathtt{b},\mathtt{c}\}$, $\mathtt{b}\{\mathtt{a},\mathtt{c}\}$, and $\mathtt{c}\{\mathtt{a},\mathtt{b}\}$. If, however, the initial marking would have assigned one token instead of two in the uppermost place, there would have been only firing sequences and no additional step sequences; and with initially three tokens in the uppermost place also $\{\mathtt{a},\mathtt{b},\mathtt{c}\}$ would

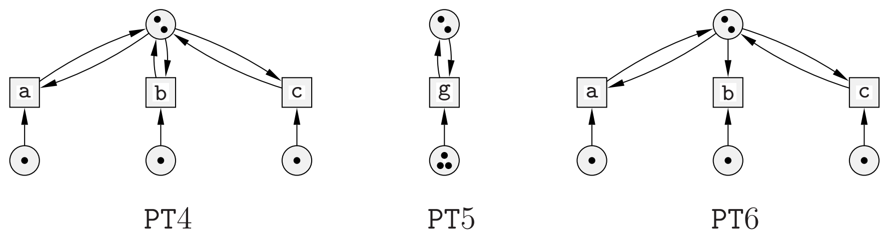


Fig. 5.20. Three PT-systems.

have been a step sequence of the system. In the second example system PT5 in Figure 5.20, auto-concurrency plays a role. It resembles PT4 with the three transitions merged (as well as their lower input places). Note that due to auto-concurrency, a diamond may degenerate to a single sequence (in this example, a sequence of two concurrent occurrences of g). Moreover, with initially only one token in the uppermost place this PT-system admits no other behaviour than the purely sequential ggg and its prefixes. Finally, in the third system PT6, we see that the transitions a and c can occur concurrently at the initial marking. If, however, b occurs first, then a and c are in conflict at the resulting marking. It is interesting to compare the step sequences and firing sequences of PT6 with those of PT4 ✓.

Before introducing a new more general notion of trace as part of a partial order approach to the operational semantics of PT-systems, we first consider the processes of PT-systems in order to gain more insight in the causality and concurrency in their behaviour.

5.6.2 Processes of PT-Systems

As before, processes formalise the idea of a concurrent run or a non-sequential observation of an execution of a system. Being a record of the changes of markings along some execution of a PT-system, they capture the intrinsic concurrency and causality (based on the production and consumption of resources) in the recorded behaviour. The notion of a process of a PT-system is a rather straightforward generalization of the process definition for EN-systems.

In what follows, $PT = (P, T, F, M_{init})$ is a fixed PT-system.

A process of PT is a labelled occurrence net that can be seen as a partial unfolding of PT in which conflicts have been resolved. Each of its events represents the occurrence of a transition and each condition corresponds to the occurrence of a *single* token in a place of PT.

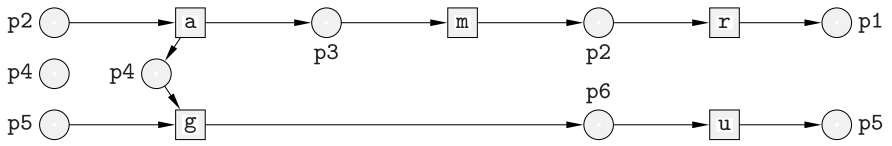


Fig. 5.21. A process ON′ of the PT-system PT1 in Figure 5.19 (node identities are omitted).

Definition 56 : processes of PT-systems

A process of PT is an occurrence net $ON = (B, E, R, \ell)$ such that:

- ℓ labels conditions with places and events with transitions.
- For every $p \in P$, $M_{init}(p) = |\{b \in C^{ON}_{init} \mid \ell(b) = p\}|$.
- ℓ is injective on the sets of input and output conditions of each event.
- For every $e \in E$, $\ell({}^\bullet e) = {}^\bullet\ell(e)$ and $\ell(e^\bullet) = \ell(e)^\bullet$.

The occurrence net in Figure 5.21 is a process of PT1 in Figure 5.19.

The difference with Definition 30 (processes of EN-systems) is that now the labelling of a process is not required to be injective on the default initial configuration. This configuration is intended to represent the initial marking of the PT-system and has for each token in each place, a condition labelled with the name of that place. Note that for (contact-free) EN-systems, Definitions 56 and 30 coincide. The structure underlying a process of a PT-system is an occurrence net and hence forms with its default initial configuration a contact-free EN-system with properties as discussed before for the processes of EN-systems. In particular, each process ON defines a dag, $dag(ON)$, representing the direct causal relationships between the events, and a partial order on its events obtained as its transitive closure $dag(ON)^+$, describing all causal dependencies. Moreover, for processes of PT-systems, their multiset and step semantics coincide. The labelling, however, will in general not be injective on the reachable configurations (the slices) and the steps executed. Consider, e.g., in Figure 5.21 the configuration reached after the execution of the event labelled by a. It has two conditions labelled by p4 together representing two tokens in place p4 of PT1.

To preserve the multiplicity of the labels associated to elements, (non-injective) labellings can be lifted to yield multisets of labels for subsets of their domain. Given a labelling $\ell : Y \to Z$ and a finite $X \subseteq Y$, define $\ell\langle X\rangle : Z \to \mathbb{N}$ by $\ell\langle X\rangle(z) \overset{\text{df}}{=} |\{x \in X \mid \ell(x) = z\}|$, for each $z \in Z$. The labelling can be applied in this way also to finite sequences of finite subsets of Y, $\ell\langle X_1 \dots X_n\rangle \overset{\text{df}}{=} \ell\langle X_1\rangle \dots \ell\langle X_n\rangle$. Note that if ℓ is injective on each of the X_i, then $\ell\langle X_1 \dots X_n\rangle$ and $\ell(X_1 \dots X_n)$ can be identified.

The multiset sequences (i.e., the step sequences) of a process of a PT-system are related via its labelling to the multiset sequences of the system.

Using an inductive argument, it can be proved that $C^{ON}_{init}[\sigma\rangle C$ in a process $ON = (B, E, R, \ell)$ of PT implies that $M_{init}[\ell\langle\sigma\rangle\rangle\ell\langle C\rangle$ in PT . Again, we let $ON)$ denote the set of all labelled firing sequences of ON from the default initial configuration to its default final configuration. The *multiset language* of ON is the set $multisetlanguage(ON)$ comprising all labelled step sequences from the default initial configuration to its default final configuration.

Since the structure and labelling of the processes reflect the flow relation of PT, it follows that all multiset sequences of PT can be derived from its processes.

Fact 57 : Let $\mathcal{ON}$ be the set of all processes of PT.

- $\mathit{firseq}(PT) = \bigcup_{ON\in\mathcal{ON}} ON)$.
- $\mathit{multisetseq}(PT) = \bigcup_{ON\in\mathcal{ON}} \mathit{multisetlanguage}(ON)$.

Conversely, all processes of a PT-system can be constructed from its multiset sequences.

Definition 58 : processes construction

For a multiset sequence $\sigma = U_1 \dots U_n$ of PT, an occurrence net ON^{σ}_{PT} can be generated as the last element in a sequence $N_0, \dots, N_n$ where each N_k is an occurrence net (B_k, E_k, R_k, ℓ_k) constructed thus.
Step 0: $B_0 \overset{\text{df}}{=} \{p^i \mid p \in P \wedge 1 \le i \le M_{init}(p)\}$ and $E_0 = R_0 \overset{\text{df}}{=} \varnothing$.
Step k: Given N_{k-1} we extend the sets of nodes as follows:

$$\begin{aligned} B_k &\overset{\text{df}}{=} B_{k-1} \cup \{p^{i+\triangle p} \mid p \in U_k^{\bullet} \wedge 1 \le i \le \textstyle\sum_{t\in{}^\bullet p} U_k(t)\} \\ E_k &\overset{\text{df}}{=} E_{k-1} \cup \{t^{i+\triangle t} \mid t \in U_k \wedge 1 \le i \le U_k(t)\} . \end{aligned}$$

In the above, the label of each node x^i is set to be x, and $\triangle x$ denotes the number of nodes of N_{k-1} labelled by x.
To define the arcs, we proceed as follows. For every $e = t^i \in E_k \setminus E_{k-1}$, we *choose*[a] two sets of conditions, $In_e \subseteq B_{k-1} \setminus codom(R_{k-1})$ and $Out_e \subseteq B_k \setminus B_{k-1}$, such that In_e comprises a distinct condition for each place in ${}^\bullet t$ and Out_e comprises a distinct condition for each place in $t^\bullet$. Moreover, for any two distinct $e, e' \in E_k \setminus E_{k-1}$, the sets In_e and In'_e as well as Out_e and Out'_e are mutually disjoint. Then:

$$R_k \overset{\text{df}}{=} R_{k-1} \cup \bigcup_{e\in E_k\setminus E_{k-1}} (In_e \times \{e\}) \cup (\{e\} \times Out_e) .$$

[a]This means that, in general, more than one process can be constructed for a given multiset sequence.

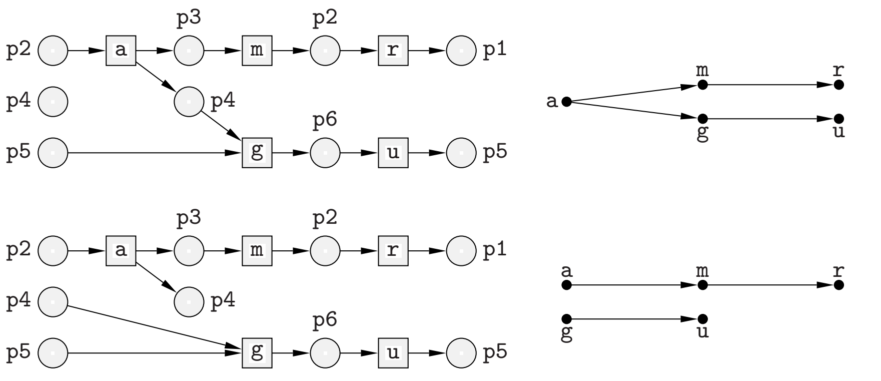


Fig. 5.22. Two processes and their causality dags. Both processes are associated with the multiset sequence $\mathtt{a}\{\mathtt{m},\mathtt{g}\}\{\mathtt{r},\mathtt{u}\}$ of PT1 in Figure 5.19.

The construction is illustrated in Figure 5.22 for the PT-system PT1 from Figure 5.19 and its multiset sequence $\mathtt{a}\{\mathtt{m},\mathtt{g}\}\{\mathtt{r},\mathtt{u}\}$. The topmost process given there is isomorphic to the process $\mathtt{ON}'$ of the PT-system PT1 in Figure 5.21.

Fact 59 : Each occurrence net constructed in Definition 58 is a process of PT and, for each process of PT, there is a run of the construction from Definition 58 generating an isomorphic occurrence net.

Thus, also for PT-systems, their operationally defined processes and axiomatically defined processes are essentially the same. The labelling of these processes is in general not injective on their slices (reachable configurations). Each slice represents through its labelling a reachable marking of the PT-system with for each token in each place, a condition labelled with the name of that place. This leads to a distinct representation of each token in a place even though in PT-systems such occurrences of tokens are usually deemed indistinguishable. When constructing a process for a given multiset sequence, there may be more than one (distinct representation of) a token available as input to a next occurrence of a transition leading to the choice referred to in Definition 58. The two different processes in Figure 5.22 are the result of choosing between the two conditions labelled by p4 as input for the event labelled with g after the occurrence of a. When the distinction between tokens in a place is undesirable, equivalence classes of processes can be used as representations of runs. The equivalence used identifies two processes whenever one can be obtained from the other by 'swapping' the parts of the occurrence nets following two conditions which occur together in a slice and have identical labels. The two processes in Figure 5.22 are swapping equivalent. Note that they give rise to different partial orders.

5.6.3 Local Traces

We are now ready for lifting the definition of traces to the level of PT-systems. In these systems the concurrency and causality relations between transitions are determined by the current marking. In terms of firing sequences (step sequences and multiset sequences), this means that independence of symbols is not globally defined for all occurrences of symbols, but in a local fashion depending on the preceding history (left-context or prefix). Another new feature is that for PT-systems, independence is not a binary relation. Consider, e.g., `PT4` in Figure 5.20. Here we have that the three transitions can occur concurrently in pairs, but not as a triple (which would be implied in the case of an EN-system ✓). Consequently, multisets (or sets when auto-concurrency is ruled out) rather than pairs have to be used to describe the independence among symbols or action occurrences. A local independence relation provides the information on when and which symbols (and how many occurrences of each) are independent.

Definition 60 : local concurrency alphabets

A local independence relation *Lind* over an alphabet Σ is a subset of $\Sigma^* \times \mathbb{M}(\Sigma)$. A local concurrency alphabet $LCA \overset{\mathrm{df}}{=} (\Sigma, Lind)$ consists of an alphabet Σ and a local independence relation *Lind* over Σ.

The pair (u, X) being an element of a local independence relation *Lind* indicates that the elements of X can occur concurrently (and with the multiplicities defined in X) once u has been executed. As an example, consider again the PT-systems in Figure 5.20. The local independence relation of `PT4` will include the pairs $(\lambda, \{\mathtt{a}, \mathtt{b}\})$, $(\lambda, \{\mathtt{a}, \mathtt{c}\})$ and $(\lambda, \{\mathtt{b}, \mathtt{c}\})$, but not $(\lambda, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\})$. In addition, $(\mathtt{c}, \{\mathtt{a}, \mathtt{b}\})$, $(\mathtt{b}, \{\mathtt{a}, \mathtt{c}\})$ and $(\mathtt{a}, \{\mathtt{b}, \mathtt{c}\})$ will also belong to this local independence relation. However, for `PT6`, $(\lambda, \{\mathtt{a}, \mathtt{c}\})$ will be included in its local independence relation, but not $(\mathtt{b}, \{\mathtt{a}, \mathtt{c}\})$. The PT-system `PT5` will give rise to the pair (λ, G_2), but not to (λ, G_3), where G_2 and G_3 are the multisets given by $G_i(g) \overset{\mathrm{df}}{=} i$. We will see later how the local concurrency alphabet of a PT-system can be defined by its behaviour.

First we introduce *local traces* based on a new (local) trace equivalence relation. Again the elementary step in the identification of sequences is the exchange of positions between adjacent independent symbol occurrences. Let $(\Sigma, Lind)$ be a local concurrency alphabet. Then, for two words, $u, v \in \Sigma^*$, we write $u \sim_{Lind} v$ if there are words $w, z \in \Sigma^*$, a multiset X over Σ, and $x, y \in \Sigma^*$ such that $(w, X) \in Lind$, $X(a) = \#_a(x) = \#_a(y)$ for all $a \in \Sigma$, and $u = wxz$ and $v = wyz$. The *local trace equivalence* $\equiv_{Lind}$ on Σ^* is the reflexive and transitive closure of $\sim_{Lind}$.

Let `Lind4` be the local independence relation associated with `PT4` (see above for its elements relevant here), then we have $\mathtt{bac} \sim_{\mathtt{Lind4}} \mathtt{abc} \sim_{\mathtt{Lind4}}$

$\texttt{acb} \sim_{\texttt{Lind4}} \texttt{cab} \sim_{\texttt{Lind4}} \texttt{cba} \sim_{\texttt{Lind4}} \texttt{bca}$ and so $\texttt{bac} \equiv_{\texttt{Lind4}} \texttt{bca}$. Thus all firing sequences of length three of PT4 are local trace equivalent.

Definition 61 : local traces

A local trace over a local concurrency alphabet $(\Sigma, Lind)$ is any equivalence class of the local trace equivalence relation $\equiv_{Lind}$.

The local trace containing a given word u is denoted by $[\![u]\!]_{Lind}$, and the set of all local traces by $\Sigma^*/_{\equiv_{Lind}}$. Whenever the independence relation $Lind$ is clear from the context, we may drop it when writing $[\![u]\!]_{Lind}$ etc. Note that the empty local trace is $[\![\lambda]\!] = \{\lambda\}$. For the PT-system PT4, we have $[\![\texttt{abc}]\!]_{\texttt{Lind4}} = \{\texttt{abc}, \texttt{bac}, \texttt{bca}, \texttt{cba}, \texttt{cab}, \texttt{acb}\}$. Note that, in the same way as before, it can be shown that bac and bca are also local trace equivalent with respect to the local independence relation Lind6 associated with PT6 even though $(\texttt{b}, \{\texttt{a}, \texttt{c}\}) \notin \texttt{Lind6}$. Hence $[\![\texttt{abc}]\!]_{\texttt{Lind6}} = [\![\texttt{abc}]\!]_{\texttt{Lind4}}$.

Local independence and local trace equivalence are generalisations of the independence relation and trace equivalence underlying the original traces.

Fact 62 : Let (Σ, Ind) be a concurrency alphabet and $Lind \overset{\mathrm{df}}{=} \{(u, X) \in \Sigma^* \times \mathbb{P}(\Sigma) \mid (X \times X) \setminus id_\Sigma \subseteq Ind\}$. Then $\equiv_{Ind}$ and $\equiv_{Lind}$ coincide.

Just like in the case of trace equivalence, it is easily seen that whenever two words are local trace equivalent, they have the same length and alphabet ✓. However, due to the local character of the independence relation, the property that the order of dependent symbols is the same in all words of a local trace does not hold true (see above where we had $\texttt{bac} \equiv_{\texttt{Lind6}} \texttt{bca}$). Consequently, one cannot associate a single well-defined dependence graph with all words in a local trace in the same way as was done for traces. Also concatenation cannot be well-defined by concatenating representatives. As an example, consider the local independence relation $\{(\lambda, \{b, c\})\}$ over the alphabet $\{a, b, c\}$. Then $[\![bc]\!] = [\![cb]\!]$, but $[\![abc]\!] \neq [\![acb]\!]$. Still, local trace equivalence is a *right-congruence*.

Fact 63 : Let $(\Sigma, Lind)$ be a local concurrency alphabet and $u, v, w \in \Sigma^*$. Then $u \equiv_{Lind} v$ implies $uw \equiv_{Lind} vw$.

Hence the right-concatenation $\oplus$ of local traces with words is well-defined by $[\![u]\!]_{Lind} \oplus w \overset{\mathrm{df}}{=} [\![uw]\!]_{Lind}$, and we say that a local trace α is a *prefix of* a local trace β if $\beta = \alpha \oplus w$ for some word w. This (quasi-)prefix ordering is well-defined ✓. We use again the $\lhd$-notation and indice the fact that α is a prefix of β as $\alpha \unlhd \beta$. Moreover, if $\alpha \unlhd \beta$ and $\alpha \neq \beta$ then we write $\alpha \lhd \beta$. Note that $[\![u]\!]_{Lind} \lhd [\![v]\!]_{Lind}$ and $v \equiv_{Lind} w$ implies that $[\![u]\!]_{Lind} \lhd [\![w]\!]_{Lind}$. However, $[\![u]\!]_{Lind} \lhd [\![v]\!]_{Lind}$ does not necessarily imply that $u \lhd v$ holds ✓.

The prefix relation of local traces provides information on the relationships between the occurrences of symbols. In particular, when a local trace is used as the representation of a run of a concurrent system, its prefixes correspond to the different histories, each of which may be extended to a sequential representation of that run. Returning once more to the examples of $\mathtt{Lind}4$ and $\mathtt{Lind}6$, it should be observed that the trace $\llbracket \mathtt{abc} \rrbracket_{\mathtt{Lind4}} = \llbracket \mathtt{abc} \rrbracket_{\mathtt{Lind6}}$ has the same prefix structure with respect to both local independence relations. Note, however, that from the prefix $\llbracket \mathtt{b} \rrbracket_{\mathtt{Lind4}}$ a concurrent step $\{\mathtt{a},\mathtt{c}\}$ can be executed leading to $\llbracket \mathtt{abc} \rrbracket_{\mathtt{Lind4}}$, whereas in order to reach $\llbracket \mathtt{abc} \rrbracket_{\mathtt{Lind6}}$ from $\llbracket \mathtt{b} \rrbracket_{\mathtt{Lind6}}$ the symbols $\mathtt{a}$ and $\mathtt{c}$ have to be executed sequentially. Adding this multiset information in the form of arcs labelled with multisets in accordance with the given local independence relation — if possible — would yield a labelled structure comparable to a state graph and allow one to distinguish between different concurrent behaviours defining the same local traces. We will come back to this issue shortly.

5.6.4 PT-Systems and Local Traces

The local independence relation associated with a PT-system describes all multisets of transitions that can occur concurrently during a run of the system.

Definition 64 : local concurrency alphabets of PT-systems

The local concurrency alphabet of PT is $LCA_{PT} \overset{\mathrm{df}}{=} (T, Lind_{PT})$ where the local independence relation $Lind_{PT}$ comprises all pairs of firing sequences of PT with multisets of transitions enabled at the corresponding marking.

Thus $Lind_{PT} = \{(u, X) \in T^* \times \mathbb{M}(\Sigma) \mid M_{init}[u\rangle M' \wedge M'[X\rangle\}$.

In order to facilitate a comparison of concurrent behaviour of different PT-systems, local independence is defined on the abstract behavioural level of firing sequences rather than at concrete markings. Since for PT-systems reachability of markings is the same for firing / step / multiset sequences, all potential concurrency in the system can be described in terms of (local) independence of transitions after a firing sequence. Note that because local traces are equivalence classes comprising words only (rather than multiset sequences), they are not affected when auto-concurrency is ignored, i.e., by restricting the local independence relation of PT to pairs (u, X) with X a subset of its transitions ✓. Such restriction hides information though and applying it would be analogous to giving each transition a self-loop to a new place of its own with one token to guarantee that the PT-system is co-safe.
The full local independence relation $\mathtt{Lind}4$ of $\mathtt{PT}4$ has (among others) the following elements: $(\lambda, \varnothing)$, $(\lambda, \{\mathtt{a}\})$, $(\lambda, \{\mathtt{b}\})$, $(\lambda, \{\mathtt{c}\})$, $(\lambda, \{\mathtt{a},\mathtt{b}\})$, $(\lambda, \{\mathtt{a},\mathtt{c}\})$, $(\lambda, \{\mathtt{b},\mathtt{c}\})$, $(\mathtt{a}, \varnothing)$, $(\mathtt{a}, \{\mathtt{b}\})$, $(\mathtt{a}, \{\mathtt{c}\})$, $(\mathtt{a}, \{\mathtt{b},\mathtt{c}\})$ and $(\mathtt{abc}, \varnothing)$. This local independence relation is finite. In general, however, the local independence relations of PT-systems may be infinite, since these systems can have infinitely

many reachable markings (like `PT1` in Figure 5.19) or exhibit repetitive behaviour (like `PT2`).

An important observation is that all words which belong to the local trace of a firing sequence of a PT-system are indeed also realisable as firing sequences.

Fact 65 : $\mathit{firseq}(PT) = \bigcup_{u \in \mathit{firseq}(PT)} [\![u]\!]$.

Since (contact-free) EN-systems can be considered as PT-systems, it follows that they define, apart from their concurrency alphabet with its structural independence relation, also a local concurrency alphabet. The next fact demonstrates that these two views are consistent. (Note that the local independence induced by the structural independence relation of the EN-system should first be restricted to the actual firing sequences of the system.)

Fact 66 : Let EN be a contact-free EN-system with concurrency alphabet (T, Ind_{EN}), and let Lind_{EN} be its local independence relation. Then $\mathit{Lind}_{EN} = \mathit{Lind} \cap \{(u, X) \mid u \in \mathit{firseq}(EN) \wedge X \subseteq T\}$ where Lind is obtained from Ind_{EN} as described in Fact 62.

So far no restrictions at all have been imposed on local independence relations, e.g., with respect to internal consistency. Yet, when applied to actual concurrent systems — such as PT-systems — one might require or expect some suitable conditions to reflect the intended interpretation.

Fact 67 : Let $(\Sigma, \mathit{Lind}) = (T, \mathit{Lind}_{PT})$ be the local concurrency alphabet of PT, and $(u, X) \in \mathit{Lind}$.

- $Y \leq X$ implies $(u, Y) \in \mathit{Lind}$.
- $Y \leq X$ implies that $(ux, Y) \in \mathit{Lind}$ for all words $x \in \Sigma^*$ such that $\#_a(x) = X(a) - Y(a)$ for all $a \in \Sigma$.
- $u \equiv_{\mathit{Lind}} v$ implies $(v, X) \in \mathit{Lind}$.

The first two items above capture the fact that independent instances of transitions indeed occur independently from one another. They can be seen as a translation of the diamond property (Fact 54 for multisets) to the local independence relation. Thanks to the diamond property, the multisets in any multiset sequence of PT can be split yielding sequential representatives for each multiset sequence. It is, moreover, guaranteed that these representatives are local trace equivalent with each other which, as representatives of the same concurrent run of the net, they should be. The third item ensures that the local independence relation of PT 'agrees' with the local trace equivalence it defines. With this latter property, adding the information which multisets of symbols are concurrently enabled after (each prefix of) a local trace is a well-defined operation ✓. In this way, the prefix ordering of a single local trace, as well as the prefix ordering on the full set $\Sigma^*/_{\equiv_{Lind}}$ of all local traces over a local concurrency alphabet $(\Sigma, Lind)$, can be enhanced leading to multiset labelled transition systems resembling state graphs. Such 'trace graphs' provide all information on local independence and make it possible to distinguish between, e.g., $[\![\mathtt{abc}]\!]_{\mathtt{Lind4}}$ and $[\![\mathtt{abc}]\!]_{\mathtt{Lind6}}$ ✓. Another property satisfied by the local concurrency alphabet of a PT-system, but not listed above is that $(ua, \varnothing) \in Lind$ implies $(u, \{a\}) \in Lind$ for all words u and symbols a. This property reflects the prefix-closedness of the overall behaviour (in terms of its set of firing sequences) of the system.

We conclude this section by relating the local traces defined by a PT-system to its processes. It is not difficult to see that thanks to the diamond property of PT-systems, every process of a PT-system can be constructed (as described in Definition 58) from a firing sequence ✓. However, due to the possible multiplicity of tokens in places, this construction will in general yield more than one process per firing sequence (even when the PT-system is co-safe). Obviously, this implies that there is no one-to-one correspondence between local traces and processes. Now recall the swapping equivalence of processes which associates an equivalence class of processes with each single concurrent run of a PT-system. These swapping equivalence classes of processes are in one-to-one correspondence with the local trace equivalence classes of its firing sequences.

Fact 68 : Every word in the language of a process of PT belongs to one local trace of PT, and each local trace of PT is the union of the languages of a set of processes from PT forming one swapping equivalence class.

Consequently, with each local trace defined by a PT-system a finite set of partial orders can be associated. Each partial order is generated by a process and describes possible causalities in the concurrent execution which may depend on how the preceding history was observed, i.e., the choice of individual tokens during the execution. Consider, e.g., Figure 5.22 where two processes and their causality dags are given. Both processes can be constructed from the local trace equivalent firing sequences `agmru` and `gamru` of the PT-system `PT1` from Figure 5.19. Note that the transition `g` can both have an 'old' token or a 'new' token as its input.

Thus not distinguishing between multiple occurrences of a token in a place leads to a partial order semantics more complicated than that of EN-systems, both when based on processes or when employing a trace-based approach. Treating, however, multiple tokens in a place as individual entities would lead to a process semantics of PT-systems with the same expressiveness as the processes of EN-systems which seems counterintuitive.

5.6.5 Bibliographical Remarks

More background on PT-systems, their behavioural features and processes can be found in, e.g., [9, 18, 2, 3, 31]. Even though we have restricted ourselves to systems without arc weights, it was still possible to convey the key ideas underlying the causality semantics.

Local traces were originally proposed in [21, 23], and further developed in [25] with the aim to extend the semantic theory of EN-systems to the more general PT-systems. In the latter reference (see also [22, 24]) co-safe PT-systems are related to local event structures in a categorical setting. As a follow-up, [32, 33] study local traces as an independent notion that can be used to identify events and relations between them without having to rely on the Petri net model. This has led to local traces with sets of concurrent events rather than multisets, but otherwise defined as here. Also other definitions of context-dependent trace equivalence in the setting of a right-congruence were investigated in, e.g., [1, 4].

5.7 Concluding Remarks

This tutorial is an introduction to the much wider field of applying language theory to the study of concurrent behaviours, and so there are several strands of related research which have not even been mentioned. For example, it is possible to develop traces for infinite system behaviours [16, 17], which also allows one to treat aspects such as fairness [34]. Moreover, we have not considered the modelling of conflicts between enabled actions while traces and processes represent single runs in which all the conflicts have already been resolved. Adding conflict amounts to the introduction of branching in processes and considering the prefix ordering of all traces which form the system behaviours. (Branching processes of Petri nets [13] are the basis for an efficient verification technique [38, 14, 30].) If, in addition, one only considers relations between events (transition occurrences) the result is the more abstract model of event structures [24, 51, 40] which have been used to study fundamental concepts of concurrency in a model-independent way. Finally, we only briefly touched upon the algebraic properties of trace concepts such as can be found in, e.g., [11, 10].

Acknowledgements

We would like to thank Carlos Martín-Vide for encouraging us to prepare this tutorial. We are grateful to our colleagues, Grzegorz Rozenberg and Hendrik Jan Hoogeboom, for technical advice and references. This work has been supported by the Epsrc project Casino.

References

1. S. Bauget and P. Gastin. On congruences and partial orders. In Jirí Wiedermann and Petr Hájek, editors, *MFCS*, volume 969 of *Lecture Notes in Computer Science*, pages 434–443. Springer, 1995.
2. E. Best and R.R. Devillers. Sequential and concurrent behaviour in Petri net theory. *Theor. Comput. Sci.*, 55(1):87–136, 1987.
3. E. Best and C. Fernandez. *Non-sequential Processes, A Petri Net View.* Number 13 in EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Heidelberg, 1988.
4. I. Biermann and B. Rozoy. Reliable generalized and context dependent commutation relations. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT*, volume 1214 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 1997.
5. J. Billington. Protocol specification using P-graphs, a technique based on coloured Petri nets. In Reisig and Rozenberg [46], pages 293–330.
6. N. Busi and G.M. Pinna. Process semantics for place/transition nets with inhibitor and read arcs. *Fundam. Inform.*, 40(2-3):165–197, 1999.
7. P. Cartier and D. Foata. *Problèmes combinatoires de commutation et réarrangements*. Number 85 in Lecture Notes in Mathematics. Springer-Verlag, Heidelberg, 1969.
8. J. Desel and G. Juhás. "What is a Petri net?". In Hartmut Ehrig, Gabriel Juhás, Julia Padberg, and Grzegorz Rozenberg, editors, *Unifying Petri Nets*, volume 2128 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2001.
9. J.Desel and W. Reisig. Place/transition Petri nets. In Reisig and Rozenberg [45], pages 122–173.
10. V. Diekert and Y. Métivier. Partial commutation and traces. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 457–533. Springer-Verlag, Heidelberg, 1997.
11. V. Diekert and G. Rozenberg, editors. *The Book of Traces.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1995.
12. S. Donatelli and G. Franceschinis. Modelling and analysis of distributed software using GSPNs. In Reisig and Rozenberg [46], pages 438–476.
13. J. Engelfriet. Branching processes of Petri nets. *Acta Inf.*, 28(6):575–591, 1991.
14. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In Tiziana Margaria and Bernhard Steffen, editors, *TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 1996.
15. ftp://inf.informatik.uni stuttgart.de/pub/techreports/theorie/traces.bib.
16. P. Gastin. Infinite traces. In I. Guessarian, editor, *Proc. Spring School of Theoretical Computer Science on Semantics of Systems of Concurrent Processes*, number 469 in Lecture Notes in Computer Science, pages 277–308, Heidelberg, 1990. Springer-Verlag.
17. P. Gastin and A. Petit. Poset properties of complex traces. In I. M. Havel and V. Koubek, editors, *Proc. 17th Symposium on Mathematical Foundations of Computer Science (MFCS'92), Prague (Czechoslovakia), 1992*, number 629 in Lecture Notes in Computer Science, pages 255–263, Heidelberg, 1992. Springer-Verlag.
18. U. Goltz and W. Reisig. The non-sequential behavior of Petri nets. *Information and Control*, 57(2/3):125–147, 1983.

19. H.J. Hoogeboom and G. Rozenberg. Diamond properties of elementary net systems. *Fundam. Inform.*, 14(3):287–300, 1991.
20. H.J. Hoogeboom and G. Rozenberg. Dependence graphs. In V. Diekert and G. Rozenberg, editors, *The Book of Traces*, chapter 2, pages 43–67. World Scientific, Singapore, 1995.
21. P.W. Hoogers, H.C.M. Kleijn, and P.S. Thiagarajan. A trace semantics for Petri nets. In W. Kuich, editor, *Proc. 19th (ICALP'92), Vienna*, number 623 in Lecture Notes in Computer Science, pages 595–604, Heidelberg, 1992. Springer-Verlag.
22. P.W. Hoogers, H.C.M. Kleijn, and P.S. Thiagarajan. Local event structures and Petri nets. In Eike Best, editor, *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 462–476. Springer, 1993.
23. P.W. Hoogers, H.C.M. Kleijn, and P.S. Thiagarajan. A trace semantics for Petri nets. *Inf. Comput.*, 117(1):98–114, 1995.
24. P.W. Hoogers, H.C.M. Kleijn, and P.S. Thiagarajan. An event structure semantics for general Petri nets. *Theor. Comput. Sci.*, 153(1&2):129–170, 1996.
25. P.W. Hoogers. *Behavioural Aspects of Petri Nets.* Dissertation, Leiden University, 1994.
26. http://www.informatik.uni hamburg.de/TGI/PetriNets/.
27. R.J. and Maciej Koutny. Semantics of inhibitor nets. *Inf. Comput.*, 123(1):1–16, 1995.
28. G.Juhás, R. Lorenz, and T. Singliar. On synchronicity and concurrency in Petri nets. In Wil M. P. van der Aalst and Eike Best, editors, *ICATPN*, volume 2679 of *Lecture Notes in Computer Science*, pages 357–376. Springer, 2003.
29. R.M. Keller. Parallel program schemata and maximal parallelism I. Fundamental results. *Journal of the Association for Computing Machinery*, 20(3):514–537, 1973.
30. V. Khomenko, M. Koutny, and W. Vogler. Canonical prefixes of Petri net unfoldings. *Acta Inf.*, 40(2):95–118, 2003.
31. H.C.M. Kleijn and M. Koutny. Process semantics of general inhibitor nets. *Inf. Comput.*, 190(1):18–69, 2004.
32. H.C.M. Kleijn, R. Morin, and B. Rozoy. Event structures for local traces. *Electr. Notes Theor. Comput. Sci.*, 16(2), 1998.
33. H.C.M. Kleijn, R. Morin, and B. Rozoy. A general categorical connection between local event structures and local traces. In Gabriel Ciobanu and Gheorghe Paun, editors, *FCT*, volume 1684 of *Lecture Notes in Computer Science*, pages 338–349. Springer, 1999.
34. M.Z. Kwiatkowska. *Fairness for non-interleaving concurrency.* PhD Thesis, University of Leicester (UK), 1989.
35. A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.
36. A.W. Mazurkiewicz. Trace theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1987.
37. A.W. Mazurkiewicz. Basic notions of trace theory. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 354 of *Lecture Notes in Computer Science*, pages 285–363. Springer, 1989.
38. K.L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In Gregor von Bochmann and David K.

Probst, editors, *CAV*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer, 1992.
39. R. Milner. *A Calculus of Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 1980.
40. M. Nielsen, G.D. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. *Theor. Comput. Sci.*, 13:85–108, 1981.
41. M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Behavioural notions for elementary net systems. 4:45–59, 1990.
42. M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Elementary transition systems. *Theor. Comput. Sci.*, 96(1):3–33, 1992.
43. J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.
44. C.A. Petri. Fundamentals of a theory of asynchronous information flow. In *Proc. of IFIP Congress'62, North Holland, Amsterdam (1962)*, pages 386–390, 1962.
45. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer, 1998.
46. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets II: Applications*, volume 1492 of *Lecture Notes in Computer Science*. Springer, 1998.
47. G. Rozenberg and J. Engelfriet. Elementary net systems. In Reisig and Rozenberg [45], pages 12–121.
48. G. Rozenberg and P.S. Thiagarajan. Petri nets: Basic notions, structure, behaviour. Number 224 in Lecture Notes in Computer Science, pages 585–668, Heidelberg, 1986. Springer-Verlag.
49. E. Szpilrajn. Sur l'extension de l'ordre partiel. *Fundam. Math.*, 16:386–389, 1930.
50. W. Vogler. Partial order semantics and read arcs. *Theor. Comput. Sci.*, 286(1):33–63, 2002.
51. G. Winskel. Event structures. In W. Brauer, editor, *Petri nets: central models and their properties; advances in Petri nets; proceedings of an advanced course, Bad Honnef, 8.-19. Sept. 1986, Vol. 2*, number 255 in Lecture Notes in Computer Science, Heidelberg, 1986. Springer-Verlag.

6

Cellular Automata – A Computational Point of View

Martin Kutrib

Institut für Informatik, Universität Giessen
Arndtstr. 2, 35392 Giessen, Germany
kutrib@informatik.uni-giessen.de

Summary. The advantages of homogeneous arrays of interacting processing elements are simplicity and uniformity. It turned out that a large array of not very powerful elements operating in parallel can be programmed to be very powerful. One type of system is of particular interest: cellular automata whose homogeneously interconnected deterministic finite automata (the cells) work synchronously at discrete time steps obeying one common transition function. Cellular automata have extensively been investigated from different points of view. Here we discuss some of the main aspects from a computational point of view. The focus is on very simple types, that is, on one-dimensional cellular automata with nearest neighbor interconnections. In particular, we consider universality issues, the problem how to simulate data structures as stacks, queues, and rings without any loss of time, the famous Firing Squad Synchronization Problem, signals, and time constructible functions as well as several aspects of cellular automata as language acceptors. Some open problems are addressed.

6.1 Introduction

Cellular automata are an old branch of computer science. In the late forties of the last century they were proposed by John von Neumann in order to solve the logical problem of nontrivial self-reproduction. From this biological point of view he employed a mathematical device which is a multitude of interconnected automata operating in parallel to form a larger automaton, a macroautomaton built by microautomata. His famous early result reveals that it is logically possible for such a nontrivial computing device to replicate itself ad infinitum [72]. The name of these automata originates from the context in which they were developed. Due to their intuitive and colorful concepts, cellular automata have soon been considered from a computational point of view. So, from the very beginning, they were both, an interesting and challenging model for theoretical computer science and an interesting model for practical applications. Their inherent massive parallelism renders obvious applications as model for systems that are beyond direct measurements.

Cellular automata are a young branch of computer science. Besides applications in industry, nowadays they open up new fields of application and modeling of natural phenomena in physics, biology, chemistry as well as in sociology, economics, and other human sciences. The development of practical and theoretical issues of cellular automata is impressive. In particular, it seems that currently the studies from a theoretical point of view follow two main branches. One focuses on the global behavior of cellular automata. Based on some topology the space of configurations is investigated. An important challenge with practical aspects is the characterization of cellular automata on the basis of their global transition function.

The other branch may be seen to deal with information. The flexibility of cellular automata to serve as programming tools can be utilized to develop tricky algorithms in order to solve classical problems as well as problems concerning the very nature of the system itself. An example for the latter case is the problem of synchronization, which gave rise to intensive research. In this connection, sources of questions are complexity issues as well as classifications in terms of formal language recognition. These questions are objects of the present article. More precisely, cellular automata are seen from a computational point of view. The main focus is on one-dimensional cellular automata which are linear arrays of cells that are connected to their nearest neighbors. The cells are exactly in one of a finite number of states, which is changed according to local rules depending on the current state of a cell itself and the current states of its neighbors. The state changes take place simultaneously at discrete time steps.

The presented topics are far from being complete. From the many interesting ones only a few could be chosen. In the following Section 6.2 basic definitions and preliminaries are given. Higher-dimensional systems with arbitrary cell interconnections are introduced as generalizations of one-dimensional systems with the mentioned nearest neighbor connections. For unbounded cellular spaces universality is obtained. After presenting an approach to evidence based on the possibility to model logical gates and information transition in two-dimensional cellular spaces with the simple rules of the Game of Life, it is shown how cellular spaces can simulate Turing machines. Besides, investigations concerning universality (often combined with other properties) are done, for example, in [1, 10, 33, 45, 46, 52, 53, 54, 30]. A survey of universality and decidability versus undecidability in cellular automata and several other models of discrete computations can be found in [44]. Next we turn to show how to simulate stacks, queues, and rings by one-dimensional cellular automata without any loss of time. The simulations may serve as tools for designing algorithms or as subroutines for programming cellular automata [6, 29].

The famous Firing Squad Synchronization Problem is dealt with in Section 6.3. It was raised by Myhill in 1957 and emerged in connection with the problem to start several parts of a parallel machine at the same time. The first published reference appeared with a solution found by McCarthy and Minsky in [50]. Roughly speaking, the problem is to set up a cellular space such that all cells in a region change to a special state for the first time after the same number of steps.

Section 6.4 is devoted to the study of signals and constructibility of functions. Signals are used to solve problems. Examples are the basic signals that appear in solutions of the Firing Squad Synchronization Problem, or complex signals that allow to generate prime numbers. So, they can be seen as tools for algorithm design. In general, signals are able to transmit or encode information in cellular spaces. They have been used for a long time, but the systematic study originated from [49]. Basic questions are what kind of signals can be sent, or which speed is possible.

One of the main branches in the theory of cellular automata is considered in Section 6.5. Clearly, the data supplied to some device can be arranged as strings of symbols. Instances of problems to solve can be encoded as strings with a finite number of different symbols. Furthermore, complex answers to problems can be encoded as binary sequences such that the answer is computed bit by bit. In order to compute one piece of the answer, the set of possible inputs is split into two sets associated with the binary outcome. From this point of view, the computational capabilities of the devices are studied in terms of string acceptance, that is, the determination to which of the two sets a given string belongs. These investigations are done with respect to and with the methods of language theory. For cellular spaces and automata they originated from [11, 12] and [61, 31]. Over the years substantial progress has been achieved, but there are still some basic open problems with deep relations to other fields. A basic hierarchy of cellular language families is established, and the levels are compared with well-known families of the Chomsky hierarchy. The results are depicted in Figure 6.31. Closure properties are summarized in Table 6.1, and decidability problems are briefly discussed.

6.2 Basics and Preliminaries

We denote the set of integers by $\mathbb{Z}$ and the set of nonnegative integers by $\mathbb{N}$. The data supplied to the devices in question can be arranged as strings of symbols. In connection with formal languages, strings are called *words*. Let A^* denote the set of all words over a finite alphabet A. The *empty word* is denoted by λ, and we set $A^+ = A^* - \{\lambda\}$. For the *reversal of a word* w we write w^R, and for its *length* we write $|w|$. We use $\subseteq$ for *inclusions* and $\subset$ for *strict inclusions*.

6.2.1 Cellular Spaces

Basically, the idea of cellular automata is to form a massively parallel device as a multitude of interacting simple processing elements. In order to keep the systems tractable, a high degree of homogeneity is preferable. Moreover, the processing elements have to be chosen as simple as possible. So, the elements – which sometimes are called cells – are represented by finite Moore automata.

Due to the need for homogeneity all cells are identical. In addition, they are arranged as grid where one dimension, that is, a linear array whose cells are identified by integers, is of particular interest in the sequel. Homogeneous local communication structures are achieved by a unique interconnection scheme that defines the cells which are interconnected to a given cell. Eventually, the cells operate synchronously at discrete time steps obeying a local transition function, which maps the current state of the cell itself and the current states of its connected cells (neighbors) to the next state.

So, a multitude of finite automata operating in parallel form a larger automaton such that global results are achieved by local interactions only.

To be more precise, we define a cellular space formally. The notion *space* is due to the fact that, potentially, we have an infinite number of cells, that is, we deal with the entire Euclidean space $\mathbb{Z}$. In order to obtain two-way information flow we assume that each cell is connected to its both nearest neighbors.

Definition 1. *A* (one-dimensional) two-way cellular space (CS) *is a system* $\langle S, \delta, q_0, A, F \rangle$*, where*

1. *S is the finite, nonempty set of* cell states*,*
2. *$\delta : S^3 \to S$ is the* local transition function*,*
3. *$q_0 \in S$ is the* quiescent state *such that* $\delta(q_0, q_0, q_0) = q_0$*,*
4. *$A \subseteq S$ is the set of* input symbols*, and*
5. *$F \subseteq S$ is the set of* final states*.*

Basically, we have an infinite space but are interested in finite supports only. That is, beginning a computation with a finite number of non-quiescent cells, by definition we obtain only finitely many non-quiescent cells at every time step. This determines the role played by the quiescent state. The set of final states has been included with an eye towards applications.

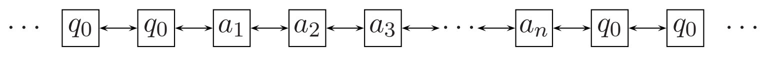


Fig. 6.1. A (one-dimensional) two-way cellular space.

In general, the global behavior of a cellular space is of interest. It is induced by the local behavior of all cells, that is, by the local transition function. More precisely, a *configuration* of a cellular space $\langle S, \delta, q_0, A, F \rangle$ at time $t \geq 0$ is a description of its global state, which is formally a mapping $c_t : \mathbb{Z} \to S$. The configuration at time 0 is defined by the given input $w = a_1 \cdots a_n \in A^+$, $n \geq 1$. We set $c_0(i) = a_i$, for $1 \leq i \leq n$, and $c_0(i) = q_0$ otherwise. Configurations may be represented as words over the set of cell states in their natural ordering, where the quiescent state is represented by the empty word. For example, the initial configuration for w is represented by $a_1 a_2 \cdots a_n$. Successor configurations are computed according to the global transition function Δ.

Let c_t, $t \geq 0$, be a configuration. Then its successor $c_{t+1} = \Delta(c_t)$ is defined by $c_{t+1}(i) = \delta(c_t(i-1), c_t(i), c_t(i+1))$, for all $i \in \mathbb{Z}$. A computation can be represented as *space-time diagram*, where each row is a configuration and the rows appear in chronological ordering.

An elementary technique in automata theory is the usage of multiple tracks. Basically, this means to consider the state set as Cartesian product of some smaller sets. Each component of a state is called *register*, and the same register of all cells together form a *track*.

In the sequel, for convenience and readability we may omit the definition of local transition functions for situations that do not change the state. Especially, we omit $\delta(q_0, q_0, q_0) = q_0$.

Example 1. The following cellular space $\mathcal{M} = \langle S, \delta, q_0, A, F\rangle$ reverses its input $w \in A^+$ in $|w|$ time steps (cf. Figure 6.2). It uses two tracks that are implemented by the state set $S = (A \cup \{\sqcup\})^2 \cup \{q_0\}$. Let (s_1, s_2), (s_3, s_4) and (s_5, s_6) be arbitrary states from $S \setminus \{q_0\}$.

$$\begin{aligned}
\delta(q_0, (s_3, s_4), q_0) &= (s_4, s_3)\\
\delta(q_0, (s_3, s_4), (s_5, s_6)) &= (s_5, s_3)\\
\delta((s_1, s_2), (s_3, s_4), q_0) &= (s_4, s_2)\\
\delta((s_1, s_2), (s_3, s_4), (s_5, s_6)) &= (s_5, s_2)
\end{aligned}$$

□

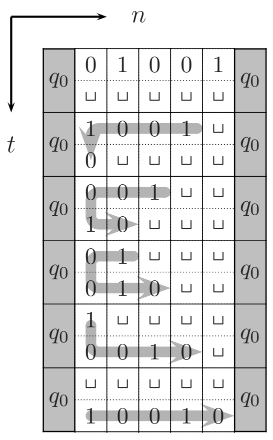


Fig. 6.2. Space-time diagram of a two-way cellular space reversing its input.

6.2.2 Important Generalizations

So far, cellular spaces have been introduced as one-dimensional arrays whose cells are connected to their immediate neighbors. Certainly, these types belong to the most important and natural ones. In particular, from a computational perspective they are best investigated. However, there are many generalizations which are just as interesting and natural. More generally speaking, the specification of a cellular space includes the type and specification of the cells, their interconnection scheme (which can imply a dimension of the system), the local rules which are formalized as local transition function, and the input and output modes. In the present subsection we briefly deal with two generalizations. First, we consider arbitrary unique interconnection schemes which are called *neighborhood-indices* and, secondly, devices whose cells are arranged as *d-dimensional grids*.

So, assume that the cells of a cellular space are arranged as d-dimensional grid such that we deal with the Euclidean space $\mathbb{Z}^d$.

Definition 2. *1. Let $d, k \geq 1$ be positive integers. A d-*dimensional neighborhood-index of degree *k is a k-tuple $N = (n_1, n_2, \dots, n_k)$ of different elements from $\mathbb{Z}^d$.*

2. Some cell $j \in \mathbb{Z}^d$ is called a neighbor *of cell $i \in \mathbb{Z}^d$, if there is a $k' \in N$ such that $j = i + k'$. The cells i and j are called* neighbors*, if either i is neighbor of j, or vice versa.*

In order to identify the neighbors of a cell i one has to add the elements of N to i. In particular, if 0 belongs to N, then cell i is its own neighbor. Only in this case the next state of a cell depends on its current state. Configurations are now mappings $c_t : \mathbb{Z}^d \to S$, and the global transition function Δ is induced by the local transition function $\delta : S^k \to S$ as follows:

$$c_{t+1} = \Delta(c_t) \iff c_{t+1}(i) = \delta(c_t(i + n_1), \dots, c_t(i + n_k)), \text{ for all } i \in \mathbb{Z}^d.$$

There are general methods that allow to simulate a cellular space by another one having a (reduced) standard neighborhood-index. So, it suffices to consider the most important standard ones. Whenever the ordering of the elements of a neighborhood-index does not matter, we may specify it as a set.

Example 2. Let $d \geq 1$, $k \geq 0$, and $m_1, \dots, m_d$ denote the components of $m \in \mathbb{Z}^d$. Then

$$H_k^d = \{m \in \mathbb{Z}^d \mid k \geq \textstyle\sum_{i=1}^d |m_i|\} \quad \text{or}$$
$$\bar{H}_k^d = \{m \in \mathbb{Z}^d \mid k \geq \textstyle\sum_{i=1}^d |m_i| \wedge m_i \geq 0, \text{ for } 1 \leq i \leq d\}$$

are (generalized) *von-Neumann* neighborhoods. Similarly,

$$M_k^d = \{m \in \mathbb{Z}^d \mid k \geq \max\{|m_i| \mid 1 \leq i \leq d\}\} \quad \text{or}$$
$$\bar{M}_k^d = \{m \in \mathbb{Z}^d \mid k \geq \max\{|m_i| \mid 1 \leq i \leq d\} \wedge m_i \geq 0, \text{ for } 1 \leq i \leq d\}$$

are (generalized) *Moore* neighborhoods (cf. Figure 6.3). □

The following famous cellular space is known as *Game of Life*. While the underlying rules are quite simple, the global behavior is rather complex. In fact, it is unpredictable.

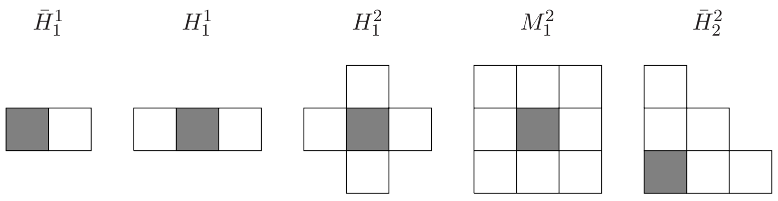


Fig. 6.3. Standard neighborhoods (the origin is shaded).

Example 3. We consider the two-dimensional space $\mathbb{Z}^2$. The cells are connected according to the Moore-neighborhood M_1^2, where each cell is connected to itself and to its eight immediate neighbors. Cells may be dead or alive, so the state set is chosen to be $\{0, 1\}$. The local transition function is defined dependent on the number of living cells in the neighborhood. In particular, a cell stays or becomes alive, if there are exactly three living cells within its Moore-neighborhood. It stays in its current state, if there are exactly four living cells within its Moore-neighborhood, and it dies from overpopulation or isolation otherwise.

The Game of Life made its first appearance in [16]. Over the years very interesting properties have been discovered. Some of them are based on the behavior of patterns that represent the arrangement of dead and living cells (cf. Figures 6.4 and 6.5). □

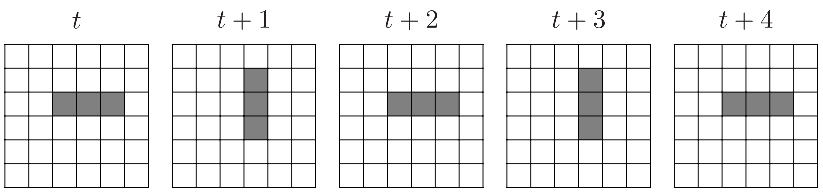


Fig. 6.4. Evolution of a periodical stationary pattern (blinker) in the Game of Life. Living cells are shaded.

6.2.3 Universality

In order to explore the power of general cellular spaces, we are now going to prove their universality. To this end, it is shown how cellular spaces can simulate Turing machines. Moreover, given a Turing machine, the corresponding cellular space should be as simple as possible. Therefore, we present a direct simulation by a one-dimensional space, where the number of states depends on the number of states and tape symbols of the Turing machine [30].

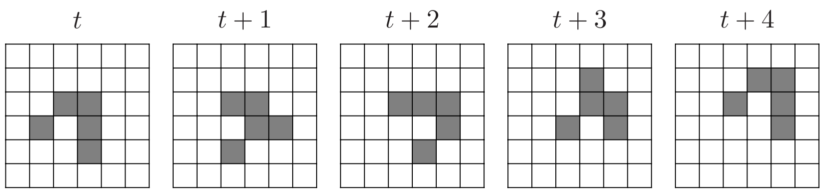


Fig. 6.5. Evolution of a periodical non-stationary pattern (glider) in the Game of Life. Living cells are shaded. Within four time steps the glider moves diagonally one cell to the north east.

But first we have another approach to evidence based on the generalizations. Roughly speaking, the idea is to model logical gates and information transmission in two-dimensional cellular spaces. Then universal computers can be build and embedded into the space. Interestingly, the constructions can be done with the simple rules of the Game of Life. So, two states are sufficient [4].

A stream of information is modeled as stream of bits. Consider a stream of gliders moving with the same space between, and assume some of the gliders are missing. Then the stream can be interpreted as stream of bits where the presence of a glider means 1 and the absence means 0. The following pattern depicted in Figure 6.6 is known as *glider gun*. The core of the gun behaves

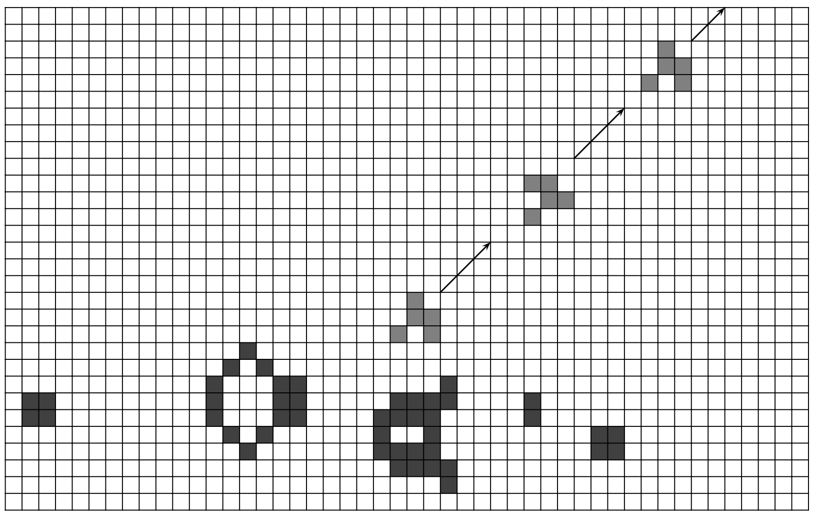


Fig. 6.6. Evolution of a glider gun in the Game of Life. Living cells are shaded. Within 30 time steps a glider is emitted to the north east. The arrows indicate the direction of the stream of gliders.

periodical. In addition, it emits a glider every 30 time steps. So, a glider gun can be seen as a source of a stream of bits consisting of ones only.

Now we turn to logical gates. In order to obtain a NOT gate, one observes that whenever two gliders collide at a right angle, then all wreckage disappears. So, the input stream to negate can be directed to a bit stream emitted by a glider gun. If a 1 (a glider) of the input stream reaches the collision area, it will collide with the incoming glider from the gun and is destroyed. If a 0 of the input stream reaches the collision area, the incoming glider will pass the collision area. In this way a 1 yields to a 0, and vice versa (cf. Figure 6.7).

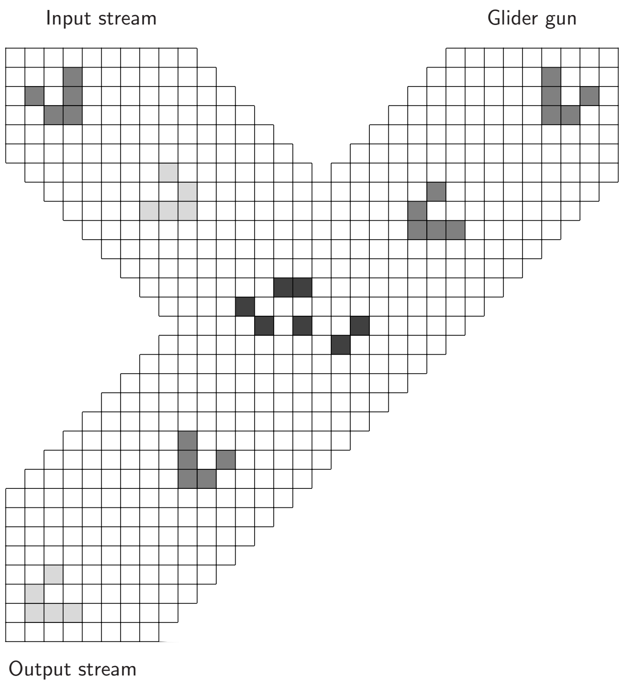


Fig. 6.7. A NOT gate in the Game of Life. Living cells are shaded. The cells shaded lightgray are not alive. They indicate the missing glider representing a 0.

Similarly, AND and OR gates are constructed. Figure 6.8 shows the schematic diagrams, where G means glider gun, and E is a pattern called *eater*. An eater absorbs incoming gliders.

The universality of cellular spaces follows since universal computers can be build from logical gates and bit streams. These computers can be embedded into the space. But it is worth mentioning that the effective construction requires to start with finite configurations of the cellular space. On the other hand, the computers may use potentially infinite memory. Nevertheless, by

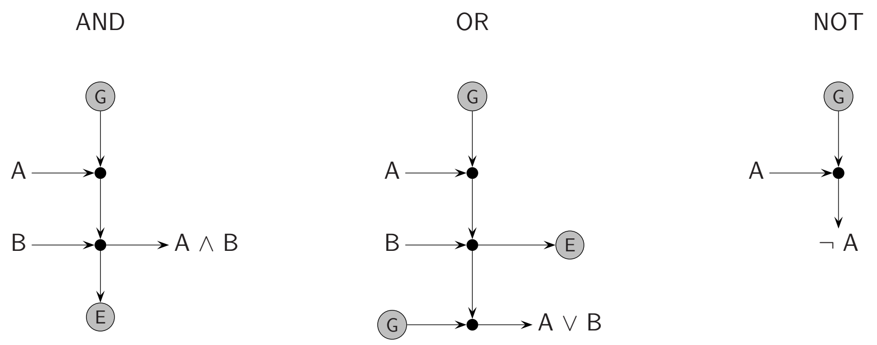


Fig. 6.8. Schematic diagrams of logical gates in the Game of Life. Glider guns and eaters are denoted by G and E.

nontrivial constructions it is possible to extend the available memory on demand of the computation [4].

Next we show how to simulate an arbitrary Turing machine by a one-dimensional cellular space with von-Neumann H_1^1 neighborhood, where the number of states depends on the number of states and tape symbols of the Turing machine. Since the Turing machine is arbitrary, in particular, the simulation of universal Turing machines is possible. There are universal Turing machines, for example, with four states and six tape symbols [56]. So, the next theorem gives also an upper bound on the size necessary for a (universal) cellular space.

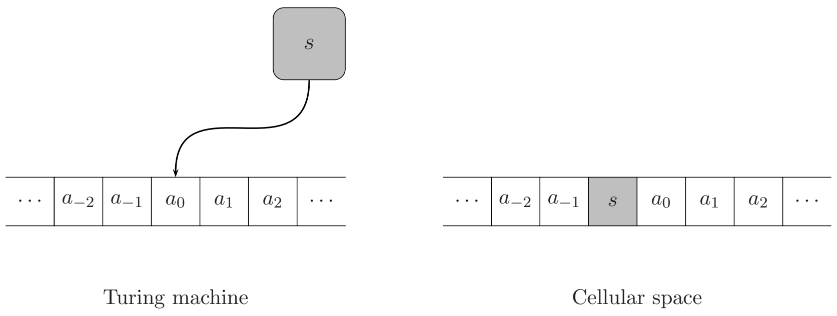
Theorem 1. *Let $\mathcal{T} = \langle S, T, \delta, s_0, \sqcup \rangle$ be a one-tape Turing machine with state set S, tape symbols T, transition function δ, initial state s_0, and blank symbol $\sqcup$. Then there is a cellular space $\mathcal{M}$ with $|T|+4|S|$ states, that simulates $\mathcal{T}$ in twice the time.*

Proof. Without loss of generality, we assume that S and T are disjoint. Each symbol of the tape inscription is stored in one cell of $\mathcal{M}$. The left neighbor of the cell storing the currently scanned tape symbol represents the current state of $\mathcal{T}$ (cf. Figure 6.9).

At first glance, due to the H_1 neighborhood of the cellular space, the problem arises that a possible left move of the head cannot be observed by the cell at the left of the cell representing the state of $\mathcal{T}$. But an intermediate step can solve the problem. In particular, the cell representing the state of $\mathcal{T}$ changes to some new state that indicates the next state as well as the intended head movement. For simplicity, we do the same for right moves and no moves. The formal construction of $\mathcal{M} = \langle S', \delta', q_0, A, F \rangle$ is as follows:

$$S' = S \cup T \cup (S \times \{\mathit{stay}, \mathit{right}, \mathit{left}\})$$

The local transition function δ' is defined dependent on δ. Let $s, s' \in S$ and $a, a' \in T$. For all $a_1, a_2 \in T$,



s
$\cdots$ a_{-2} a_{-1} a_0 a_1 a_2 $\cdots$
$\cdots$ a_{-2} a_{-1} s a_0 a_1 a_2 $\cdots$
Turing machine
Cellular space

Fig. 6.9. Correspondent configurations of a Turing machine and a simulating cellular space.

$$
\begin{aligned}
\delta(s,a) = (s',a',\mathit{stay}) \Longrightarrow (\; & \delta'(a_1,s,a) = (s',\mathit{stay}),\\
& \delta'(s,a,a_1) = a',\\
& \delta'(a_1,(s',\mathit{stay}),a') = s'),\\
\delta(s,a) = (s',a',\mathit{right}) \Longrightarrow (\; & \delta'(a_1,s,a) = (s',\mathit{right}),\\
& \delta'(s,a,a_1) = a',\\
& \delta'(a_1,(s',\mathit{right}),a') = a',\\
& \delta'((s',\mathit{right}),a',a_1) = s'),\\
\delta(s,a) = (s',a',\mathit{left}) \Longrightarrow (\; & \delta'(a_1,s,a) = (s',\mathit{left}),\\
& \delta'(s,a,a_1) = a',\\
& \delta'(a_1,(s',\mathit{left}),a') = a_1,\\
& \delta'(a_2,a_1,(s',\mathit{left})) = s').
\end{aligned}
$$

In all other situations cells do not change their states. □

6.2.4 Simulation of Data Structures

This subsection is devoted to show how to simulate certain data structures by (one-dimensional) cellular spaces without any loss of time. The simulations may serve as tools for designing algorithms or as subroutines for programming cellular spaces. First we consider pushdown stores (stacks) [6, 29], that is, stores obeying the principle *last in first out*. Assume without loss of generality that at most one symbol is pushed onto or popped from the stack at each time step. We distinguish one cell that simulates the top of the pushdown store. It suffices to use three additional tracks for the simulation. Let the three pushdown registers of each cell be numbered one, two, and three from top to bottom, and suppose that the third register is connected to the first register of the right neighbor. The content of the pushdown store is identified by scanning the registers in their natural ordering beginning in the distinguished cell, whereby empty registers are ignored (cf. Figure 6.10).

The pushdown store dynamics of the transition function is defined such that each cell prefers to have only the first two registers filled. The third

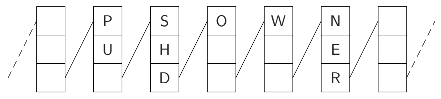


Fig. 6.10. Pushdown registers exemplarily storing the string *PUSHDOWNER*.

register is used as a buffer. In order to reach that charge it obeys the following rules (cf. Figure 6.11).

1. If all three registers of its left (upper) neighbor are filled, it takes over the symbol from the third register of the neighbor and stores it in its first register. The old contents of the first and second registers are shifted to the second and third register.
2. If the second register of its left neighbor is free, it erases its own first register. Observe that the erased symbol is taken over by the left neighbor. In addition, the cell stores the content of its second register into its first one, if the second one is filled. Otherwise, it takes the symbol of the first register of its right neighbor, if this register is filled.
3. Possibly more than one of these actions are superimposed.

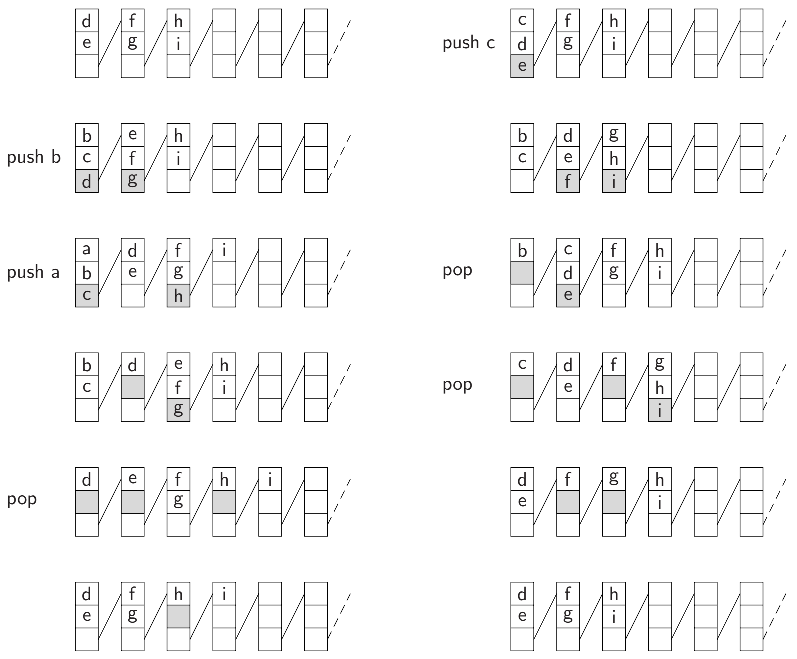


Fig. 6.11. Principle of a pushdown store simulation. Subfigures are in row-major order.

The main difference between pushdown stores and rings or queues is the way how to access the data. A *ring* obeys the principle *first in first out*, that is, the first symbol of the stored string is read and possibly erased while, in addition, a new symbol may be added at the end of the string. So, a ring can write and erase at the same time. A *queue* is a special case of a ring. It can either write or erase a symbol, but not both at the same time. In order to simulate a ring or queue, also no more than three additional registers are needed. The first two registers are used to store the symbols, where the second one is needed to cope with the situation when symbols are erased consecutively. The third track is used to move the new symbols from the front to the back of the string (cf. Figure 6.12).

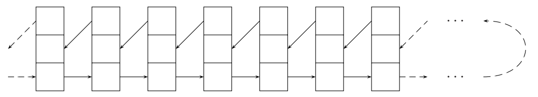


Fig. 6.12. Logical connections between ring registers.

Again, without loss of generality, we may assume that at most one symbol is entered to or erased from the ring at every time step. Moreover, each cell prefers to have the first two registers filled. Altogether, it obeys the following rules (cf. Figure 6.13).

1. If the third register of its left neighbor is filled, it takes over the symbol from that register. The cell stores the symbol into its first free register, if possible. Otherwise, it stores the symbol into its own third register.
2. If the third register of its left neighbor is free, it marks its own third register as free.
3. If the second register of its left neighbor is free, it erases its own first register. Observe that the erased symbol is taken over by the left neighbor. In addition, the cell stores the content of its second register into its first one, if the second one is filled. Otherwise it takes the symbol of the first register of its right neighbor, if this register is filled.
4. If the second register of its left neighbor is filled and its own second register is free, then the cell takes the symbol from the first register of its right neighbor and stores it into its own second register.
5. Possibly, more than one of these actions are superimposed.

6.3 Synchronization

The famous *Firing Squad Synchronization Problem* (FSSP) was raised by Myhill in 1957. It emerged in connection with the problem to start several parts of a parallel machine at the same time. The first published reference

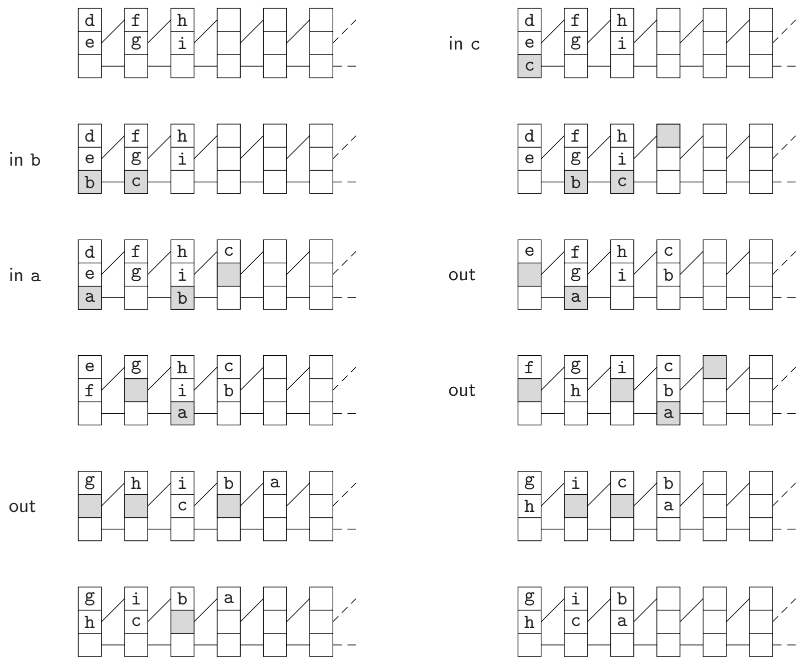


Fig. 6.13. Principle of a ring (queue) simulation. Subfigures are in row-major order.

appeared with a solution found by McCarthy and Minsky in [50]. Roughly speaking, the problem is to set up a cellular space such that all cells in a region change to a special state for the first time after the same number of steps. Originally, the problem has been stated as follows: Consider a finite but arbitrary long chain of finite automata that are all identical except for the automata at the ends. The automata are called soldiers, and the automaton at the left end is the general. The automata work synchronously, and the state of each automaton at time step $t+1$ depends on its own state and on the states of its both immediate neighbors at time step t. The problem is to find states and state transitions such that the general may initiate a synchronization in such a way that all soldiers enter a distinguished state, the firing state, for the first time at the same time step. At the beginning all non-general soldiers are in the quiescent state. More formally, the FSSP is defined as follows.

Definition 3. *Let C be the set of all cellular space configurations of the form $\texttt{#}gq_0\cdots q_0\texttt{#}$, that is, for some $n \geq 1$, $c(0) = c(n+1) = \texttt{#}$, $c(1) = g$ and $c(i) = q_0$, for $i \notin \{0,1,n+1\}$. The* Firing Squad Synchronization Problem *is to specify a cellular space $\langle S, \delta, q_0, A, F\rangle$ such that for all $c \in C$,*

1. *there is a $t \geq 1$ such that $\big(\Delta^t(c)\big)(i) = f$, for $1 \leq i \leq n$ and some $f \in S$,*
2. *for all $0 \leq t' < t$ it holds $\big(\Delta^{t'}(c)\big)(i) \neq f$, for $1 \leq i \leq n$, and*
3. *$\delta(q_0, q_0, \texttt{#}) = \delta(\texttt{#}, q_0, q_0) = \delta(q_0, q_0, q_0) = q_0$.*

While the first solution of the problem takes $3n$ time steps to synchronize the n cells in between the cells in state #, Goto [18] was the first who presented a minimal time solution.

Lemma 1. *The minimal solution time for the FSSP is $2n-2$, where n is the number of cells to be synchronized.*

Proof. In contrast to the assertion assume there is a faster solution taking some time $t_f < 2n-2$. Observe that the cells which are initially in the quiescent state may leave the quiescent state not before their left neighbor is in a non-quiescent state. Therefore, the rightmost cell n cannot leave the quiescent state before time $n-1$. It takes another $n-1$ time steps to send a feedback of this activation back to the general. Since $t_f < 2n-2$, the general fires independently of such a feedback.

Now consider the problem with $2n-1$ cells. Since the cells are deterministic, the general fires again at time $t_f < 2n-2$. But at this time step the rightmost cell $2n-1$ is still in the quiescent state, since it takes at least $2n-2$ time steps to activate it. □

Next we present an algorithm that is not time optimal. It takes $3n$ time, but reveals basic procedural methods.

Algorithm 1. The FSSP can be solved by dividing the array in two, four, eight etc. parts of (almost) the same length until all cells are cut-points. Exactly at this time the cells change to the firing state synchronously. The divisions are performed recursively. First the array is divided into two parts. Then the process is applied to both parts in parallel, etc.

In order to divide the array into two parts, the general sends two signals $S1$ and $S2$ to the right (cf. Figure 6.14). Signal $S1$ moves with speed 1, that is, one cell per time step, and signal $S2$ with speed 1/3, that is, one cell every three time steps. When signal $S1$ reaches the right end, a signal $S3$ is sent back to the left with speed 1. Signals $S2$ and $S3$ meet in the center of the array. Dependent on whether the length of the array is even or odd the center is represented by two or one cell. Next, the center cell(s) becomes a general. It sends signals $S1$ and $S2$ to the left and to the right. This process repeats until all cells are generals. At this time they change to the firing state synchronously.

Since the times needed to divide the sub-arrays are bounded by $3n/2$, $3n/4$, $3n/8$, and so on, altogether the algorithm takes at most $3n$ time steps. □

The next step towards a time optimal solution is to set up additional signals in order to determine the cut-points earlier.

Algorithm 2. The previous algorithm is modified as follows (cf. Figure 6.15). When signal $S1$ arrives at the right end, the end cell becomes a general and sends two signals $S3$ and $S4$ to the left. Signal $S4$ behaves as signal $S2$ except for the moving direction, that is, it moves with speed 1/3 to the left. The center

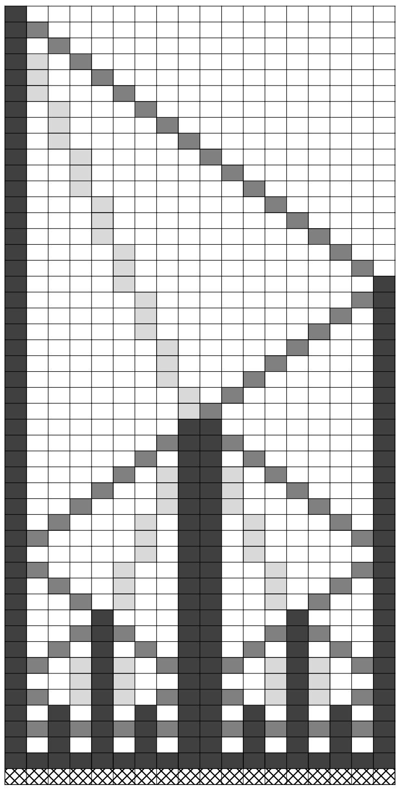


Fig. 6.14. Firing Squad Synchronization with a slow algorithm. Darkgray cells are generals, gray cells contain a signal with speed 1, lightgray cells a signal with speed 1/3, and crosshatched cells are in the firing state.

of the array is again determined by the collision of $S2$ and $S3$. The center cell(s) behaves as for the previous algorithm. In particular, it sends signals $S1$ and $S3$ to the right. The collision of $S1$ and $S4$ determines the center of the right half of the array after $3n/2 + n/4$ time steps. After another $n/8$ time steps the center of the third quarter of the array is known. If the remaining cut-points could be determined similarly, the total synchronization time would not exceed $2n$ time steps: $3n/2 + n/4 + n/8 + n/16 + \cdots = 2n$. Since without general there are only $n - 1$ cells to be passed through, the synchronization obeys the optimal time bound $2(n - 1)$.

Unfortunately, the presented procedure is not a solution, since only one of two cut-points is found, respectively. Clearly, one can determine the center of the left half of the array, if the general sends an additional signal $S5$ with speed 1/7 at initial time to the right. But then the next problem is to find the center of the left quarter of the array. To this end, the general can send another signal with speed 1/15 to the right. Altogether, for a solution the

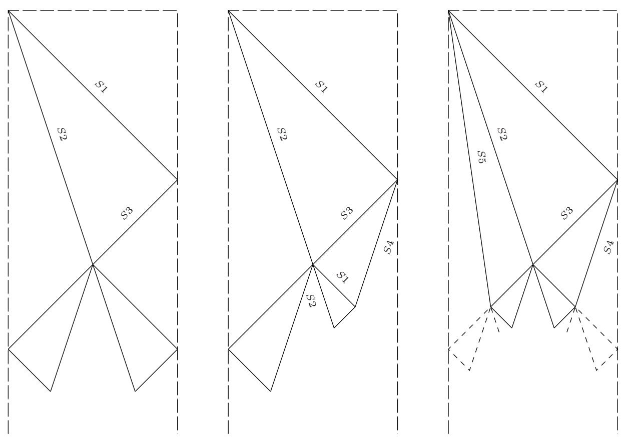


Fig. 6.15. Schematic diagrams of signals. Slow FSSP algorithm (left), additional signals for right cut-points (center), and additional signals for left cut-points (right).

general has to send signals with speeds $1/(2^k - 1)$, $k \geq 1$. Thus, the number of signals depends on the length of the array, and the problem is not solved. □

Nevertheless, there is a solution based on this approach [73]. The idea is rather simple, the additional signals are generated and moved by trigger signals (cf. Figure 6.16). The trigger signals themselves are emitted by signals $S1$ and $S3$ in the opposite direction at each other time step. Whenever a trigger signal reaches the leftmost or rightmost cell, a new signal to be triggered is generated. Whenever a trigger signal reaches a triggered signal, the latter is moved one cell ahead. On the other hand, any triggered signal absorbs each other trigger signal. That way, the desired behavior is achieved, and a minimal time solution for the FSSP is obtained.

Apart from time optimality there is a natural interest in efficient solutions with respect to the number of states or the number of bits to be communicated to neighbors. While there exists a time optimal solution where just one bit of information is communicated [47], the minimal number of states is still an open problem. The first time optimal solution [18] uses several thousand states. The presented algorithm from [73] takes 16 states. About one year later, an eight state time optimal solution was published [3]. Currently, a six state solution is known [48]. In the same paper it is proved that there does not exist a time optimal four state algorithm. It is a challenging open problem to prove or disprove that there exists a five state solution.

Many modifications and generalizations of the FSSP have been investigated. Just to mention a few of them, solutions for higher dimensions can be found in [19, 57, 60, 63, 70], fault tolerant synchronizations are studied in [41, 67], generalized positions of the general are considered in [51], and

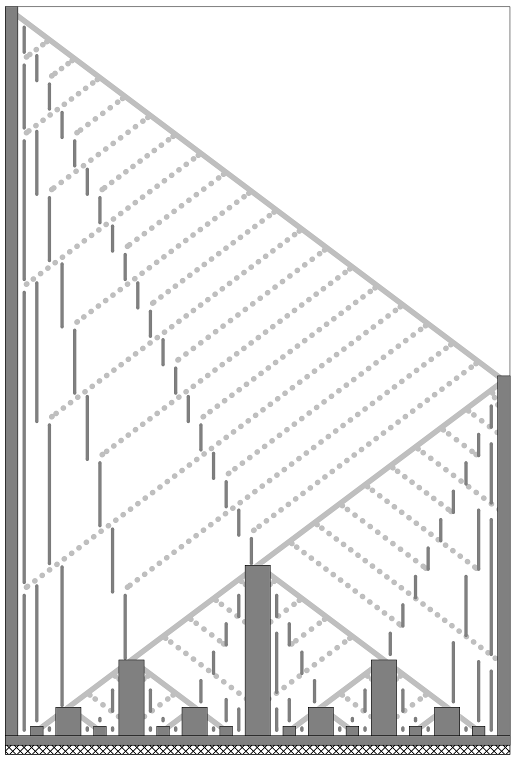


Fig. 6.16. Firing Squad Synchronization with a time optimal algorithm using trigger signals.

growing squads in [21]. In [32] the problem is solved for reversible cellular spaces, and in [27, 35, 38, 39] more general graphs are considered.

6.4 Signals and Time Constructibility

Signals are used to solve problems. Examples are the basic signals that appear in solutions of the FSSP, or complex signals that allow to generate prime numbers. So, they can be seen as tools for algorithm design. In general, signals are used to transmit or encode information in cellular spaces. They have been used for a long time, but the systematic study originated from [49]. Basic questions are what kind of signals can be send, or what speed is possible.

6.4.1 Signals

Roughly speaking, signals are described as follows: If some cell changes to the state s of its neighbor after some $k \geq 1$ time steps, and if subsequently its neighbors and their neighbors do the same, then the basic signal s moves with speed $1/k$ in the corresponding direction.

By this description it becomes intuitively clear what signals are. But the concept is much more complex. So a formal treatment is advisable. Obviously, the maximal speed is one, that is, one cell per time step. Signals are formalized as mappings, where the signal is distinguished from its implementation, since not every mapping of the appropriate type can be implemented. The mapping takes a time step and yields the cell in which the signal resides at this time.

Definition 4. *A* signal *is a mapping* $\xi : \mathbb{N} \to \mathbb{Z}$, *where for all* $t \geq 0$, $\xi(t+1) \in \{\xi(t)-1, \xi(t), \xi(t)+1\}$.

The current site of an implemented signal is indicated by special states.

Definition 5. *A signal* ξ *is* CS-practicable, *if there is a cellular space* $\langle S, \delta, q_0, A, F\rangle$ *with distinguished state* $s \in S$, *subset* $S' \subseteq S$, *and initial configuration* $c_0(0) = s$, $c_0(i) = q_0$, *for* $i \neq 0$, *such that* $c_t(i) \in S' \iff \xi(t) = i$.

It is evident that there are simple and complex signals. In general, auxiliary signals are needed in order to implement complex ones. Signal ξ is said to be *basic*, if the sequence of elementary moves $(\xi(t+1) - \xi(t))_{t \geq 0}$ is ultimately periodic. It is *rightmoving* (*leftmoving*), if it never moves to the left (right), that is, $\xi(t+1) \in \{\xi(t), \xi(t)+1\}$ $(\xi(t+1) \in \{\xi(t)-1, \xi(t)\})$.

Example 4. The signal $\xi : \mathbb{N} \to \mathbb{Z}$ with $\xi(n) = \lfloor \frac{n}{3} \rfloor$ is basic, since the sequence $0, 0, 1, 0, 0, 1, \ldots$ of elementary moves is periodic (cf. Figure 6.17). □

Example 5. The signal $\xi : \mathbb{N} \to \mathbb{Z}$ with

$$\xi(0) = 0 \quad \text{and} \quad \xi(n) = \frac{1}{4} \cdot 2^{\lceil \log_2 n \rceil} - \left| n - \frac{3}{4} \cdot 2^{\lceil \log_2 n \rceil} \right|$$

is obviously not basic (cf. Figure 6.18). □

The next lemma clarifies the relation between basic signals and implementations.

Lemma 2. *A signal* ξ *is basic if and only if it can be implemented in a cellular space such that all cells not containing* ξ *are in the quiescent state* ($i \neq \xi(t) \iff c_t(i) = q_0$).

With other words, a signal is basic if and only if it can be implemented without auxiliary signals.

Definition 6.

1. *Let* ξ *be a basic signal whose sequence of elementary moves after some time* n_0 *is periodic with period length* p. *Let* $u = \xi(t+p) - \xi(t)$, *for some* $t > n_0$.

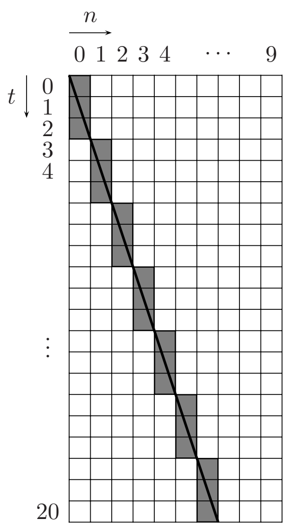


Fig. 6.17. The basic signal of Example 4.

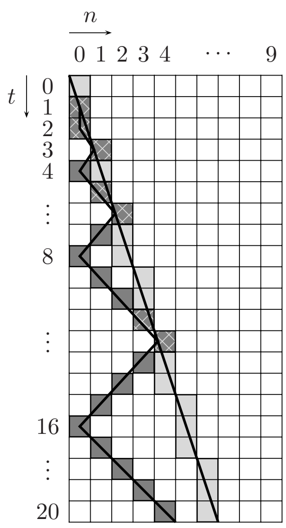


Fig. 6.18. Gray cells contain the signal of Example 5, lightgray cells a basic auxiliary signal.

   a) The slope *of ξ is p/u.*
   b) The speed *of ξ is u/p.*
2. *A monotone increasing (decreasing) function $\rho : \mathbb{N} \to \mathbb{N}$ is called* characteristic function *of a rightmoving (leftmoving) signal ξ, if $\xi(\rho(n)) = n$ and $\xi(\rho(n) - 1) \neq n$.*

Since the speed is at most 1, the slope is at least 1. The characteristic function takes a cell and yields the time step at which the signal arrives at the cell for the first time. Clearly, $\rho(n) \geq n$ for a characteristic function of a CS-practicable signal that is generated in cell 0.

6.4.2 Practicable Signals

In order to obtain a rich family of practicable signals we first show that certain classes of signals are practicable. Then we provide operations that preserve this property. So, one can construct new practicable signals from practicable ones by applying the operations.

Signals with exponential characteristic function

Lemma 3. *Let $b \geq 2$ be a positive integer. Then the signal ξ with characteristic function b^n is CS-practicable.*

Proof. At initial time signal ξ resides in cell 0. At each time step b^n, $n \geq 1$, it moves one cell to the right. To this end, two auxiliary signals α and β are used. In general, signals with speed $\frac{y}{x} \leq 1$ may be implemented by alternating y right moves and $x - y$ no moves. Signal α is generated at time $b - 2$ in cell 0. Signal β is generated at time $\frac{1}{2}(b^2 + b - 2)$ in cell $\frac{1}{2}b(b - 1)$ (cf. Figure 6.19). Whenever ξ meets α, signal ξ stays for one time step and then moves one cell to the right. Signal α also stays for one time step, and then it starts to move right with speed 1 until it meets β. Next, it moves back to the left with speed 1 until it meets ξ again. Initially and whenever β meets α, signal β moves b cells to the right within $b + 1$ time steps. Subsequently, it moves with speed $\frac{(b-1)}{(b+1)}$ to the right.

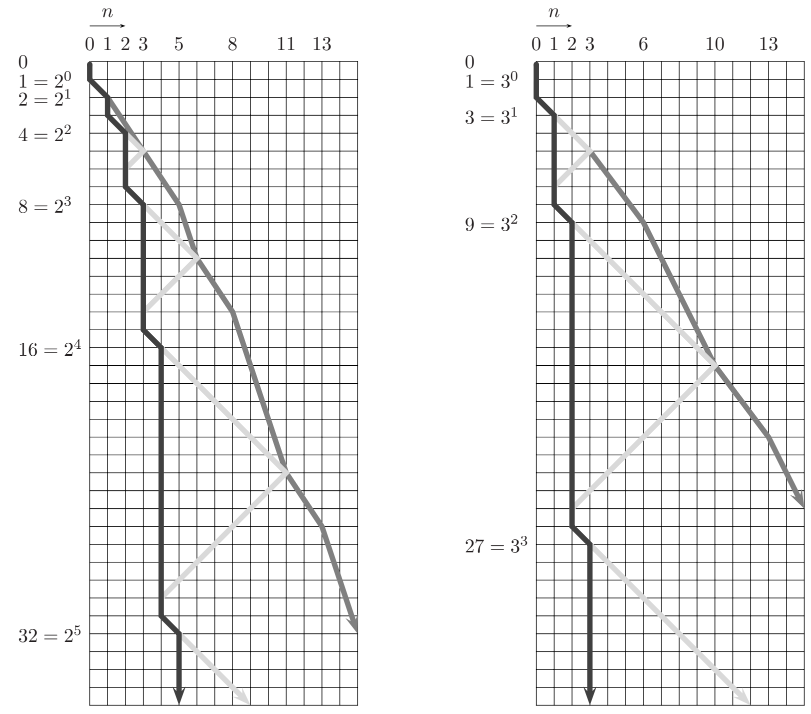


Fig. 6.19. Signals ξ with characteristic functions 2^n and 3^n (darkgray), auxiliary signals α (lightgray) and β (gray).

Exemplarily, the correctness of the construction is shown by induction. It is proved that α meets ξ at time $b^n - 2$ in cell $n - 1$ and, subsequently, meets β at time $\frac{1}{2}(b^{n+1} + b^n - 2)$ in cell $n - 1 + \frac{1}{2}(b^n(b - 1))$.

The induction basis $n = 1$ follows immediately from the generations of the signals. Assume now, the assertion is true for some $n \geq 1$. After meeting β, signal α meets ξ at time $\frac{b^{n+1}+b^n-2}{2} - 1 + \frac{b^n(b-1)}{2} = b^{n+1} - 2$ in cell n. At time $b^{n+1}-1$ both signals stay in cell n. Subsequently, at time b^{n+1} they move to cell $n+1$. Next, signal α passes through cells $n+1+k$ at time steps $b^{n+1}+k$, $k=1,2,\dots$ Especially for $k=-1+\frac{1}{2}(b^{n+1}(b-1))$, signal α is in cell $n+\frac{1}{2}(b^{n+1}(b-1))$ at time $\frac{1}{2}(b^{n+2}+b^{n+1}-2)$.

After its last meeting with α, signal β first has moved b cells to the right within $b+1$ time steps. Next it started to move with speed $\frac{(b-1)}{(b+1)}$ to the right. Therefore, it passes through cells $n-1+\frac{1}{2}(b^n(b-1))+b+k(b-1)$ at time steps $\frac{1}{2}(b^{n+1}+b^n-2)+b+1+k(b+1)$, $k=1,2,\dots$ Especially for $k=\frac{1}{2}(b^{n+1}-b^n-2)$, signal β is in cell $n+\frac{1}{2}(b^{n+1}(b-1))$ at time $\frac{1}{2}(b^{n+2}+b^{n+1}-2)$. □

Signals with polynomial characteristic function

A signal with characteristic function n^2 can be derived from $(n+1)^2 = n^2+2n+1$. In particular, before signal ξ may move from cell n to $n+1$ it has to stay for $2n$ time steps in cell n. The delay is exactly the time needed by an auxiliary signal α that moves from cell n to cell 0 and back (cf. Figure 6.20). Proceeding inductively, a signal with characteristic function n^b can be implemented by utilizing auxiliary signals with polynomial characteristic functions whose degrees are less than b.

Lemma 4. *Let $b \geq 1$ be a positive integer. Then the signal with characteristic function n^b is CS-practicable.*

Proof. Exemplarily, the construction for $b=3$ is shown, where an auxiliary signal with characteristic function n^2b is used (cf. Figure 6.21). Constructions for arbitrary b are straightforward.

First, we derive $(n+1)^3 = n^3+3n^2+3n+1$, and obtain the necessary time of delay. A signal with characteristic function n^3 has to stay for $3n^2+3n$ time steps in cell n before it moves to cell $n+1$. The delay $3n$ is exactly the time needed by an auxiliary signal α that moves from cell n to cell 0 and back, and once more to cell 0. Subsequently, in cell 0 a quadratic signal β is generated, which moves from cell 0 to cell n and back, and once more to cell n. □

Signals whose characteristic functions contain square roots

The problem whether the following lemma is true for $k=1$ was left open in [49]. It has been solved in [66].

Lemma 5. *Let $k \geq 1$ be a positive integer. Then the signal with characteristic function $kn+\lfloor\sqrt{n}\rfloor$ is CS-practicable.*

Signals whose characteristic functions contain logarithms

Lemma 6. *Let $b \geq 2$ be a positive integer. Then the signal with characteristic function $n+\lfloor\log_b(n)\rfloor$ is CS-practicable.*

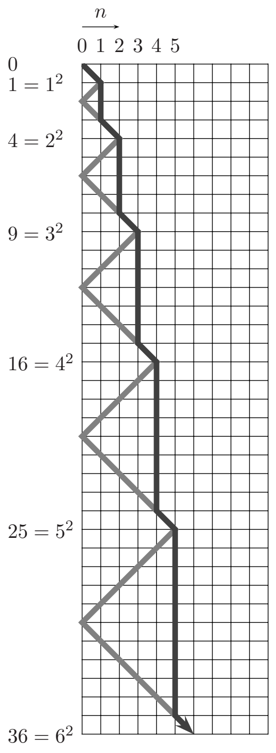


Fig. 6.20. Signal ξ with characteristic function n^2 (darkgray), auxiliary signal α (gray).

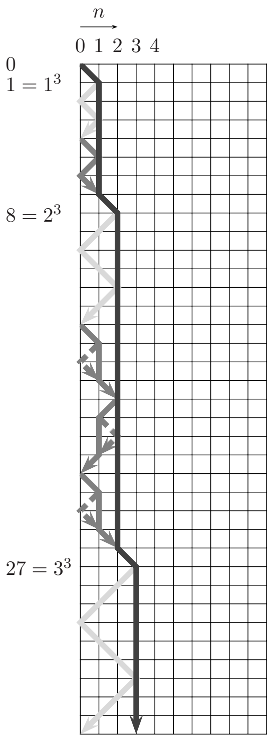


Fig. 6.21. Signal ξ with characteristic function n^3 (darkgray), auxiliary signals α (lightgray) and β (gray, gray dashed).

A gap in the family of practicable signals

Signals with characteristic functions of the form $n + \log_b(n)$ are lower bounds of CS-practicable signals beyond the identity (plus some constant). In between there is a gap.

Lemma 7. *Let $\rho(n) \geq n$, for all $n \geq 0$, be the characteristic function of a CS-practicable signal. Then $\rho(n) - n$ either is ultimately constant or there is some $b \geq 2$ such that $\rho(n) \geq n + \lfloor \log_b(n) \rfloor$, for all $n \geq 1$.*

Proof. Let $\mathcal{M}$ be a cellular space with state set S implementing the signal with characteristic function ρ. As usual, we denote its configurations by c_t, $t \geq 0$. We assume that $\rho(n) \geq n + \lfloor \log_b(n) \rfloor$ does *not* hold for all $b \geq 2$. In particular, it does not hold for $b = |S|$, where we may assume $|S| \geq 2$ without loss of generality. Therefore, there is an n_0 such that $\rho(n_0) < n_0 + \lfloor \log_b(n_0) \rfloor$. Since $\rho(n_0) \geq n_0$, we obtain $n_0 \geq b$.

Observe that due to the maximal speed of auxiliary signals, any cell $i \geq 0$ cannot participate in the implementation of the signal before time i. So, we consider the sequence of $m \geq 1$ successive states of some cell $i \geq 0$ beginning at time step i, that is, $c_i(i)c_{i+1}(i)\cdots c_{i+m-1}(i)$, and denote it by $w(i,m)$. The number of different sequences of length $\lfloor \log_b(n_0) \rfloor$ is at most n_0. Therefore, there are numbers $i \geq 0$ and $j \geq 1$ with $i+j \leq n_0$ such that $w(i, \lfloor \log_b(n_0) \rfloor) = w(i+j, \lfloor \log_b(n_0) \rfloor)$. This implies $w(\ell, \lfloor \log_b(n_0) \rfloor) = w(\ell + kj, \lfloor \log_b(n_0) \rfloor)$, for all $k \geq 0$ and $\ell \geq i$.

At time $\rho(n_0)$ the signal resides in cell n_0 which is indicated by a distinguished state. By $\rho(n_0) - n_0 < \lfloor \log_b(n_0) \rfloor$ follows that at time steps $\rho(n_0)+kj$ the cells n_0+kj are in the same state. Therefore, $\rho(n_0+kj) = \rho(n_0)+kj$ and due to the maximal speed of signals we obtain $\rho(n_0+k) = \rho(n_0)+k$, for all $k \geq 0$. We derive $\rho(n_0) - n_0 = \rho(n) - n$, for all $n \geq n_0$. Thus, ρ is ultimately constant. □

6.4.3 Time Constructibility

The investigation of time constructible functions in cellular spaces originates from [15], where a cellular space is constructed whose cell at the origin distinguishes exactly the time steps that are prime numbers. In [49] the systematic study of this concept was started. Since all values of a function have to be constructed, we consider strictly increasing functions. Initially, all cells except the one at the origin are quiescent.

Definition 7. *A strictly increasing function $f : \mathbb{N} \to \mathbb{N}$ is* CS-time-constructible *if there is a cellular space $\langle S, \delta, q_0, A, F \rangle$ with distinguished state $s \in S$ and initial configuration $c_0(0) = s$, $c_0(i) = q_0$, for $i \neq 0$, such that cell 0 is in some state from F at time t, if and only if $t = f(i)$ for some $i \geq 1$. The* family of CS-time-constructible functions *is denoted by $\mathscr{F}(CS)$.*

Lemma 8. *Let $b \geq 2$ be a positive integer. Then the function b^n is CS-time-constructible.*

Proof. In order to time construct the function b^n, an auxiliary signal β with speed $\frac{(b-1)}{(b+1)}$ is generated at time 0 in cell 0. It arrives at cells $\frac{kb(b-1)}{2}$ at time steps $\frac{kb(b+1)}{2}$. A second auxiliary signal α is generated at time b in cell 0. Subsequently, it repeatedly moves with speed 1 to the right until it meets β, bounces and moves with speed 1 back to cell 0. At its arrival cell 0 changes to some state from F.

If α leaves cell 0 at some time b^n, then it arrives at cell $\frac{1}{2}b^n(b-1)$ at time $b^n + \frac{1}{2}b^n(b-1)$. Exactly at this time signal β is in the same cell (for $k = b^{n-1}$). Therefore, signal α is back at cell 0 at time $b^n + b^n(b-1) = b^{n+1}$. □

At first glance, it seems that CS-time-constructible functions cannot grow faster than exponential functions. Among others, the next lemma says that this is a false impression.

Lemma 9. *1. The factorial function $n!$ is CS-time-constructible.*

2. The function that maps n to the nth prime number is CS-time-constructible.

The two families of CS-time-constructible functions and CS-practicable signals are very rich. Moreover, they are closely related. The next two results bridge the gap between the notions.

Theorem 2. *Let $h : \mathbb{N} \to \mathbb{N}$ be a strictly increasing function. If the signal with characteristic function h is CS-practicable, then h is CS-time-constructible.*

With other words, all characteristic functions of Section 6.4.2 are CS-time-constructible. Unfortunately, the converse is not true in general. For example, functions of the form $n + \lfloor \log^i \rfloor$, where $\log^i$ denotes the $i$fold iterated logarithm, $i \geq 2$, are CS-time-constructible. But by Lemma 7 they are not characteristic functions of CS-practicable signals. Nevertheless, for most of the relevant functions, the converse is true. Whenever the difference between $f(n)$ and n is at least linear, a corresponding signal can be derived from a CS-time-constructible function f.

Theorem 3. *Let f be a CS-time-constructible function. If $(k-1)f(n) \geq kn$, for some positive integer $k \geq 1$, then the signal with characteristic function f is CS-practicable.*

Finally, we summarize closure properties of the family $\mathscr{F}(\text{CS})$ in order to be able to construct new functions by certain operations.

Theorem 4. *Let f and g be functions belonging to $\mathscr{F}(CS)$.*

1. *Let k be a positive rational constant such that $\lfloor k \cdot f \rfloor$ is strictly increasing. Then $\lfloor k \cdot f \rfloor$ belongs to $\mathscr{F}(CS)$.*
2. *The sum $f + g$ belongs to $\mathscr{F}(CS)$.*
3. *If $f(n) \geq g(n)$, for all $n \geq 1$, and $(k+1)f - kg$ is strictly increasing, for some positive integer $k \geq 1$, then the function $(k+1)f - kg$ belongs to $\mathscr{F}(CS)$.*
4. *The composition $f(g)$ belongs to $\mathscr{F}(CS)$.*

Further results about signals as well as time constructible and time computable functions can be found, for example, in [5, 6, 7, 13, 34, 68, 69].

6.5 Cellular Language Acceptors

Now we turn to one of the main branches in the theory of automata. Clearly, the data supplied to some device can be arranged as strings of symbols. Instances of problems to solve can be encoded as strings with a finite number of different symbols. Furthermore, complex answers to problems can be encoded as binary sequences such that the answer is computed bit by bit. In order to compute one piece of the answer, the set of possible inputs is split into two sets associated with the binary outcome. From this point of view, the computational capabilities of the devices are studied in terms of string acceptance, that is, the determination to which of the two sets a given string belongs. These investigations are done with respect to and with the methods of language theory. For cellular spaces and automata they originated from [11, 12] and [61, 31]. Over the years substantial progress has been achieved, but there are still some basic open problems with deep relations to other fields.

6.5.1 Cellular Automata

Once we have a universal device there is a natural interest in realistic models that meet certain restrictions. Similar to the step from Turing machines to linear bounded automata, that is in terms of formal languages, from recursively enumerable to context-sensitive languages, the step from cellular spaces to cellular automata is to bound the number of available cells by the length of the input. For simplicity, the boundaries in space are modelled by a so-called permanent *boundary symbol* #. Due to the nearest neighbor connections, cells cannot communicate across a boundary. So, we may focus on the computations in between the boundaries and may disregard the computations outside. A widely studied question is to what extend one-way information flow reduces the computational capabilities of cellular automata. One-way information flow from right to left is achieved by providing the $\bar{H}_1$ neighborhood (cf. Example 2), that is, the next state of a cell depends on the current states of the cell itself and its immediate neighbor to the right.

Definition 8. *A* (one-dimensional) two-way cellular automaton (*CA*) *is a system* $\langle S, \delta, \#, A, F \rangle$*, where*

1. *S is the finite, nonempty set of* cell states*,*
2. *$\# \notin S$ is the permanent* boundary symbol*,*
3. *$A \subseteq S$ is the nonempty set of* input symbols*,*
4. *$F \subseteq S$ is the set of* final states*, and*
5. *$\delta : (S \cup \{\#\}) \times S \times (S \cup \{\#\}) \to S$ is the* local transition function*.*

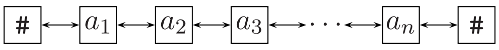


Fig. 6.22. A two-way cellular automaton.

If the flow of information is restricted to one-way, the resulting device is a *one-dimensional one-way cellular automaton* (OCA).

A *configuration* of a cellular automaton $\langle S, \delta, \#, A, F \rangle$ at time $t \geq 0$ is formally a mapping $c_t : \{1, \dots, n\} \to S$, for $n \geq 1$. The configuration at

$$\begin{tikzcd}
\boxed{a_1} & \boxed{a_2} \arrow[l] & \boxed{a_3} \arrow[l] & \cdots \arrow[l] & \boxed{a_n} \arrow[l] & \boxed{\#} \arrow[l]
\end{tikzcd}$$

Fig. 6.23. A one-way cellular automaton.

time 0 is defined by the given input $w = a_1 \cdots a_n \in A^+$. We set $c_0(i) = a_i$, for $1 \le i \le n$. So, $\#a_1a_2 \cdots a_n\#$ represents the initial configuration for w including the boundary symbols. Let c_t, $t \ge 0$, be a configuration with $n \ge 2$, then c_{t+1} is defined as follows:

$$c_{t+1} = \Delta(c_t) \iff \begin{cases} c_{t+1}(1) = \delta(\#, c_t(1), c_t(2)) \\ c_{t+1}(i) = \delta(c_t(i-1), c_t(i), c_t(i+1)), i \in \{2, \dots, n-1\} \\ c_{t+1}(n) = \delta(c_t(n-1), c_t(n), \#) \end{cases}$$

for CAs, and

$$c_{t+1} = \Delta(c_t) \iff \begin{cases} c_{t+1}(i) = \delta(c_t(i), c_t(i+1)), i \in \{1, \dots, n-1\} \\ c_{t+1}(n) = \delta(c_t(n), \#) \end{cases}$$

for OCAs. For $n = 1$, the next state of the sole cell is $\delta(\#, c_t(1), \#)$ or $\delta(c_t(1), \#)$.

6.5.2 Mode of Acceptance and Speed-Up

What is the result of the computation? One can partition the whole set of possible configurations into accepting and rejecting ones. This general approach is insufficient, since it could be much harder to determine whether a resulting configuration is accepting or not. So, it should be easy, say trivial, to recognize an accepting configuration. We define a configuration to be accepting when the cell receiving the first symbol of the input (cell 1) is in a final state from F. Further definitions of accepting configurations are studied, for example, in [26, 62], while more general input modes are considered in [42].

More precisely, an input w is accepted by an OCA, CA, or CS $\mathcal{M}$, if at some time during its course of computation cell 1 enters a final state. The *language accepted by* $\mathcal{M}$ is denoted by $L(\mathcal{M})$. Let $t : \mathbb{N} \to \mathbb{N}$, $t(n) \ge n$ be a mapping. If all $w \in L(\mathcal{M})$ are accepted within at most $t(|w|)$ time steps, then $L(\mathcal{M})$ is said to be of time complexity t. The family of languages that are accepted by OCAs (CAs, CSs) with time complexity t is denoted by $\mathscr{L}_t(\text{OCA})$ ($\mathscr{L}_t(\text{CA})$, $\mathscr{L}_t(\text{CS})$). The index is omitted for arbitrary time. Actually, arbitrary time in linearly space bounded devices is exponential time. If $t(n) = n$, acceptance is said to be in *real time* and we write $\mathscr{L}_{rt}(\text{OCA})$ ($\mathscr{L}_{rt}(\text{CA})$, $\mathscr{L}_{rt}(\text{CS})$). The *linear-time* languages $\mathscr{L}_{lt}(\text{OCA})$ are defined according to $\mathscr{L}_{lt}(\text{OCA}) = \bigcup_{k \in \mathbb{Q},\, k \ge 1} \mathscr{L}_{k \cdot n}(\text{OCA})$, and similarly for CAs and CSs.

In order to avoid technical overloading in writing, two languages L and L' are considered to be equal, if they differ at most in the empty word, that is, $L - \{\lambda\} = L' - \{\lambda\}$.

Example 6. The language $\{a^n b^n \mid n \geq 1\}$ is accepted by some OCA in real time (cf. Figure 6.24). During the first step, each cell with input symbol a changes into a state a'. In addition, the rightmost cell recognizes its position by means of the neighboring boundary symbol, and changes into a state r. Afterwards, at each time step the cell states b and r are shifted to the left. Whenever a b meets an a, the corresponding cell changes into state c. When r meets an a, the corresponding cell enters a final state R that is no longer shifted to the left. The construction is easily modified to reject inputs having a wrong format. □

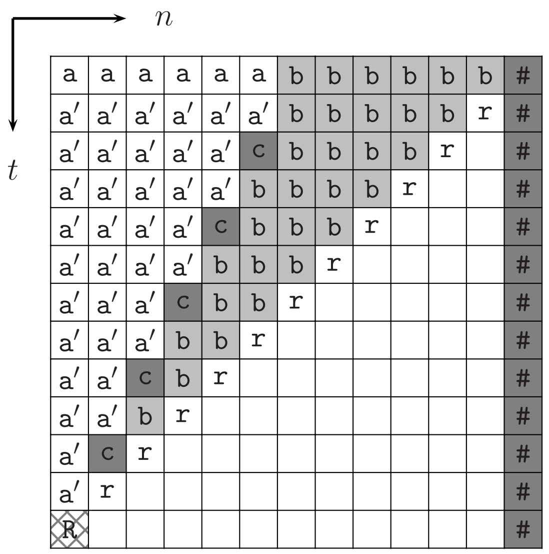


Fig. 6.24. Space-time diagram of an OCA accepting an input from the language $\{a^n b^n \mid n \geq 1\}$ in real time.

Helpful tools in connection with time complexities are speed-up theorems. Strong results are obtained in [24, 25], where the parallel language families are characterized by certain types of customized sequential machines. Among others, such machines have been developed for CSs, CAs, and OCAs. In particular, it is possible to speed up the time beyond real time linearly. Therefore, linear-time computations can be sped up close to real time. Later, the question whether real time can be achieved is discussed in detail later.

Theorem 5. *Let $\mathcal{M}$ be a CS, CA, or OCA obeying time complexity $rt + r(n)$, where $r : \mathbb{N} \to \mathbb{N}$ is a mapping and rt denotes real time. Then for all $k \geq 1$ an equivalent device $\mathcal{M}'$ of the same type obeying time complexity $rt + \lfloor \frac{r(n)}{k} \rfloor$ can effectively be constructed.*

The next example states that any constant beyond real time can be omitted.

Example 7. Let $k_0 \geq 1$ and $\mathcal{M}$ be a device in question with time complexity $rt + k_0$. Then there is an equivalent real-time device $\mathcal{M}'$ of the same type. It suffices to set $k = k_0 + 1$ and to apply Theorem 5 in order to obtain $rt + \lfloor \frac{k_0}{k} \rfloor = rt + \lfloor \frac{k_0}{k_0+1} \rfloor = rt$ for the time complexity of $\mathcal{M}'$. □

Next, a linear-time computation is sped up close to real time.

Example 8. Let $k_0 \geq 1$ and $\mathcal{M}$ be a device in question with time complexity $rt+k_0 \cdot rt$. Then for all rational numbers $\varepsilon > 0$ there is an equivalent device $\mathcal{M}'$ of the same type with time complexity $\lfloor (1+\varepsilon) \cdot rt \rfloor$. We set $k = \lceil \frac{k_0}{\varepsilon} \rceil$ and apply Theorem 5 in order to obtain $rt + \left\lfloor \frac{k_0 \cdot rt}{\lceil k_0/\varepsilon \rceil} \right\rfloor \leq rt + \left\lfloor \frac{k_0 \cdot rt}{k_0/\varepsilon} \right\rfloor = rt + \lfloor \varepsilon \cdot rt \rfloor = \lfloor (1+\varepsilon) \cdot rt \rfloor$. □

6.5.3 Basic Hierarchy of Languages

The goal of this section is to establish a basic hierarchy of cellular language families, and to compare the levels with well-known families of the Chomsky hierarchy. The properness of some inclusions are long-standing open problems with deep relations to sequential complexity problems. In order to establish the hierarchy we start at the upper end.

In Theorem 1 it is shown how to simulate deterministic Turing machines by cellular spaces. Since the number of non-quiescent cells is just one more than the space complexity of the Turing machine, CAs can simulate linearly space-bounded Turing machines. Conversely, a straightforward construction of Turing machines from CSs and of linearly space-bounded Turing machines from CAs shows the following lemma [31].

Lemma 10. *The family $\mathscr{L}(CS)$ is identical with the recursively enumerable languages. The family $\mathscr{L}(CA)$ is identical with the complexity class* DSPACE(n)*, that is, with the deterministic context-sensitive languages.*

Corollary 1. *The family $\mathscr{L}(CA)$ is properly included in $\mathscr{L}(CS)$.*

The family $\mathscr{L}$(OCA) is very powerful. It contains the context-free languages as well as a PSPACE-complete language [8, 22]. For structural reasons it is contained in $\mathscr{L}$(CA). It is an open problem whether or not the inclusion is proper.

Corollary 2. *The family $\mathscr{L}(OCA)$ is included in $\mathscr{L}(CA)$.*

We continue with the lower end of the hierarchy, and consider the weakest devices in question, the real-time OCAs.

Lemma 11. *The regular languages are properly included in $\mathscr{L}_{rt}(OCA)$.*

Proof. Let L be a regular language represented by some deterministic finite automaton $\mathcal{E}$. We construct a real-time OCA $\mathcal{M}$ with two tracks that simulates $\mathcal{E}$. In fact, the first register of each cell is used to simulate $\mathcal{E}$, whereas the second track is used to shift the input to the left, that is, to feed it into the simulation of $\mathcal{E}$. So, the first register of the leftmost cell fetches the whole input and simulates $\mathcal{E}$ completely.

The properness of the stated inclusion follows from Example 6 which shows that the non-regular language $\{a^n b^n \mid n \geq 1\}$ belongs to $\mathscr{L}_{rt}(\text{OCA})$. □

In order to reach the next level of the hierarchy we consider unary languages. It turns out that even massively parallel OCAs with a certain time bound cannot accept more unary languages than a single deterministic finite automaton [59].

Lemma 12. *Let $L \subseteq \{a\}^+$ be a unary language accepted by some OCA $\mathcal{M}$. If for all $b \geq 2$ there is a $w_b \in L$ which is accepted by $\mathcal{M}$ in $t(|w_b|) < |w_b| + \lfloor \log_b(|w_b|) \rfloor$ time steps, then there are $k_0, k \geq 1$ such that $a^{k_0+m\cdot k} \in L$ for all $m \geq 0$.*

Proof. Let $\mathcal{M} = \langle S, \delta, \texttt{\#}, A, F \rangle$. In particular, for $b = (|S|+1)^3$ there exists a $w_b \in L$ whose length is denoted by n_0, and which is accepted in $t(n_0)$, $n_0 \leq t(n_0) < n_0 + \lfloor \log_{(|S|+1)^3}(n_0) \rfloor$, time steps. It follows $\lfloor \log_{(|S|+1)^3}(n_0) \rfloor \geq 1$, and thus $n_0 > |S|^3$. Moreover, we have $\lfloor n_0^{\frac{1}{2}} \rfloor > |S|$, for $|S| > 1$.

For convenience now we assume that the cells of the OCA are numbered from right to left. For a computation with initial configuration $\texttt{\#}a^{n_0}\texttt{\#}$ we consider the words $c_{n-1}(n)c_n(n)c_{n+1}(n)\cdots c_{n+\lfloor \log_{|S|^2}(n_0)\rfloor -1}(n)$, for all $1 \leq n \leq n_0$, and denote them by e_n. All these words have the same length $\lfloor \log_{|S|^2}(n_0) \rfloor + 1$. The number of different words is at most

$$\begin{aligned}
|S|^{\lfloor \log_{|S|^2}(n_0)\rfloor+1} &= |S| \cdot |S|^{\lfloor \log_{|S|^2}(n_0)\rfloor} \leq |S| \cdot \lfloor |S|^{\log_{|S|^2}(n_0)} \rfloor \\
&= |S| \cdot \lfloor |S|^{\frac{1}{2}\log_{|S|}(n_0)} \rfloor = |S| \cdot \lfloor (|S|^{\log_{|S|}(n_0)})^{\frac{1}{2}} \rfloor \\
&= |S| \cdot \lfloor n_0^{\frac{1}{2}} \rfloor < \lfloor n_0^{\frac{1}{2}} \rfloor \cdot \lfloor n_0^{\frac{1}{2}} \rfloor \leq n_0.
\end{aligned}$$

Therefore, at least two of $e_1, \ldots, e_{n_0}$ are identical, say e_i and e_j with $i < j$. Since initially all cells are in the same state, e_{n+1} is uniquely determined by e_n. So, $e_i = e_j$ implies $e_{n_0-(j-i)} = e_{n_0}$ and, furthermore, if the array is long enough, $e_{n_0+m(j-i)} = e_{n_0}$, for all $m \geq -1$. For $k_0 = n_0$ and $k = j - i$, $e_{k_0+m\cdot k} = e_{n_0}$ follows, for all $m \geq -1$. Since a^{n_0} is accepted in less than $n_0 + \lfloor \log_{(|S|+1)^3}(n_0) \rfloor$ time steps, word e_{n_0} contains an accepting state due to $\lfloor \log_{(|S|+1)^3}(n_0) \rfloor \leq \lfloor \log_{|S|^2}(n_0) \rfloor + 1$. Therefore, for all $m \geq 1$, input $a^{k_0+m\cdot k}$ is also accepted. □

For real-time computations a closer look at the proof of the previous lemma reveals the following lemma.

Lemma 13. *Each unary real-time OCA language is regular.*

Proof. Considering the proof of Lemma 12 in case of real time, one observes that the relevant information of the words e_n consists of the first two states only. Moreover, the first state appears in all cells to the left at the same time step. So, it is easy to construct an equivalent deterministic finite automaton with two registers that computes the first state of the next word e_{n+1} by applying the transition function to twice the current first state, and the second state of the next word e_{n+1} by applying the transition function to the current first state and the current second state. □

Example 9. In general, Lemma 12 cannot be used to prove that an accepted unary language is regular. For example, consider the non-regular language $L = \{a^{2^n} \mid n \geq 1\} \cup \{a^{2n-1} \mid n \geq 1\}$, and suppose there is an OCA accepting $\{a^{2^n} \mid n \geq 1\}$ with time complexity $t(n)$ that is at least of order $n + \log(n)$ (cf. Example 10). Clearly, the second subset $\{a^{2n-1} \mid n \geq 1\}$ which contains all words of odd length can be accepted in real time. So, an OCA accepting L by accepting the subsets on different tracks in parallel obeys the time complexity $t(n)$ if n is even, and real time if n is odd. Therefore, the conditions of Lemma 12 are met, and it is applicable for $k_0 = 1$ and $k = 2$. □

On the other hand, in particular cases Lemma 12 *can* be used to prove that a non-regular unary language is not accepted in less than $n + \log(n)$ time.

Theorem 6. *Let $r \in o(\log)$, $r : \mathbb{N} \to \mathbb{N}$, be a function. Then language $L = \{a^{2^n} \mid n \geq 1\}$ does not belong to $\mathscr{L}_{rt+r}(OCA)$.*

Proof. In contrast to the assertion, assume $L \in \mathscr{L}_{rt+r}(\text{OCA})$. Then, for all $b \geq 1$, there is a $w_b \in L$ which is accepted in $t(|w_b|) < |w_b| + \lfloor \log_b(|w_b|) \rfloor$ time steps. By Lemma 12 we conclude that there are $n_0, k \geq 1$, such that $a^{2^{n_0}} \in L$ and $a^{2^{n_0}+m\cdot k} \in L$, for all $m \geq 1$, which is a contradiction. □

The next example gives a tight bound for the OCA time complexity necessary to accept language $\{a^{2^n} \mid n \geq 1\}$.

Example 10. The following OCA $\mathcal{M} = \langle S, \delta, \#, A, F \rangle$ accepts the unary language $\{a^{2^n} \mid n \geq 1\}$ with time complexity $t(n) = n + \log(n)$.

The basic idea of the construction is to generate a binary counter in the rightmost cell with one step delay (cf. Figure 6.25). The counter moves to the left whereby the cells passed through are counted. The length of the counter is increased when necessary. In addition, cells which are passed through by the counter have to check whether all bits are 1. In this case the value of the counter is $2^n - 1$, for some $n \geq 1$. Due to the delayed generation this indicates a correct input length and the cell enters the final state. Clearly, the desired time complexity is obeyed. A formal construction is as follows.

$S = \{a, e, 1, +, 0, \overset{\bullet}{\scriptstyle 0}, \overset{+}{\scriptstyle 1}\}$, $A = \{a\}$, $F = \{+\}$, and for all $s_1, s_2 \in S$:

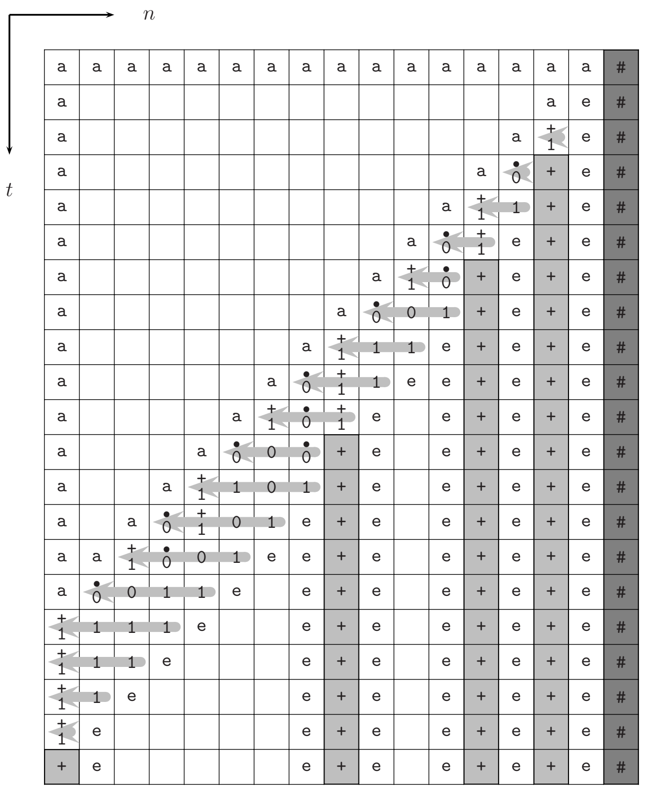


Fig. 6.25. Space-time diagram of an OCA accepting an input from the language $\{a^{2^n} \mid n \geq 1\}$ in $n + \log(n)$ time. Lightgray arrows mark the moving counter, whose digits are 0, 1, or $\overset{\bullet}{0}$. The latter is a 0 reporting a carry-over. A $\overset{+}{1}$ indicates that, so far, the cell has been passed through by 1s only.

$$\delta(s_1, s_2) = \begin{cases} e \text{ if } \big(s_1 \notin \{\overset{\bullet}{0}, \overset{+}{1}, +, a\} \wedge s_2 \in \{e, +\}\big) \vee \big(s_1 = a \wedge s_2 = \#\big) \\ + \text{ if } \big(s_1 = \overset{+}{1} \wedge s_2 = e\big) \\ \overset{+}{1} \text{ if } \big(s_1 = a \wedge s_2 \in \{e, \overset{\bullet}{0}\}\big) \vee \big(s_1 = \overset{+}{1} \wedge s_2 = 1\big) \\ \overset{\bullet}{0} \text{ if } \big(s_1 = a \wedge s_2 = \overset{+}{1}\big) \vee \big(s_1 = \overset{\bullet}{0} \wedge s_2 \in \{\overset{+}{1}, 1\}\big) \\ 0 \text{ if } \big(s_1 \neq a \wedge s_2 \in \{\overset{\bullet}{0}, 0\}\big) \\ 1 \text{ if } \big(s_1 = \overset{\bullet}{0} \wedge s_2 \in \{0, e, +\}\big) \vee \big(s_1 \neq \overset{+}{1} \wedge s_2 = 1\big) \\ s_1 \text{ otherwise} \end{cases}$$

□

Corollary 3. *The family $\mathscr{L}_{rt}(OCA)$ is properly included in $\mathscr{L}_{rt+\log}(OCA)$.*

For structural reasons, the next inclusion follows immediately. Its properness and, in fact, infinite proper hierarchies in between $\mathscr{L}_{rt}(\text{OCA})$ and $\mathscr{L}_{lt}(\text{OCA})$ have been shown in [37].

Corollary 4. *The family $\mathscr{L}_{rt+\log}(OCA)$ is properly included in $\mathscr{L}_{lt}(OCA)$.*

Since real-time and linear-time CSs use at most linearly many cells, they can be simulated by real-time and linear-time CAs. So, we do not need to consider them separately. Once we know that, in general, a linear-time OCA language cannot be accepted by any real-time OCA, the question arises whether two-way information flow can help in this respect. The next result gives a (partial) answer [9, 71]. The answer is not complete, since the input has to be reversed. Alternatively, one could reverse the neighborhood of the cells in an OCA. Then the rightmost cell indicates the result of the computation. In this case the input could remain as it is. In any case, the condition cannot be relaxed since it is an open problem whether the corresponding language families are closed under reversal.

Theorem 7. *A language is accepted by a linear-time OCA if and only if its reversal is accepted by a CA in real time.*

Proof. Let $\mathcal{M}$ be a real-time CA. The cells of a linear-time OCA $\mathcal{M}'$ accepting $L^R(\mathcal{M})$ collect the information necessary to simulate one transition of $\mathcal{M}$ in an intermediate step. Therefore, the first step of $\mathcal{M}$ is simulated in the second step of $\mathcal{M}'$. We obtain a behavior as depicted in Figure 6.26.

Altogether, $\mathcal{M}'$ cannot simulate the last step of $\mathcal{M}$. So, the construction has to be extended slightly. Each cell has an extra register that is used to simulate transitions of $\mathcal{M}$ under the assumption that the cell is the leftmost one (cf. Figure 6.27). The transitions of the real leftmost cell now correspond to the missing transitions of the previous simulation. $\square$

It turned out that for OCAs linear time is strictly more powerful than real time. The problem is still open for CAs. The next inclusions follow for structural reasons and by the closure of $\mathscr{L}_{lt}(\text{CA})$ under reversal.

Corollary 5. *Any linear-time OCA language as well as its reversal belong to $\mathscr{L}_{lt}(CA)$.*

Now we can join the upper and the lower part of the hierarchy. The question whether or not one-way information flow is a strict weakening of two-way information flow for unbounded time is a long-standing open problem. Even the inclusion does not follow for structural reasons. It is proved in [8, 22] in terms of simulations of equivalent sequential machines. In the same paper it is shown that a PSPACE-complete language is accepted by OCAs. In fact, it is an open question whether real-time CAs are strictly weaker than unbounded time CAs. If both classes coincide, then a PSPACE-complete language would be accepted in polynomial time! The basic hierarchy obtained is depicted in Figure 6.31 on page 221.

Theorem 8. *The family $\mathscr{L}_{lt}(CA)$ is included in $\mathscr{L}(OCA)$.*

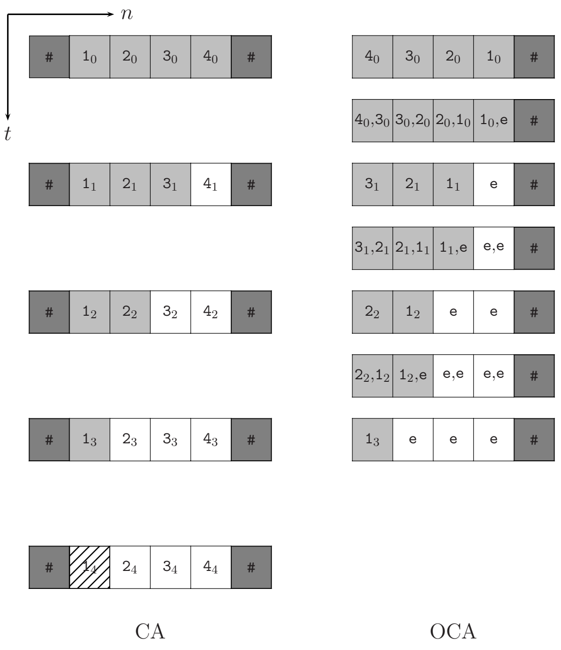


Fig. 6.26. Intermediate steps in the construction of the proof of Theorem 7.

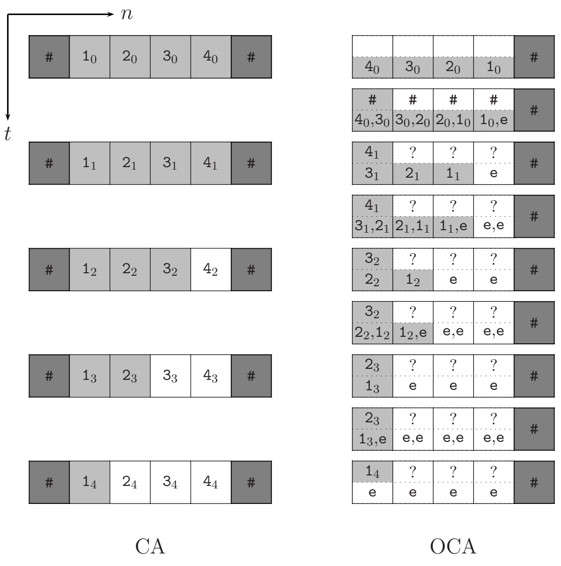


Fig. 6.27. Example of a linear-time OCA simulation of a real-time CA computation on reversed input.

6.5.4 Relations to Context-Free Languages

The relations between the language families in question and the regular, (deterministic) context-sensitive and recursively enumerable languages of the Chomsky hierarchy are quite clear. But what about the context-free languages? In [8] it is shown that they are properly included in the family $\mathscr{L}$(OCA). On the other hand, the family $\mathscr{L}_{rt}$(OCA) is incomparable with the family of context-free languages [65] since it contains, for example, the language $\{a^n b^n c^n \mid n \geq 1\}$, and does not contain the two-linear language LL with

$$L = \{a^n b^n \mid n \geq 1\} \cup \{a^n bwab^n \mid w \in \{a,b\}^*, n \geq 1\}.$$

Theorem 9.

1. *The context-free languages are properly included in the family $\mathscr{L}(OCA)$.*
2. *The family of context-free languages is incomparable with the families $\mathscr{L}_{rt}(OCA)$ and $\mathscr{L}_{rt+\log}(OCA)$.*

Nevertheless, even the real-time OCA languages contain important subfamilies, for example, the linear context-free languages [61], the Dyck languages [59], and the bracketed context-free languages [14]. Furthermore, the non-semilinear language $\{(a^i b)^* \mid i \geq 0\}$ [59] and the inherently ambiguous language $\{a^i b^j c^k \mid i = j \text{ or } j = k \text{ for } i,j,k \geq 1\}$ [31] belong to $\mathscr{L}_{rt}$(OCA).

Whether or not the context-free languages are included in the family $\mathscr{L}_{rt}$(CA) is an open question raised in [31]. It is related to the open question whether or not sequential one-tape Turing machines are able to accept the context-free languages in square-time. A proof for the inclusion would imply the existence of square-time Turing machines. In fact, also the problem whether or not the context-free languages are included in $\mathscr{L}_{lt}$(CA) is open. But for the important metalinear and deterministic context-free languages we can answer the inclusion problem in the affirmative [40].

Theorem 10. *The metalinear languages are properly included in the family $\mathscr{L}_{rt}(CA)$.*

Proof. Let L be a metalinear language. Then there exists a $k \geq 1$ such that L is k-linear. Therefore, we can represent L as union of finitely many concatenations $L_1 \cdot L_2 \cdot \cdots \cdot L_k$, where each L_i is a linear context-free language. The family $\mathscr{L}_{rt}$(CA) is closed under union. The family $\mathscr{L}_{rt}$(OCA) is closed under reversal [59]. Since the linear context-free languages [61] belong to the family $\mathscr{L}_{rt}$(OCA), there exist real-time OCAs for each of the languages $L_1^R, \ldots, L_k^R$. Since the concatenation of a real-time and a linear-time OCA language is again a linear-time OCA language [22], we obtain $L_k^R \cdot \cdots \cdot L_1^R \in \mathscr{L}_{lt}$(OCA). From the equality $\mathscr{L}_{lt}(\text{OCA}) = \mathscr{L}_{rt}^R(\text{CA})$ it follows $L_1 \cdot \cdots \cdot L_k = L \in \mathscr{L}_{rt}$(CA). □

Theorem 11. *The deterministic context-free languages are properly included in the family $\mathscr{L}_{rt}(CA)$.*

Proof. Here we cannot use an ordinary stack simulation because we are concerned with deterministic pushdown automata that are allowed to perform λ-transitions. But without loss of generalization we may assume that a given deterministic pushdown automaton $\mathcal{M}$ pushes at most $k \geq 1$ symbols onto the stack in every non-λ-transition, and erases exactly one symbol from the stack in every λ-transition [17]. Moreover, the first transition is a non-λ-transition.

By the equality $\mathscr{L}_{lt}(\text{OCA}) = \mathscr{L}_{rt}^{R}(\text{CA})$ it suffices to construct a $((k+1)\cdot n)$-time OCA $\mathcal{M}'$ that accepts the language $L^R(\mathcal{M})$. To this end, let $\mathcal{M}$ be a deterministic pushdown automaton with state set S, set of stack symbols G, set of input symbols A, initial state s_0, bottom-of-stack symbol $\perp \in G$, set of accepting states F, and transition function $\delta : S \times G \times (A \cup \{\lambda\}) \to S \times G^*$.

Now we construct the OCA $\mathcal{M}' = \langle S', \delta', \#, A, F' \rangle$.

Each cell of $\mathcal{M}$ has $k+2$ registers, where the first one can store either an input symbol, or a distinguished special symbol $\$$, or a state of $\mathcal{M}$. The second register is used to implement a finite counter with range 0 to k. The remaining k registers can store stack symbols of $\mathcal{M}$ and may be empty. Accordingly, S' is defined to be $(S \cup A \cup \{\$\}) \times \{0, \dots, k\} \times (G \cup \{\lambda\})^k$. The transition function δ' ensures that at every time step $t \geq 1$ exactly one cell contains a symbol from S in its first register. This symbol is the current state of $\mathcal{M}$. So, F' is defined to be $F \times \{0, \dots, k\} \times (G \cup \{\lambda\})^k$.

Let $a_1 a_2 \cdots a_n$ be an input of $\mathcal{M}$. We consider $\mathcal{M}'$ when fed with the reverse input $a_n a_{n-1} \cdots a_1$. Initially all counter registers are set to 0, and all k registers for stack symbols are empty. Since the first transition of $\mathcal{M}'$ is a non-λ-transition and the rightmost cell can identify itself, for all $a \in A$, the initial step of $\mathcal{M}'$ is defined as follows (cf. Figure 6.28).

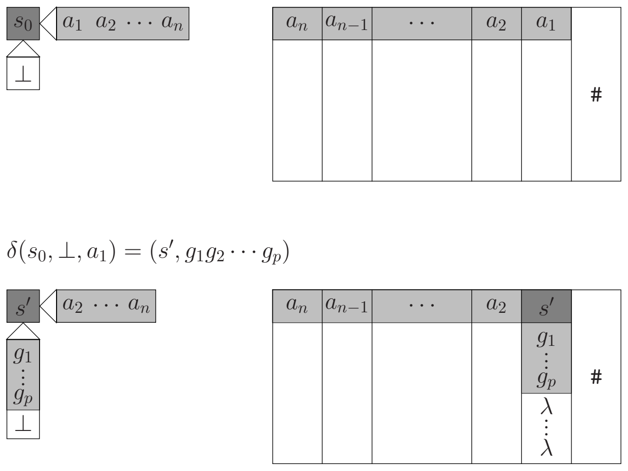


Fig. 6.28. The initial step of the pushdown automaton $\mathcal{M}$ (left) and the corresponding transition of the OCA $\mathcal{M}'$ (right). Counters are not depicted.

$$\delta(s_0,\perp,a)=(s',g_1g_2\cdots g_p) \iff \delta'((a,0,\lambda^k),\#)=(s',0,g_1g_2\cdots g_p\lambda^{k-p})$$

Proceeding inductively, at every time step there is exactly one distinguished cell containing the current state of $\mathcal{M}$ in its first register. All cells to its right are marked by the special symbol \$, and all cells to its left store still their input symbols (cf. Figures 6.28,6.29,6.30).

Every simulation of a transition of $\mathcal{M}$ is performed in two phases. During the first phase, the new state and the new symbols at the top of the stack of $\mathcal{M}$ are computed. (The first phase is indicated by 0 in the counter registers.) Let $\mathcal{M}$ perform a non-λ-transition (cf. Figure 6.29). The cell to the left of the distinguished cell has the necessary information. For all $a\in A$, $s\in S$, and $g_j\in G$,

$$\begin{aligned}&\delta(s,g_1,a)=(s',g'_1g'_2\cdots g'_p) \iff\\ &\delta'((a,0,\lambda^k),(s,0,g_1g_2\cdots g_k))=(s',k-p,g'_1g'_2\cdots g'_p\lambda^{k-p}).\end{aligned}$$

All other cells of the left part keep their states. For all $a,\tilde{a}\in A$,

$$\delta'((a,0,\lambda^k),(\tilde{a},0,\lambda^k))=(a,0,\lambda^k).$$

The distinguished cell observes that $\mathcal{M}$ does not perform a λ-transition. It stores the special symbol \$ in its first register. For all $s\in S$,

$$\begin{aligned}&\delta(s,g_1,a)\text{ is defined for some }a\in A \iff\\ &\delta'((s,0,g_1g_2\cdots g_k),\#)=(\$,0,g_2g_3\cdots g_k\lambda)\text{ and}\\ &\delta'((s,0,g_1g_2\cdots g_k),(\$,0,g_{k+1}g_{k+2}\cdots g_{2k}))=(\$,0,g_2g_3\cdots g_kg_{k+1}).\end{aligned}$$

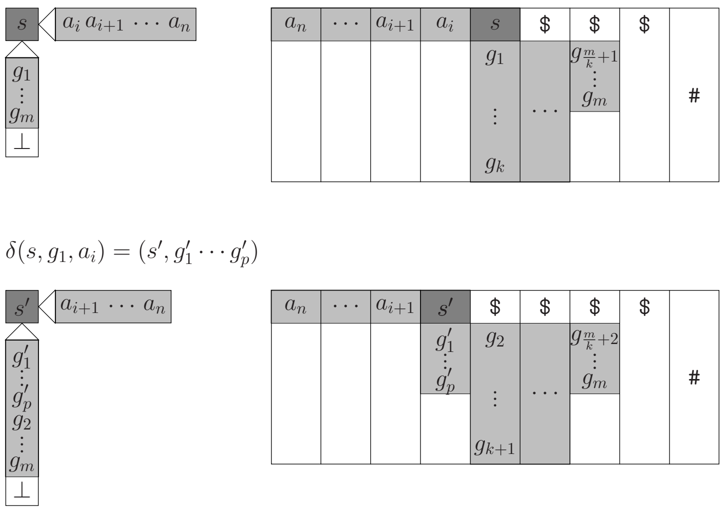


Fig. 6.29. A non-λ-transition of the pushdown automaton $\mathcal{M}$ (left) and the corresponding transition of the OCA $\mathcal{M}'$ (right). Counters are not depicted.

At each time step, cells containing the special symbol \$ shift the contents of the k stack symbol registers one position to the top where the last register is filled with the symbol shifted out by the right neighbor. For all $g_j \in G$,

$$\delta'((\$, 0, g_1 g_2 \cdots g_k), \#) = (\$, 0, g_2 g_3 \cdots g_k \lambda) \text{ and}$$
$$\delta'((\$, 0, g_1 g_2 \cdots g_k), (\$, 0, g_{k+1} g_{k+2} \cdots g_{2k})) = (\$, 0, g_2 g_3 \cdots g_k g_{k+1})$$

The purpose of the counter is to pack the stack symbols after a non-λ-transition. If the content of the counter is greater than 0, the second phase is performed in the distinguished cell. For all $s \in S$, $g_j \in G$, and $1 \leq i \leq k$,

$$\delta'((s, i, g_1 g_2 \cdots g_p \lambda^i), (\$, 0, g_{p+1} g_{p+2} \cdots g_{p+k})) = (s, i-1, g_1 g_2 \cdots g_p g_{p+1} \lambda^{i-1}).$$

If the counter has been decreased to 0, then the next transition of $\mathcal{M}$ is simulated. The distinguished cell as well as its left neighbor recognize whether it is a λ-transition. Since during λ-transitions the top-of-stack symbol is erased, from the above described behavior we get the packing for free (cf. Figure 6.30). For all $a \in A$, $s \in S$, and $g_j \in G$,

$$\delta(s, g_1, \lambda) \text{ is defined or } i > 0 \iff$$
$$\delta'((a, 0, \lambda), (s, i, g_1 g_2 \cdots g_k)) = (a, 0, \lambda)$$

$$\delta(s, g_1, \lambda) = (s', \lambda) \iff$$
$$\delta'((s, 0, g_1 g_2 \cdots g_k), \#) = (s', 0, g_2 g_3 \cdots g_k \lambda) \text{ and}$$
$$\delta'((s, 0, g_1 g_2 \cdots g_k), (\$, 0, g_{k+1} g_{k+2} \cdots g_{2k})) = (s', 0, g_2 g_3 \cdots g_k g_{k+1})$$

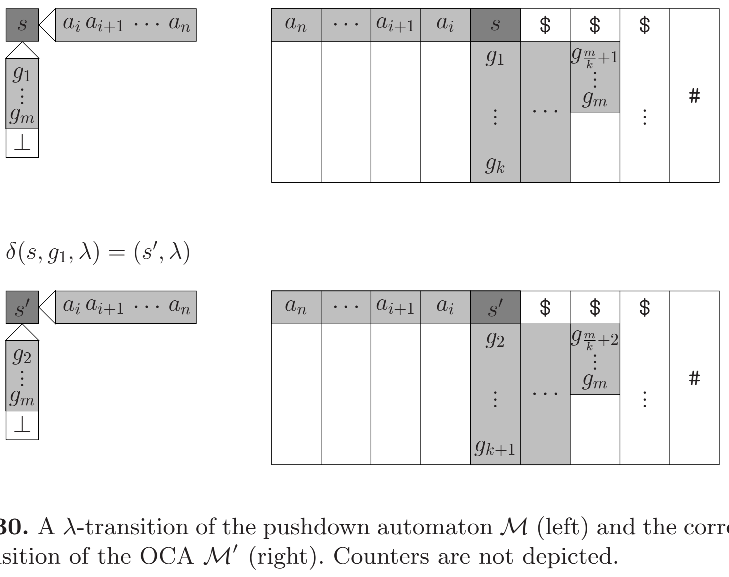


Fig. 6.30. A λ-transition of the pushdown automaton $\mathcal{M}$ (left) and the corresponding transition of the OCA $\mathcal{M}'$ (right). Counters are not depicted.

The OCA $\mathcal{M}'$ takes at most $(k+1)\cdot n$ time steps. It has to simulate n non-λ-transitions of $\mathcal{M}$. This takes n time steps. During each of these transitions some p symbols are pushed onto the stack which cause $k-p$ packing steps. In addition, there are at most p additional λ-transitions that erase the p symbols. So, every non-λ-transition causes at most k further steps. It follows that $\mathcal{M}'$ obeys the time complexity $(k+1)\cdot n$. $\square$

Altogether we obtain the hierarchy depicted in Figure 6.31, where the only known proper inclusions are at the top and the lower end.

$$\begin{tikzcd}
 & \mathscr{L}(\mathrm{CS}) \arrow[r, equal] & \mathrm{RE} & \\
 & \mathscr{L}(\mathrm{CA}) \arrow[u] \arrow[r, equal] & \mathrm{DCSL} \arrow[u] & \\
 & \mathscr{L}(\mathrm{OCA}) \arrow[u, dashed] & \mathrm{CFL} \arrow[l] \arrow[u] & \\
 & \mathscr{L}_{lt}(\mathrm{CA}) \arrow[u, dashed] & & \\
\mathscr{L}_{lt}(\mathrm{OCA}) \arrow[ur, dashed] & \mathscr{L}_{rt}(\mathrm{CA}) = \mathscr{L}_{lt}^{R}(\mathrm{OCA}) \arrow[u, dashed] & & \mathrm{DCFL} \arrow[ll] \arrow[uul] \\
\mathscr{L}_{rt+\log}(\mathrm{OCA}) \arrow[u] & & \mathrm{METALIN} \arrow[uuu] \arrow[ul] \arrow[ull] & \\
 & \mathscr{L}_{rt}(\mathrm{OCA}) \arrow[ul] \arrow[uu] & \mathrm{LIN} \arrow[l] \arrow[u] & \mathrm{REG} \arrow[l] \arrow[uu]
\end{tikzcd}$$

Fig. 6.31. Basic hierarchy of language families. A solid arrow indicates a proper inclusion, a dashed arrow an inclusion, and a double arrow an equality. Linear, meta-linear, and deterministic context-free languages are denoted by LIN, METALIN, and DCFL. Regular, context-free, deterministic context-sensitive, and recursively enumerable languages are denoted by REG, CFL, DCSL, and RE.

6.5.5 Summary of Closure Properties and Decidability Problems

Finally, this subsection is devoted to summarize closure properties of and decidability results for the language families in question.

Closure properties

The closure properties of $\mathscr{L}(\mathrm{CS})$ and $\mathscr{L}(\mathrm{CA})$ are those of the recursively enumerable and deterministic context-sensitive languages. In [8, 22] strong closure properties are derived for the family of OCA languages. It is shown that

$\mathscr{L}$(OCA) is an AFL, that is, an abstract family of languages (cf., e.g., [58]) which is in addition closed under reversal.

The closure under reversal is of crucial importance. It is an open problem for $\mathscr{L}_{rt}$(CA) and, equivalently, for $\mathscr{L}_{lt}$(OCA). Moreover, it is linked with the open closure property under concatenation for the same family. If the answer to the open reversal closure of $\mathscr{L}_{rt}$(CA) is negative, we have to deal with two different language families. Since the properness of the inclusion $\mathscr{L}_{rt}(\text{CA}) \subseteq \mathscr{L}_{lt}(\text{CA})$ is also open, the problem gains in importance. A negative answer of the former problem would imply a proper inclusion. A language $L \in \mathscr{L}_{rt}(\text{CA})$ whose reversal does not belong to $\mathscr{L}_{rt}$(CA) may serve as witness since $\mathscr{L}_{lt}$(CA) is closed under reversal. In fact, the following stronger relation is shown in [23].

Theorem 12. *The family $\mathscr{L}_{rt}(CA)$ is closed under reversal if and only if $\mathscr{L}_{rt}(CA)$ and $\mathscr{L}_{lt}(CA)$ are identical.*

The question whether or not the family $\mathscr{L}_{rt}$(OCA) is closed under concatenation was open for a long time. It has been solved negatively in [64].

The question whether or not one of the families $\mathscr{L}_{rt}(\text{CA}) = \mathscr{L}_{lt}^{R}(\text{OCA})$ or $\mathscr{L}_{lt}$(CA) is closed under concatenation is another famous open problem in this field. Nevertheless, it is shown in [23] that the closure of $\mathscr{L}_{rt}$(CA) under reversal implies its closure under concatenation. Since in this case we obtain $\mathscr{L}_{rt}(\text{CA}) = \mathscr{L}_{lt}(\text{CA})$, the family of linear-time CA languages were also closed under concatenation. The concatenation closure for *unary* real-time CA languages has been solved in the affirmative [23].

Table 6.1 summarizes some closure properties of the language families in question.

Decidability problems

It is well known that all nontrivial decidability problems for Turing machines are undecidable [55]. Moreover, many of them are not even semidecidable, for example, neither finiteness nor infiniteness. Now we turn to summarize undecidable properties of cellular automata. Most of the early results are shown in [59] by reductions of Post Correspondence Problems. In terms of trellis automata the undecidability of emptiness, equivalence, and universality is derived in [28]. Here we present improved results that show the *non-semidecidability* of the properties. Almost all results in this section are proved in [43] by reductions of Turing machine problems. To this end, valid computations of Turing machines are considered. Roughly speaking, these are histories of accepting Turing machine computations which can be encoded in small grammars [20]. The generated languages are accepted by real-time OCAs.

Theorem 13. *For any language family that effectively contains $\mathscr{L}_{rt}(OCA)$ emptiness, universality, finiteness, infiniteness, equivalence, inclusion, context-freeness, and regularity are not semidecidable.*

	$\mathscr{L}_{rt}$(OCA)	$\mathscr{L}_{rt}$(CA)	$\mathscr{L}_{lt}$(CA)	$\mathscr{L}$(OCA)	$\mathscr{L}$(CA)	$\mathscr{L}$(CS)
$\cup$, $\cap$	+	+	+	+	+	+
complementation, −	+	+	+	+	+	−
reversal	+	?	+	+	+	+
concatenation	−	?	?	+	+	+
λ-free iteration	−	?	?	+	+	+
concatenation REG	+	+	?	+	+	+
REG concatenation	+	?	?	+	+	+
marked concatenation	+	+	+	+	+	+
marked λ-free iteration	+	+	+	+	+	+
hom^{-1}	+	+	+	+	+	+
deterministic gsm^{-1}	+	+	+	+	+	+
gsm^{-1}	−	?	?	+	+	+
inj. length-pres. *hom*	+	+	+	+	+	+
λ-free *hom*	−	?	?	+	+	+
λ-free *gsm*	−	?	?	+	+	+
λ-free substitution	−	?	?	+	+	+
hom	−	−	−	−	−	+

Table 6.1. Summary of closure properties. Concatenation REG denotes the concatenation with regular languages at the right, REG concatenation at the left, *hom* denotes homomorphisms, *gsm* generalized sequential machine mappings, and inj. length-pres. abbreviates injective length-preserving. A + indicates closure, a − non-closure, and a question mark an open problem.

Next the question arises whether some structural properties of cellular language acceptors are (semi)decidable. For example, whether or not a real-time two-way language is a real-time one-way language. The questions turned out to be not even semidecidable.

Theorem 14. *For any language family $\mathscr{L}$ that effectively contains $\mathscr{L}_{rt}(CA)$ it is not semidecidable whether $L \in \mathscr{L}$ is a real-time OCA language.*

In general, a family $\mathscr{L}$ of languages possesses a pumping lemma in the narrow sense if for each $L \in \mathscr{L}$ there exists a constant $n \geq 1$ computable from L such that each $z \in L$ with $|z| > n$ admits a factorization $z = uvw$, where $|v| \geq 1$ and $u'v^iw' \in L$, for infinitely many $i \geq 0$. The prefix u' and the suffix w' depend on u, w and i.

Theorem 15. *Any language family whose word problem is semidecidable and that effectively contains $\mathscr{L}_{rt}(OCA)$ does not possess a pumping lemma (in the narrow sense).*

Theorem 16. *There is no minimization algorithm converting some CA or OCA (with arbitrary time complexity) to an equivalent automaton of the same type with a minimal number of states.*

Nevertheless, there are nontrivial decidable properties of cellular spaces. It is known that injectivity of the global transition function is equivalent to the reversibility of the automaton. It is shown in [2] that global reversibility is decidable for one-dimensional CSs, whereas the problem is undecidable for higher dimensions [36].

References

1. K. Albert, Čulik II. A simple universal cellular automaton and its one-way and totalistic version. *Complex Systems*, 1:1–16, 1987.
2. S. Amoroso and Y.N Patt. Decision procedures for surjectivity and injectivity of parallel maps for tessellation structures. *J. Comput. System Sci.*, 6:448–464, 1972.
3. R.M. Balzer. An 8-state minimal time solution to the firing squad synchronization problem. *Inform. Control*, 10:22–42, 1967.
4. E.R. Berlekamp, J.H. Conway, and R.K. Guy. Winning Ways for your Mathematical Plays. volume 2, chapter 25, Academic Press, 1982.
5. Th. Buchholz and M. Kutrib. On the power of one-way bounded cellular time computers. In *In Developments in Language Theory (DLT 1997)*, pp. 365-375, 1997.
6. Th. Buchholz and M. Kutrib. Some relations between massively parallel arrays. *Parallel Comput.*, 23:1643–1662, 1997.
7. Th. Buchholz and M. Kutrib. On time computability of functions in one-way cellular automata. *Acta Inform.*, 35:329–352, 1998.
8. J.H. Chang, O.H. Ibarra, and A. Vergis. On the power of one-way communication. *J. ACM*, 35:697–726, 1998.
9. C. Choffrut and K. Čulik II. On real-time cellular automata and trellis automata. *Acta Inform.*, 21:393–407, 1984.
10. E.F. Codd. *Cellular Automata.* Academic Press, New York, 1968.
11. S.N. Cole. Real-time computation by n-dimensional iterative arrays of finite-state machines. In *IEEE Symposium on Switching and Automata Theory (SWAT 1966).*
12. S.N. Cole. Real-time computation by n-dimensional iterative arrays of finite-state machines. *IEEE Trans. Comput.*, pages 349–365, 1969.
13. J.C. Dubacq and V. Terrier. Signals for cellular automata in dimension 2 or higher. In *Theoretical Informatics (LATIN 2002)*, LNCS, vol. 2286, pp. 147-163.
14. C.R. Dyer. One-way bounded cellular automata. *Inform. Control*, 44:261–281, 1980.
15. P.C. Fischer. Generation of primes by a one-dimensional real-time iterative array. *J. ACM*, 12:388–394, 1965.
16. M. Gardner. Mathematical games. *Sci. Amer*, 224:112–117, 1971.
17. S. Ginsburg. *The Mathematical Theory of Context-Free Languages.* McGraw Hill, New York, 1996.
18. E. Goto. A minimal time solution of the firing squad problem. Technical report.
19. A. Grasselli. Synchronization of cellular arrays: The firing squad problem in two dimensions. *Inform. Control*, 28:113–124, 1975.
20. J. Hartmanis. Context-free languages and Turing machine computations. In *Proceedings of the Symposia in Applied Mathematics, 19*, pages 42–51, 1967.
21. G.T. Herman, W.H. Liu, S. Rowland, and A. Walker. Synchronization of growing cellular arrays. *Inform. Control*, 25:103–122, 1974.

22. O.H. Ibarra and T. Jiang. On one-way cellular arrays. *SIAM J. Comput.*, 16:1135–1154, 1987.
23. O.H. Ibarra and T. Jiang. Relating the power of cellular arrays to their closure properties. *Theoret. Comput. Sci.*, 57:225–238, 1998.
24. O.H. Ibarra, S.M. Kim, and S. Moran. Sequential machine characterizations of trellis and cellular automata and applications. *SIAM J. Comput.*, 14:426–447, 1985.
25. O.H. Ibarra and M.A. Palis. Some results concerning linear iterative (systolic) arrays. *J. Parallel Distributed Comput.*, 2:182–218, 1985.
26. O.H. Ibarra, M.A. Palis, and S.M. Kim. Fast parallel language recognition by cellular automata. *Theoret. Comput. Sci.*, 41(2): 231-246, 1985.
27. K. Čulik II and S. Dube. An efficient solution of the firing mob problem. *Theoret. Comput. Sci.*, 91:57–69, 1991.
28. K. Čulik II, J. Gruska, and A. Salomaa. Systolic trellis automata: Stability, decidability and complexity. *Inform. Control*, 71:218–230, 1986.
29. K. Čulik II and S. Yu. Iterative tree automata. *Theoret. Comput. Sci.*, 32:227–247, 1984.
30. A.R. Smith III. Simple computation–universal cellular spaces. *J. ACM*, 18:339–353, 1971.
31. A.R. Smith III. Real-time language recognition by one-dimensional cellular automata. *J. Comput. System Sci.*, 6:233–253, 1972.
32. K. Imai and K. Morita. Firing squad synchronization problem in reversible cellular automata. *Theoret. Comput. Sci.*, 165:475–482, 1996.
33. K. Imai and K. Morita. A computation-universal two-dimensional 8-state triangular reversible cellular automaton. *Theoret. Comput. Sci.*, 231:181–191, 2000.
34. C. Iwamoto, T. Hatsuyama, K. Morita, and K. Imai. Constructible functions in cellular automata and their applications to hierarchy results. *Theoret. Comput. Sci.*, 270:797–809, 2002.
35. T. Jiang. The synchronization of nonuniform networks of finite automata. *Inform. Comput.*, 97:234–261, 1992.
36. J. Kari. Reversibility and surjectivity problems of cellular automata. *J. Comput. System Sci.*, 48:149–182, 1994.
37. A. Klein and M. Kutrib. Fast one-way cellular automata. *Theoret. Comput. Sci.*, 1(3):233–250, 2003.
38. K. Kobayashi. The firing squad synchronization problem for a class of polyautomata networks. *J. Comput. System Sci.*, 17:300–318, 1978.
39. K. Kobayashi. On the minimal firing time of the firing squad synchronization problem for polyautomata networks. *Theoret. Comput. Sci.*, 7:149–167, 1978.
40. M. Kutrib. *Automata arrays and context-free languages*, pages 139–148. Kluwer Academic Publishers, 2001.
41. M. Kutrib and R. Vollmar. The firing squad synchronization problem in defective cellular automata. *IEICE Trans. Inf. Syst.*, pages 895–900, 1995.
42. M. Kutrib and Th. Worsch. Investigation of different input modes for cellular automata. In Ch. Jesshope, V. Jossofov and W. Wihelmi, editors, *Parallel Processing by Cellular Automata and Arrays (Parcella 1994)*, Akademie Verlag, Berlin, pp. 141-150, 1994.
43. A. Malcher. Descriptional complexity of cellular automata and decidability questions. *J. Autom. Lang. Comb.*, 7:549–560, 2002.
44. M. Margenstern. Frontier between decidability and undecidability: a survey. *Theoret. Comput. Sci.*, 231:217–251, 2000.

45. B. Martin. Efficient unidimensional universal cellular automaton. In *Mathematical Foundations of Computer Science (MFCS 1992)*, volume 629 of *Lecture Notes in Computer Science*, pages 374–382, Berlin, 1992.
46. B. Martin. A universal cellular automaton in quasi-linear time and its S–m–n form. *Theoret. Comput. Sci.*, 123:199–237, 1994.
47. J. Mazoyer. A minimal time solution to the firing squad synchronization problem with only one bit of information exchanged. Technical report.
48. J. Mazoyer. A six-state minimal time solution to the firing squad synchronization problem. *Theoret. Comput. Sci.*, 50:183–238, 1987.
49. J. Mazoyer and V. Terrier. Signals in one-dimensional cellular automata. *Theoret. Comput. Sci.*, 217:53–80, 1999.
50. E.F. Moore. *The firing squad synchronization problem*, pages 213–214. Addison-Wesley, 1964.
51. F.R. Moore and G.C. Langdon. A generalized firing squad problem. *Inform. Control*, 12:17–33, 1968.
52. K. Morita. Computation-universality of one-dimensional one-way reversible cellular automata. *Inform. Process. Lett.*, 42:325–329, 1992.
53. K. Morita. Reversible simulation of one-dimensional irreversible cellular automata. *Theoret. Comput. Sci.*, 148:157–163, 1995.
54. K. Morita and S. Ueno. Computation-universal models of two-dimensional 16-state reversible cellular automata. *Trans. IEICE*, 75:141, 1992.
55. H.G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 89:25–59, 1953.
56. Y. Rogozhin. Small universal turing machines. *Theoret. Comput. Sci.*, 168:215–240, 1996.
57. P. Rosenstiehl, J.R. Fiksel, and A. Holliger. *Intelligent graphs: Networks of finite automata capable of solving graph problems.* Academic Press, New York.
58. A. Salomaa. *Formal Languages.* Academic Press, New York, 1973.
59. S.R. Seidel. Language recognition and the synchronization of cellular automata. Technical report.
60. I. Shinahr. Two- and three-dimensional firing-squad synchronization problems. *Inform. Control*, 24:163–180, 1974.
61. A.R. Smith. Cellular automata and formal languages. In *IEEE Symposium on Switching and Automata Theory (SWAT 1970).* IEEE Press, pp. 216-224, 1970.
62. R. Sommerhalder and S.C. van Westrhenen. Parallel language recognition in constant time by cellular automata. *Acta Inform.*, 19:397–407, 1983.
63. H. Szwerinski. Time optimal solution of the firing squad synchronization problem for n-dimensional rectangles with the general at an arbitrary position. *Theoret. Comput. Sci.*, 19:305–320, 1982.
64. V. Terrier. On real time one-way cellular array. *Theoret. Comput. Sci.*, 141:331–335, 1995.
65. V. Terrier. Language not recognizable in real time by one-way cellular automata. *Theoret. Comput. Sci.*, 156:281–287, 1996.
66. V. Terrier. Construction of a signal of ratio $n + \lfloor\sqrt{n}\rfloor$. Unpublished manuscript, 2002.
67. H. Umeo. A simple design of time-efficient firing squad synchronization algorithms with fault-tolerance. *IEICE Trans. Inf. Syst.*, pages 733–739, 2004.
68. H. Umeo and N. Kamikawa. A design of real-time non-regular sequence generation algorithms and their implementations on cellular automata with 1-bit inter-cell communications. *Fund. Inform.*, 52:257–275, 2002.

69. H. Umeo and N. Kamikawa. Real-time generation of primes by a 1-bit-communication cellular automaton. *Fund. Inform.*, 58:421–435, 2003.
70. H. Umeo, M. Maeda, M. Hisaoka, and M. Teraoka. A state-efficient mapping scheme for designing two-dimensional firing squad synchronization algorithms. *Fund. Inform.*, 74:603–623, 2006.
71. H. Umeo, K. Morita, and K. Sugata. Deterministic one-way simulation of two-way real-time cellular automata and its related problems. *Inform. Process. Lett.*, 14:158–161, 1982.
72. J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press. Edited and completed by Arthur W. Burks.
73. A. Waksman. An optimum solution to the firing squad synchronization problem. *Inform. Control*, 9:66–78, 1996.

7

Probabilistic Parsing

Mark-Jan Nederhof[1] and Giorgio Satta[2]

[1] School of Computer Science, University of St Andrews
North Haugh, St Andrews, Fife, KY16 9SX, Scotland
http://www.cs.st-andrews.ac.uk/∽mjn

[2] Department of Information Engineering, University of Padua
via Gradenigo, 6/A, I-35131 Padova, Italy
satta@dei.unipd.it

7.1 Introduction

A paper in a previous volume [51] explained *parsing*, which is the process of determining the parses of an input string according to a formal grammar. Also discussed was *tabular parsing*, which solves the task of parsing in polynomial time by a form of dynamic programming. In passing, we also mentioned that parsing of input strings can be easily generalised to parsing of finite automata.

In applications involving natural language, the set of parses for a given input sentence is typically very large. This is because formal grammars often fail to capture subtle properties of structure, meaning and use of language, and consequently allow many parses that humans would not find plausible.

In natural language systems, parsing is commonly one stage of processing amongst several others. The effectiveness of the stages that follow parsing generally relies on having obtained a small set of preferred parses, ideally only one, from amongst the full set of parses. This is called (syntactic) *disambiguation*. There are roughly two ways to achieve this. First, some kind of filter may be applied to the full set of parses, to reject all but a few. This filter may look at the meanings of words and phrases, for example, and may be based on linguistic knowledge that is very different in character from the grammar that was used for parsing.

A second approach is to augment the parsing process so that weights are attached to parses and subparses. The higher the weight of a parse or subparse, the more confident we are that it is correct. This is called *weighted parsing*. If the weights are chosen to define a probability distribution over parses or strings, this may also be called *probabilistic parsing*. Disambiguation is achieved by computing the parse with the highest weight or, where appropriate, highest probability.

The simplest form of probabilistic parsing relies on an assignment of probabilities to individual rules from a context-free grammar. These probabilities are then multiplied upon combination of rules to form parses. Models that are close to this basic idea, such as [18, 21], have been highly influential from the 1990s onward. The success of probabilistic parsing is due to its flexibility and scalability, in contrast to approaches to disambiguation that rely on much deep knowledge of language. For general discussions about statistical natural language processing see [12, 43, 9].

In Section 7.2 we discuss both weighted and probabilistic context-free grammars. We investigate intersection of weighted context-free grammars and finite automata in Section 7.3. By normalisation, discussed in Section 7.4, it can be shown that for the sake of disambiguation we may restrict our attention to probabilistic context-free grammars. Parsing is treated in Section 7.5, and how the probabilities of grammar rules can be obtained empirically is explained in Section 7.6.

Section 7.7 discusses the computation of prefix probabilities, and probabilistic push-down automata are the subject of Section 7.8. By considering semirings, a number of computations involving context-free grammars and push-down automata can be unified, as demonstrated in Section 7.9. We end with additional bibliographic remarks in Section 7.10.

7.2 Weighted and Probabilistic Context-Free Grammars

A *weighted context-free grammar* (WCFG) $\mathcal{G}$ is a 5-tuple (Σ, N, S, R, μ), where (Σ, N, S, R) is a context-free grammar and μ is a mapping from rules in R to positive real numbers. We refer to these numbers as *weights*, and they should be thought of as a measure of the desirability of using the corresponding rules. In general, a rule with a high weight is preferred over one with a low weight.

Let $d = \pi_1 \cdots \pi_m \in R^*$ be a string of rules (or alternatively, of labels that uniquely identify rules), and let α and β be strings of grammar symbols. The expression $\alpha \overset{d}{\Rightarrow} \beta$ means that β can be obtained from α by a left-most derivation in m steps, and the i-th step replaces the left-most nonterminal A_i by γ_i according to rule $\pi_i = (A_i \rightarrow \gamma_i)$. All derivations in this paper are assumed to be left-most. If $S \overset{d}{\Rightarrow} w$, we define the *yield* of d as $y(d) = w$.

We now define $\mu(\alpha \overset{d}{\Rightarrow} \beta)$ to be $\prod_{i=1}^{m} \mu(\pi_i)$ if $\alpha \overset{d}{\Rightarrow} \beta$ holds and to be 0 otherwise. In words, if the expression $\alpha \overset{d}{\Rightarrow} \beta$ denotes a valid left-most derivation, we compute the product of the weights of the used rules, and otherwise we take 0. This notation allows us to define the weight of a string w as:

$$\mu(w) = \sum_{d} \mu(S \overset{d}{\Rightarrow} w). \tag{7.1}$$

In words, to obtain the weight of a string we sum the weights of all left-most derivations of that string. For choices of d such that $S \overset{d}{\Rightarrow} w$ does not denote a valid left-most derivation, nothing is contributed to the sum. If $S \overset{d}{\Rightarrow} w$ holds, we also write $\mu(d)$ in place of $\mu(S \overset{d}{\Rightarrow} w)$.

Example 1. In the grammar below, the rules are labelled by names π_i and the weights are the numbers between brackets.

$$\begin{array}{lll} \pi_1 : S \rightarrow A\ A & (3) \\ \pi_2 : S \rightarrow a\ a & (1) \\ \pi_3 : A \rightarrow a & (2) \end{array}$$

This grammar is ambiguous, as there are two left-most derivations of aa, namely $S \overset{\pi_1}{\Rightarrow} AA \overset{\pi_3}{\Rightarrow} aA \overset{\pi_3}{\Rightarrow} aa$ with weight $\mu(\pi_1) \cdot \mu(\pi_3) \cdot \mu(\pi_3) = 3 \cdot 2 \cdot 2 = 12$ and $S \overset{\pi_2}{\Rightarrow} aa$ with weight $\mu(\pi_2) = 1$. The weight of aa is therefore $\mu(aa) = \mu(S \overset{\pi_1\pi_3\pi_3}{\Rightarrow} aa) + \mu(S \overset{\pi_2}{\Rightarrow} aa) = 12 + 1 = 13$.

We say a WCFG is *convergent* if $\sum_{d,w} \mu(S \overset{d}{\Rightarrow} w)$ is a finite number. A WCFG can be called a *probabilistic context-free grammar* (PCFG) if μ maps all rules to numbers no greater than 1 [28, 29, 10, 73]. Where we are dealing with PCFGs, we will often replace the name μ of the weight assignment by p.

We say a WCFG is *proper* if for every nonterminal A:

$$\sum_{\pi=(A\rightarrow\alpha)} \mu(\pi) = 1. \tag{7.2}$$

In other words, for each nonterminal A in a parse tree or in a sentential form, μ gives us a probability distribution over the rules that we can apply.

A WCFG is said to be *consistent* if:

$$\sum_{d,w} \mu(S \overset{d}{\Rightarrow} w) = 1. \tag{7.3}$$

This means that μ is a probability distribution over derivations of terminal strings. An equivalent statement is that μ is a probability distribution over terminal strings, as:

$$\sum_{d,w} \mu(S \overset{d}{\Rightarrow} w) = \sum_{w} \mu(w). \tag{7.4}$$

Clearly, consistency implies convergence. Properness and consistency are two closely related concepts but, as we will see below, neither implies the other.

An important auxiliary concept for much of the theory that is to follow is the *partition function* Z, which maps each nonterminal A to:

$$Z(A) = \sum_{d,w} \mu(A \overset{d}{\Rightarrow} w). \tag{7.5}$$

Note that a WCFG is consistent if and only if $Z(S) = 1$.

By decomposing derivations into smaller derivations, and by making use of the fact that multiplication distributes over addition, we can rewrite:

$$Z(A) = \sum_{\pi=(A\to\alpha)} \mu(\pi) \cdot Z(\alpha), \tag{7.6}$$

where we define:

$$Z(\epsilon) = 1, \tag{7.7}$$

$$Z(a\beta) = Z(\beta), \tag{7.8}$$

$$Z(B\beta) = Z(B) \cdot Z(\beta), \text{ for } \beta \neq \epsilon. \tag{7.9}$$

The partition function may be approximated by only considering derivations up to a certain depth. We define for all A and $k \geq 0$:

$$Z_k(A) = \sum_{d,w:\, depth(d) \leq k} p(A \overset{d}{\Rightarrow} w), \tag{7.10}$$

where the depth of a left-most derivation is the largest number of rules visited on a path from the root to a leaf in the familiar representation as parse tree. More precisely, $depth(\epsilon) = 0$ and if $\pi = (A \to X_1 \cdots X_m)$ and $X_i \overset{d_i}{\Rightarrow} w_i$ $(1 \leq i \leq m)$, then $depth(\pi d_1 \cdots d_m) = 1 + \max_i depth(d_i)$.

By again decomposing derivations, we obtain a recursive characterisation:

$$Z_{k+1}(A) = \sum_{\pi=(A\to\alpha)} p(\pi) \cdot Z_k(\alpha), \tag{7.11}$$

and $Z_0(A) = 0$ for all A, where we define:

$$Z_k(\epsilon) = 1, \tag{7.12}$$

$$Z_k(a\beta) = Z_k(\beta), \tag{7.13}$$

$$Z_k(B\beta) = Z_k(B) \cdot Z_k(\beta), \text{ for } \beta \neq \epsilon. \tag{7.14}$$

Naturally, for all A:

$$\lim_{k\to\infty} Z_k(A) = Z(A). \tag{7.15}$$

If we interpret (7.6) together with (7.7) through (7.9) as a system of polynomial equations over variables $Z(A)$, for the set of nonterminals $A \in N$, then there may be several solutions. The intended solution, as given by (7.5), is the smallest non-negative solution. This follows from the fact that the operation implied by (7.11) that computes values $Z_{k+1}(A)$ from values $Z_k(B)$ is monotone, and the least fixed-point of this operation corresponds to (7.5), following (7.15).

The values $Z(A)$ may be approximated by computing $Z_k(A)$ for $k = 1, \ldots$ until the values stabilise. Another option is to use Newton's method [22]. In special cases, the solution can be found analytically.

Example 2. Consider the following proper WCFG:

$$\begin{array}{lll} S \rightarrow S\ S & (q) \\ S \rightarrow a & (1-q) \end{array}$$

for a certain choice of q between 0 and 1. Using (7.6) through (7.9), we obtain:

$$Z(S) = q \cdot Z(S)^2 + (1-q). \tag{7.16}$$

We can solve this equation, distinguishing between two cases. If $q \leq \frac{1}{2}$, then $Z(S) = 1$ and if $q > \frac{1}{2}$, then $Z(S) = \frac{1-q}{q}$. We make use of the fact that we need the smallest non-negative solution. It follows that the WCFG is consistent only if $q \leq \frac{1}{2}$. The intuition for the case $q > \frac{1}{2}$ is that probability mass is lost in 'infinite derivations'.

A WCFG can also be consistent without being proper. An example is:

$$\begin{array}{ll} S^{\dagger} \rightarrow S & (\frac{q}{1-q}) \\ S \rightarrow S\ S & (q) \\ S \rightarrow a & (1-q) \end{array}$$

for $\frac{1}{2} < q < 1$.

7.3 Weighted Intersection

It was shown by [5] that context-free languages are closed under intersection with regular languages. The proof relies on the construction of a new CFG out of an input CFG and an input finite automaton. Here we extend that construction by letting the input grammar be a weighted CFG. For an even more general construction, where also the finite automaton is weighted, we refer to [49].

To avoid a number of technical complications, we assume here that the finite automaton has no epsilon transitions, and has only one final state. Thus, a finite automaton (FA) $\mathcal{M}$ is a 5-tuple $(\Sigma, Q, q_0, q_f, \Delta)$, where Σ and Q are two finite sets of *input symbols* and *states*, respectively, q_0 is the *initial* state, q_f is the *final* state, and Δ is a finite set of *transitions*, each of the form $s \overset{a}{\mapsto} t$, where $s, t \in Q$ and $a \in \Sigma$.

For a FA $\mathcal{M}$ as above and a PCFG $\mathcal{G} = (\Sigma, N, S, R, \mu)$ with the same set Σ, we construct a new PCFG $\mathcal{G}_{\cap} = (\Sigma, N_{\cap}, S_{\cap}, R_{\cap}, \mu_{\cap})$, where $N_{\cap} = Q \times (\Sigma \cup N) \times Q$, $S_{\cap} = (q_0, S, q_f)$, and $R_{\cap}$ is the set of rules that is obtained as follows.

- For each $A \rightarrow X_1 \cdots X_m$ in R and each sequence $s_0, \ldots, s_m \in Q$, with $m \geq 0$, let $(s_0, A, s_m) \rightarrow (s_0, X_1, s_1) \cdots (s_{m-1}, X_m, s_m)$ be in $R_{\cap}$; if $m = 0$, the new rule is of the form $(s_0, A, s_0) \rightarrow \epsilon$. Function $\mu_{\cap}$ assigns the same weight to the new rule as μ assigned to the original rule.

- For each $s \overset{a}{\mapsto} t$ in Δ, let $(s, a, t) \to a$ be in $R_\cap$. Function $\mu_\cap$ assigns weight 1 to this rule.

Observe that a rule of $\mathcal{G}_\cap$ is constructed either out of a rule of $\mathcal{G}$ or out of a transition of $\mathcal{M}$. On the basis of this correspondence between rules and transitions of $\mathcal{G}_\cap$, $\mathcal{G}$ and $\mathcal{M}$, it can be stated that each derivation $d_\cap$ in $\mathcal{G}_\cap$ deriving a string w corresponds to a unique derivation d in $\mathcal{G}$ deriving the same string and a unique computation c in $\mathcal{M}$ recognising the same string. Conversely, if there is a derivation d in $\mathcal{G}$ deriving string w, and some computation c in $\mathcal{M}$ recognising the same string, then the pair of d and c corresponds to a unique derivation $d_\cap$ in $\mathcal{G}_\cap$ deriving the same string w. Furthermore, the weights of d and $d_\cap$ are equal, by the definition of $\mu_\cap$.

Parsing of a string $w = a_1 \cdots a_n$ can be seen as a special case of the construction, where there is a linear FA, with states $q_0 = s_0, s_1, \ldots, s_n = q_f$, and transitions of the form $s_{i-1} \overset{a_i}{\mapsto} s_i$ $(1 \leq i \leq n)$. The intersection grammar constructed as explained above can be seen as a succinct representation of all parses of w. As weights are copied unchanged from $\mathcal{G}$ to $\mathcal{G}_\cap$, we can find the parse of w with the highest weight on the basis of $\mathcal{G}_\cap$. We will return to this issue in Section 7.5.

We say a nonterminal in a CFG is *generating* if at least one terminal string can be derived from that nonterminal. We say a nonterminal is *reachable* if a string containing that nonterminal can be derived from the start symbol. A nonterminal is called *useless* if it is non-generating or non-reachable or both. A grammar $\mathcal{G}_\cap$ as obtained above generally contains a large number of useless nonterminals, to the extent that the construction as given may not be practical.

Introduction of non-generating nonterminals can be avoided by constructing rules in a bottom-up phase. That is, a rule is introduced only if all the members in the right-hand side have been found to be generating. This ensures that the left-hand side nonterminal is also generating. In a following top-down phase, the non-reachable nonterminals can be eliminated, by a standard technique that is linear in the size of the grammar [65].

Below, we will assume one more improvement. The motivation is that the number of rules of the form $(s_0, A, s_m) \to (s_0, X_1, s_1) \cdots (s_{m-1}, X_m, s_m)$ is exponential in m. Our improvement effectively postpones enumeration of all relevant combinations of $s_1, \ldots, s_{m-1}$ until (s_0, A, s_m) is found to be reachable in the top-down phase. During the bottom-up phase, given in Figure 7.1, such rules are constructed incrementally by items of the form $(s_0, A \to \alpha \bullet \beta, s_i)$, where $A \to \alpha\beta$ is a rule and $i = |\alpha|$. Existence of such an item in table $\mathcal{I}$ means that there are $s_1, \ldots, s_{i-1}$ such that $(s_0, X_1, s_1), \ldots, (s_{i-1}, X_i, s_i)$ are all generating nonterminals, with $\alpha = X_1 \cdots X_i$. We also have a separate table $\mathcal{N}$ to store such generating nonterminals.

The bottom-up phase is similar to a bottom-up variant of the parsing algorithm by [26], and the complexity is very similar. The time complexity in

```
$first\_phase$:
  $\mathcal{N} = \emptyset$ {table of generating nonterminals for $N_\cap$}
  $\mathcal{I} = \emptyset$ {table of items, partially representing rules for $R_\cap$}
  $\mathcal{A} = \emptyset$ {agenda, items yet to be processed}
  for all $(s \overset{a}{\mapsto} t) \in \Delta$ do
    $add\_symbol(s, a, t)$
  for all $s \in Q$ do
    for all $(A \to \epsilon) \in R$ do
      $\mathcal{A} = \mathcal{A} \cup \{(s, A \to \bullet, s)\}$
  while $\mathcal{A} \neq \emptyset$ do
    choose $(s, A \to \alpha \bullet \beta, t) \in \mathcal{A}$
    $\mathcal{A} = \mathcal{A} - \{(s, A \to \alpha \bullet \beta, t)\}$
    $add\_item(s, A \to \alpha \bullet \beta, t)$

$add\_symbol(s, X, t)$:
  if $(s, X, t) \notin \mathcal{N}$
    $\mathcal{N} = \mathcal{N} \cup \{(s, X, t)\}$
    for all $(r, A \to \alpha \bullet X\beta, s) \in \mathcal{I}$ do
      $\mathcal{A} = \mathcal{A} \cup \{(r, A \to \alpha X \bullet \beta, t)\}$
    for all $(A \to X\beta) \in R$ do
      $\mathcal{A} = \mathcal{A} \cup \{(s, A \to X \bullet \beta, t)\}$

$add\_item(r, A \to \alpha \bullet \beta, s)$:
  if $(r, A \to \alpha \bullet \beta, s) \notin \mathcal{I}$
    $\mathcal{I} = \mathcal{I} \cup \{(r, A \to \alpha \bullet \beta, s)\}$
    if $\beta = \epsilon$
      $add\_symbol(r, A, s)$
    else
      let $X\gamma = \beta$
      for all $(s, X, t) \in \mathcal{N}$ do
        $\mathcal{A} = \mathcal{A} \cup \{(r, A \to \alpha X \bullet \gamma, t)\}$
```

Fig. 7.1. The bottom-up phase of the intersection algorithm. Input are PCFG $\mathcal{G}$ and FA $\mathcal{M}$. The tables $\mathcal{N}$ and $\mathcal{I}$ will be used in the subsequent top-down phase.

our case is cubic in the number of states of $\mathcal{M}$ and linear in the size of $\mathcal{G}$. The space complexity is quadratic in the number of states of $\mathcal{M}$.

Let us now turn to the construction of $\mathcal{G}_\cap$ out of $\mathcal{N}$ and $\mathcal{I}$ in the top-down phase, given in Figure 7.2. From the start symbol (q_0, S, q_f), we descend and construct rules for reachable nonterminals that were also found to be generating in the bottom-up phase. Nonterminals are individually added to $N_\cap$ in such a way that rules cannot be constructed more than once.

Some remarks about the implementation are in order. First, the agenda $\mathcal{A}$ is here represented as a set to avoid the presence of duplicate elements. The maximum number of elements the agenda may contain at any given time is thereby quadratic in the number of states of $\mathcal{M}$. If we alternatively represent

```
second_phase:
  make_rules($q_0, S, q_f$)

make_rules($r, A, s$): {if second argument is nonterminal}
  if $(r, A, s) \notin N_\cap$
    $N_\cap = N_\cap \cup \{(r, A, s)\}$
    for all $\pi = (A \to X_1 \cdots X_m) \in R$ do
      $s_0 = r$
      $s_m = s$
      for all $s_1, \ldots, s_{m-1} \in Q$ such that $(s_0, X_1, s_1), \ldots, (s_{m-1}, X_m, s_m) \in \mathcal{N}$ do
        $\rho = (r, A, s) \to (s_0, X_1, s_1) \cdots (s_{m-1}, X_m, s_m)$
        $R_\cap = R_\cap \cup \{\rho\}$
        $\mu_\cap(\rho) = \mu(\pi)$
        for all $i$ such that $1 \leq i \leq m$ do
          make_rules($s_{i-1}, X_i, s_i$)

make_rules($r, a, s$): {if second argument is terminal}
  if $(r, a, s) \notin N_\cap$
    $N_\cap = N_\cap \cup \{(r, a, s)\}$
    $\rho = (r, a, s) \to a$
    $R_\cap = R_\cap \cup \{\rho\}$
    $\mu_\cap(\rho) = 1$
```

Fig. 7.2. The top-down phase of the intersection algorithm. On the basis of table $\mathcal{N}$, the nonterminals and rules of $\mathcal{G}_\cap$ are constructed, together with the weight function $\mu_\cap$ on rules.

the agenda as a queue or stack, allowing elements to be present more than once, the space complexity may become cubic.

Second, one may use $\mathcal{I}$ in the top-down phase to guide the search for relevant rules from $\mathcal{G}$ and states from $\mathcal{M}$. This process is further simplified by having the bottom-up phase record a list of the reasons why a certain element is in $\mathcal{N}$ or $\mathcal{I}$. For example, if $(r, A \to \alpha X \bullet \beta, t)$ was obtained from $(r, A \to \alpha \bullet X\beta, s)$ and (s, X, t), then the mentioned list for $(r, A \to \alpha X \bullet \beta, t)$ contains amongst others the pair consisting of $(r, A \to \alpha \bullet X\beta, s)$ and (s, X, t). Such a pair is recorded in the list by *add_symbol* if (s, X, t) is added to $\mathcal{N}$ after $(r, A \to \alpha \bullet X\beta, s)$ is added to $\mathcal{I}$, and it is recorded by *add_item* otherwise.

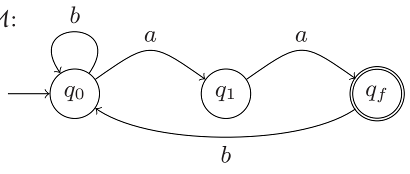
The additional bookkeeping however is at the cost of having larger tables at the end of the bottom-up phase. This increase is from square to cubic in the number of states of $\mathcal{M}$, as a pair consisting of $(r, A \to \alpha \bullet X\beta, s)$ and (s, X, t) contains three states. See also [3, Exercise 4.2.21].

With or without the above optimisations, the space complexity is $\mathcal{O}(|Q|^{r+1})$, where r is the length of the longest right-hand side. This can be reduced to $\mathcal{O}(|Q|^3)$, either by transforming the original grammar to binary form (that is, with $r = 2$) before the intersection, or by refining the intersection algorithm to return a grammar in binary form.

$\mathcal{G}$:

$$\begin{aligned}
S &\rightarrow a\ S\ a\ (\tfrac{1}{2})\\
S &\rightarrow b\ S\ b\ (\tfrac{1}{8})\\
S &\rightarrow b \qquad (\tfrac{3}{8})
\end{aligned}$$

$\mathcal{M}$:



$\mathcal{G}_\cap$:

$$\begin{aligned}
(q_0,S,q_f) &\rightarrow (q_0,a,q_1)\ (q_1,S,q_1)\ (q_1,a,q_f)\ (\tfrac{1}{2})\\
(q_1,S,q_1) &\rightarrow (q_1,a,q_f)\ (q_f,S,q_0)\ (q_0,a,q_1)\ (\tfrac{1}{2})\\
(q_f,S,q_0) &\rightarrow (q_f,b,q_0)\ (q_0,S,q_0)\ (q_0,b,q_0)\ (\tfrac{1}{8})\\
(q_f,S,q_0) &\rightarrow (q_f,b,q_0)\ (q_0,S,q_f)\ (q_f,b,q_0)\ (\tfrac{1}{8})\\
(q_f,S,q_0) &\rightarrow (q_f,b,q_0)\ (\tfrac{3}{8})\\
(q_0,S,q_0) &\rightarrow (q_0,b,q_0)\ (q_0,S,q_0)\ (q_0,b,q_0)\ (\tfrac{1}{8})\\
(q_0,S,q_0) &\rightarrow (q_0,b,q_0)\ (q_0,S,q_f)\ (q_f,b,q_0)\ (\tfrac{1}{8})\\
(q_0,S,q_0) &\rightarrow (q_0,b,q_0)\ (\tfrac{3}{8})
\end{aligned}$$

$$\begin{aligned}
(q_0,b,q_0) &\rightarrow b\ (1)\\
(q_0,a,q_1) &\rightarrow a\ (1)\\
(q_1,a,q_f) &\rightarrow a\ (1)\\
(q_f,b,q_0) &\rightarrow b\ (1)
\end{aligned}$$

Fig. 7.3. Example of intersection of PCFG $\mathcal{G}$ and FA $\mathcal{M}$, resulting in $\mathcal{G}_\cap$, which is presented here without useless nonterminals.

Example 3. Figure 7.3 shows the end result $\mathcal{G}_\cap$ of applying the algorithm in Figures 7.1 and 7.2 on an example PCFG $\mathcal{G}$ and FA $\mathcal{M}$.

7.4 Normalisation

An obvious question is whether general convergent WCFGs have any advantages over proper and consistent PCFGs. In this section we will show that if we are only interested in the ratios between the weights of derivations, rather than in absolute values, the answer is negative. This allows us to restrict our attention to proper and consistent PCFGs for the purpose of disambiguation.

The argument hinges on *normalisation* of WCFGs [69, 1, 49, 52], which can be defined as the construction of a proper and consistent PCFG (Σ, N, S, R, p) out of a convergent WCFG (Σ, N, S, R, μ). The function p is given by:

$$p(\pi) = \frac{\mu(\pi) \cdot Z(\alpha)}{Z(A)}, \tag{7.17}$$

for each rule $\pi = (A \rightarrow \alpha)$. In words, the probability of a rule is normalised to the portion it represents of the total weight mass of derivations from the left-hand side nonterminal A.

That the ratios between weights of derivations are not affected by normalisation follows from a result in [49]:

$$p(S \overset{d}{\Rightarrow} w) = \frac{\mu(S \overset{d}{\Rightarrow} w)}{Z(S)}, \tag{7.18}$$

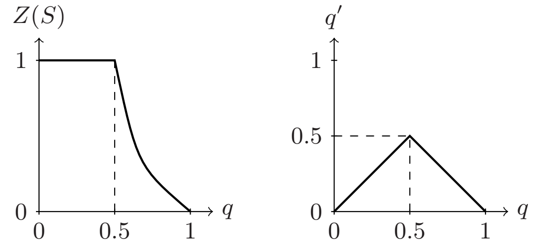


Fig. 7.4. The values of $Z(S)$ and q' as functions of q, for Example 4.

for each derivation d and string w. In other words, the weights of all derivations change by the same factor. Note that this factor is 1 if the original grammar is already consistent. This implies that consistent WCFGs and proper and consistent PCFGs describe the same class of probability distributions over derivations.

Example 4. Let us return to the WCFG from Example 2, with the values of μ between brackets:

$$\begin{array}{ll} S \to S\ S & (q) \\ S \to a & (1-q) \end{array}$$

The result of normalisation is the proper and consistent PCFG below, with the values of p between brackets:

$$\begin{array}{ll} S \to S\ S & (q') \\ S \to a & (1-q') \end{array}$$

For $q \leq \frac{1}{2}$, we have $q' = q$. For $q > \frac{1}{2}$ however, we have:

$$q' = \tfrac{q \cdot Z(SS)}{Z(S)} = \tfrac{q \cdot Z(S)^2}{Z(S)} = q \cdot Z(S) = q \cdot \tfrac{1-q}{q} = 1 - q. \tag{7.19}$$

The values of $Z(S)$ and q' as functions of q are represented in Figure 7.4.

7.5 Parsing

As explained in Section 7.3, context-free parsing is strongly related to computing the intersection of a context-free grammar and a finite automaton. If the input grammar is probabilistic, the probabilities of the rules are simply copied to the intersection grammar. The remaining task is to find the most probable derivation in the intersection grammar.

Note that the problem of finding the most probable derivation in a PCFG does not rely on that PCFG being the intersection of another PCFG and a FA. Let us therefore consider an arbitrary PCFG $\mathcal{G} = (\Sigma, N, S, R, p)$, and our task is to find d and w such that $p(S \overset{d}{\Rightarrow} w)$ is maximal. Let p_{max} denote this maximal value. We further define $p_{max}(X)$ to be the maximal value of $p(X \overset{d}{\Rightarrow} w)$, for any d and w, where X can be a terminal or nonterminal. Naturally, $p_{max} = p_{max}(S)$ and $p_{max}(a) = 1$ for each terminal a.

Much of the following discussion will focus on computing p_{max} rather than on computing a choice of d and w such that $p_{max} = p(S \overset{d}{\Rightarrow} w)$. The justification is that most algorithms to compute p_{max} can be easily extended to a computation of relevant d and w using additional data structures that record how intermediate results were obtained. These data structures however make the discussion less transparent, and are therefore largely ignored.

Consider the graph that consists of the nonterminals as vertices, with an edge from A to B iff there is a rule of the form $A \rightarrow \alpha B \beta$. If $\mathcal{G}$ is non-recursive, then this graph is acyclic. Consequently, the nonterminals can be arranged in a topological sort $A_1, \ldots, A_{|N|}$. This allows us to compute for $j = |N|, \ldots, 1$ in this order:

$$p_{max}(A_j) = \max_{\pi=(A_j \rightarrow X_1 \cdots X_m)} p(\pi) \cdot p_{max}(X_1) \cdot \ldots \cdot p_{max}(X_m). \quad (7.20)$$

The topological sort ensures that any value for a nonterminal in the right-hand side has been computed at an earlier step.

A topological sort can be found in linear time in the size of the graph [19]. See [44] for an application strongly related to ours. In many cases however, there is a topological sort that follows naturally from the way that $\mathcal{G}$ is constructed. For example, assume that $\mathcal{G}$ is the intersection of a PCFG $\mathcal{G}'$ in Chomsky normal form and a linear FA with states $s_0, \ldots, s_n$ as in Section 7.3. We impose an arbitrary linear ordering $\prec_N$ on the set of nonterminals from $\mathcal{G}'$. As topological sort we can now take the linear ordering $\prec$ defined by:

$$\begin{aligned}(s_i, A, s_j) \prec (s_{i'}, A', s_{j'}) \text{ iff } & j > j' \vee \\ & (j = j' \wedge i < i') \vee \\ & (j = j' \wedge i = i' \wedge A \prec_N A').\end{aligned} \quad (7.21)$$

By this ordering, the computation of the values in (7.20) can be seen as a probabilistic extension of CYK parsing [31]. This amounts to a generalised form of Viterbi's algorithm [71], which was designed for probabilistic models with a finite-state structure.

If $\mathcal{G}$ is recursive, then a different algorithm is needed. We may use the fact that the probability of a derivation is always smaller than (or equal to) that of any of its subderivations. The reason is that the probability of a derivation is the product of the probabilities of a list of rules, and these are positive numbers not exceeding 1. We also rely on monotonicity of multiplication, i.e. for any positive numbers c_1, c_2, c_3, if $c_1 < c_2$ then $c_1 \cdot c_3 < c_2 \cdot c_3$.

The algorithm in Figure 7.5 is a special case of an algorithm by Knuth [35], which generalises Dijkstra's algorithm to compute the shortest path in a weighted graph [19]. In each iteration, the value of $p_{max}(A)$ is established for a

```
$\mathcal{E} = \Sigma$
**repeat**
    $\mathcal{F} = \{A \mid A \notin \mathcal{E} \wedge \exists A \rightarrow X_1 \cdots X_m [X_1, \ldots, X_m \in \mathcal{E}]\}$
    **if** $\mathcal{F} = \emptyset$
        report failure and halt
    **for all** $A \in \mathcal{F}$ **do**
        $q(A) = \max_{\substack{\pi=(A \rightarrow X_1 \cdots X_m): \\ X_1, \ldots, X_m \in \mathcal{E}}} p(\pi) \cdot p_{max}(X_1) \cdot \ldots \cdot p_{max}(X_m)$
    choose $A \in \mathcal{F}$ such that $q(A)$ is maximal
    $p_{max}(A) = q(A)$
    $\mathcal{E} = \mathcal{E} \cup \{A\}$
**until** $S \in \mathcal{E}$
output $p_{max}(S)$
```

Fig. 7.5. Knuth's generalisation of Dijkstra's algorithm, applied to finding the most probable derivation in a PCFG.

nonterminal A. The set $\mathcal{E}$ contains all grammar symbols X for which $p_{max}(X)$ has already been established; this is initially Σ, as we set $p_{max}(a) = 1$ for each $a \in \Sigma$. The set $\mathcal{F}$ contains the nonterminals not yet in $\mathcal{E}$ that are candidates to be added next. Each nonterminal A in $\mathcal{F}$ is such that a derivation from A exists consisting of a rule $A \rightarrow X_1 \cdots X_m$, and derivations from $X_1, \ldots, X_m$ matching the values of $p_{max}(X_1)$, ..., $p_{max}(X_m)$ found earlier. The nonterminal A for which such a derivation has the highest probability is then added to $\mathcal{E}$.

Knuth's algorithm can be combined with construction of the intersection grammar, along the lines of [46], which also allows for variants expressing particular parsing strategies. See also [34].

A problem related to finding the most probable parse is to find the k most probable parses. This was investigated by [32, 54, 27].

Much of the discussed theory of probabilistic parsing carries over to more powerful formalisms, such as probabilistic tree adjoining grammars [58, 63].

We want to emphasise that finding the most probable string is much harder than finding the most probable derivation. In fact, the decision version of the former problem is NP-complete if there is a specified bound on the string length [64], and it remains so even if the PCFG is replaced by a probabilistic finite automaton [11]; see also [70]. If the bound on the string length is dropped, then this problem becomes undecidable, as shown in [55, 8].

7.6 Parameter Estimation

Whereas rules of grammars are often written by linguists, or derived from structures defined by linguists, it is very difficult to correctly estimate the probabilities that should be attached to these rules on the basis of linguistic intuitions. Instead, one often relies on two techniques called *supervised* and *unsupervised* estimation.

7.6.1 Supervised Estimation

Supervised estimation relies on explicit access to a sample of data in which one can observe the events whose probabilities are to be estimated. In the case of PCFGs, this sample is a bag $\mathcal{D}$ of derivations of terminal strings, often called a *tree bank*. We assume a fixed order $d_1, \ldots, d_m$ of the derivations in tree bank $\mathcal{D}$. The bag is assumed to be representative for the language at hand, and the probability of a rule is estimated by the ratio of its frequency in the tree bank and the total frequency of rules with the same left-hand side. This is a form of *relative frequency estimation*.

Formally, define $C(\pi, d)$ to be the number of occurrences of rule π in derivation d. Similarly, $C(A, d)$ is the number of times nonterminal A is expanded in derivation d, or equivalently, the sum of all $C(\pi, d)$ such that π has left-hand side A. Summing these numbers for all derivations in the tree bank, we obtain:

$$C(\pi, \mathcal{D}) = \sum_{1 \leq h \leq m} C(\pi, d_h), \tag{7.22}$$

$$C(A, \mathcal{D}) = \sum_{1 \leq h \leq m} C(A, d_h). \tag{7.23}$$

Our estimation for the probability of a rule $\pi = (A \rightarrow \alpha)$ now is:

$$p_{\mathcal{D}}(\pi) = \frac{C(\pi, \mathcal{D})}{C(A, \mathcal{D})}. \tag{7.24}$$

One justification for this estimation is that it maximises the likelihood of the tree bank [16]. This likelihood for given p is defined by:

$$p(\mathcal{D}) = \prod_{1 \leq h \leq m} p(d_h). \tag{7.25}$$

The PCFG that results by taking estimation $p_{\mathcal{D}}$ as above is guaranteed to be consistent [13, 60, 16].

Note that supervised estimation assigns probability 0 to rules that do not occur in the tree bank, which means that probabilistic parsing algorithms ignore such rules. A tree bank may contain zero occurrences of rules because it is too small to contain all phenomena in a language, and some rules that do not occur in one tree bank may in fact be valid and would occur if the tree bank were larger. To circumvent this problem one may apply a form of *smoothing*, which means shifting some probability mass from observed events to those that did not occur. Rules that do not occur in the tree bank thereby obtain a small but non-zero probability. For a study of smoothing techniques used for natural language processing, see [14].

Example 5. Consider the following CFG:

$$\begin{aligned}
\pi_1 &: S \rightarrow S\ S \\
\pi_2 &: S \rightarrow a\ S\ b \\
\pi_3 &: S \rightarrow a\ b \\
\pi_4 &: S \rightarrow b\ a \\
\pi_5 &: S \rightarrow c
\end{aligned}$$

Assume a tree bank consisting of only two derivations, $\pi_1\pi_3\pi_5$ and $\pi_2\pi_3$, with yields *abc* and *aabb*, respectively. Without smoothing, the estimation is $p(\pi_1) = \frac{1}{5}$, $p(\pi_2) = \frac{1}{5}$, $p(\pi_3) = \frac{2}{5}$, $p(\pi_4) = \frac{0}{5}$, $p(\pi_5) = \frac{1}{5}$.

7.6.2 Unsupervised Estimation

We define an (unannotated) *corpus* as a bag $\mathcal{W}$ of strings in a language. As in the case of tree banks, the bag is assumed to be representative for the language at hand. We assume a fixed order $w_1, \ldots, w_m$ of the strings in corpus $\mathcal{W}$. Estimation of a probability assignment p to rules of a CFG on the basis of a corpus is called unsupervised as there is no direct access to frequencies of rules. A string from the corpus may possess several derivations, each representing different bags of rule occurrences.

A common unsupervised estimation for PCFGs is a form of Expectation-Maximisation (EM) algorithm [20]. It computes a probability assignment p by successive refinements $p_0, p_1, \ldots$, until the values stabilise. The initial assignment p_0 may be arbitrarily chosen, and subsequent estimates p_{t+1} are computed on the basis of p_t, in a way to be explained below. In each step, the likelihood $p_t(\mathcal{W})$ of the corpus increases. This likelihood for given p is defined by:

$$p(\mathcal{W}) = \prod_{1 \leq h \leq m} p(w_h). \tag{7.26}$$

The algorithm converges to a local optimum (or a saddlepoint) with respect to the likelihood of the corpus, but no algorithm is known to compute the global optimum, that is, the assignment p such that $p(\mathcal{W})$ is maximal.

Computation of p_{t+1} on the basis of p_t corresponds to a simple idea. With unsupervised estimation, we do not have access to a single derivation for each string in the corpus, and therefore cannot determine frequencies of rules by simple counts. Instead, we consider all derivations for each string, and the counts we would obtain for individual derivations are combined by taking a weighted average. The weighting of this average is determined by the current assignment p_t, which offers us probabilities $\frac{p_t(d)}{p_t(w)}$, where $y(d) = w$, which is the conditional probability of derivation d given string w.

More precisely, an estimated count $C_t(\pi)$ of a rule π in a corpus, given assignment p_t, can be defined by:

$$C_t(\pi) = \sum_{1 \le h \le m} \sum_{d: y(d) = w_h} \frac{p_t(d)}{p_t(w_h)} \cdot C(\pi, d). \tag{7.27}$$

Similarly:

$$C_t(A) = \sum_{1 \le h \le m} \sum_{d: y(d) = w_h} \frac{p_t(d)}{p_t(w_h)} \cdot C(A, d). \tag{7.28}$$

Using these values we compute the next estimation $p_{t+1}(\pi)$ for each rule $\pi = (A \to \alpha)$ as:

$$p_{t+1}(\pi) = \frac{C_t(\pi)}{C_t(A)}. \tag{7.29}$$

Note the similarity of this to (7.24).

Example 6. Consider the CFG from Example 5, with a corpus consisting of strings $w_1 = abc$, $w_2 = acb$ and $w_3 = abab$. The first two strings can only be derived by $d_1 = \pi_1\pi_3\pi_5$ and $d_2 = \pi_2\pi_5$, respectively. However, w_3 is ambiguous as it can be derived by $d_3 = \pi_1\pi_3\pi_3$ and $d_4 = \pi_2\pi_4$.

For a given p_t, we have:

$$\begin{aligned}
C_t(\pi_1) &= 1 + \frac{p_t(d_3)}{p_t(w_3)} \\
C_t(\pi_2) &= 1 + \frac{p_t(d_4)}{p_t(w_3)} \\
C_t(\pi_3) &= 1 + 2 \cdot \frac{p_t(d_3)}{p_t(w_3)} \\
C_t(\pi_4) &= \frac{p_t(d_4)}{p_t(w_3)} \\
C_t(\pi_5) &= 2 \\
C_t(S) &= 5 + 3 \cdot \frac{p_t(d_3)}{p_t(w_3)} + 2 \cdot \frac{p_t(d_4)}{p_t(w_3)}
\end{aligned}$$

The assignment that p_t converges to depends on the initial choice of p_0. We investigate two such initial choices:

	$p_t(\pi_1)$	$p_t(\pi_2)$	$p_t(\pi_3)$	$p_t(\pi_4)$	$p_t(\pi_5)$
$t = 0$	0.200	0.200	0.200	0.200	0.200
$t = 1$	0.163	0.256	0.186	0.116	0.279
$t = 2$	0.162	0.257	0.184	0.117	0.279
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$t = \infty$	0.160	0.260	0.180	0.120	0.280

and:

	$p_t(\pi_1)$	$p_t(\pi_2)$	$p_t(\pi_3)$	$p_t(\pi_4)$	$p_t(\pi_5)$
$t = 0$	0.100	0.100	0.600	0.100	0.100
$t = 1$	0.229	0.156	0.330	0.028	0.257
$t = 2$	0.236	0.146	0.344	0.019	0.255
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$t = \infty$	0.250	0.125	0.375	0.000	0.250

In the first case, the likelihood of the corpus is $2.14 \cdot 10^{-5}$ and in the second case $2.57 \cdot 10^{-5}$.

As strings may allow a large number of derivations, a direct implementation of (7.27) and (7.28) is often not feasible. To obtain a more practical algorithm, we first rewrite $C_t(\pi)$ as below. Treatment of $C_t(A)$ is similar.

$$C_t(\pi) = \sum_{1 \leq h \leq m} \frac{1}{p_t(w_h)} \sum_{d:y(d)=w_h} p_t(d) \cdot C(\pi, d). \tag{7.30}$$

The value $p_t(w_h)$ is just $Z(S_{t,h})$, where $S_{t,h}$ is the start symbol of the intersection of the PCFG with probability assignment p_t and the linear FA accepting the singleton language $\{w_h\}$. How this value can be computed has already been explained in Section 7.2. Let us therefore concentrate on the second part of the above expression, fixing an assignment p, rule $\pi = (A \to \alpha)$ and string $w = a_1 \cdots a_n$. We rewrite:

$$\begin{aligned}\sum_{d:y(d)=w} p(d) \cdot C(\pi, d) &= \sum_{i,j} \sum_{d_1,d_2,d_3,\beta} p(S \overset{d_1}{\Rightarrow} a_1 \cdots a_i A\beta) \cdot p(\pi) \cdot \\ &\qquad\qquad p(\alpha \overset{d_2}{\Rightarrow} a_{i+1} \cdots a_j) \cdot \\ &\qquad\qquad p(\beta \overset{d_3}{\Rightarrow} a_{j+1} \cdots a_n) \qquad (7.31)\\ &= \sum_{i,j} \mathit{outer}(A, i, j) \cdot p(\pi) \cdot \mathit{inner}(\alpha, i, j), \qquad (7.32)\end{aligned}$$

where we define:

$$\mathit{outer}(A, i, j) = \sum_{d_1,d_3,\beta} p(S \overset{d_1}{\Rightarrow} a_1 \cdots a_i A\beta) \cdot p(\beta \overset{d_3}{\Rightarrow} a_{j+1} \cdots a_n), \tag{7.33}$$

$$\mathit{inner}(\alpha, i, j) = \sum_{d_2} p(\alpha \overset{d_2}{\Rightarrow} a_{i+1} \cdots a_j). \tag{7.34}$$

The intuition is that the occurrences of π in the different d such that $y(d) = w$ are grouped according to the substring $a_{i+1} \cdots a_j$ that they cover. For each choice of i and j we look at the sum of probabilities of matching derivations, dividing them into the subderivations that are 'inside' and 'outside' the relevant occurrence of π.

The values of $\mathit{inner}(\alpha, i, j)$ can be computed similarly to the computation of the partition function Z, which was explained in Section 7.2. For the remaining values, we have:

$$\begin{aligned}&outer(A,i,j) = \\ &\quad \delta(A = S \wedge i = 0 \wedge j = n) + \\ &\quad \sum_{\pi=(B\rightarrow\gamma A\eta),i',j'} outer(B,i',j') \cdot p(\pi) \cdot inner(\gamma,i',i) \cdot inner(\eta,j,j')\end{aligned} \quad (7.35)$$

with δ defined to return 1 if its argument is true and 0 otherwise. Here we divide the derivations 'outside' a nonterminal occurrence into parts outside parent nonterminal occurrences, and the parts below siblings on the left and on the right. A special case is if the nonterminal occurrence can be the root of the parse tree, which corresponds to a value of 1, which is the product of zero rule probabilities.

If we fill in the values for *inner*, we obtain a system of linear equations with $outer(A,i,j)$ as variables, which can be solved in polynomial time. The system is of course without cyclic dependencies if the grammar is without cycles.

The algorithm we have described is called the *inside-outside algorithm* [4, 39, 31, 57]. It generalises the *forward-backward algorithm* for probabilistic models with a finite-state structure [6]. Generalised PCFGs, with right-hand sides representing regular languages, were considered in [37]. The inside-outside algorithm is guaranteed to result in consistent PCFGs [60, 16, 52].

7.7 Prefix Probabilities

Let p be the probability assignment of a PCFG. The *prefix probability* of a string w is defined to be $Pref(w) = \sum_v p(wv)$. Prefix probabilities have important applications in speech recognition. For example, assume a prefix of the input is w, and the next symbol suggested by the speech recogniser is a. The probability that a is the next symbol according to the PCFG is given by:

$$\frac{Pref(wa)}{Pref(w)}. \quad (7.36)$$

For given w, there may be infinitely many v such that $p(wv) > 0$. As we will show, the difficulty of summing infinitely many values can be overcome by isolating a finite number of auxiliary values whose computation can be carried out 'off-line', that is, independent of any particular w. On the basis of these values, $Pref(w)$ can be computed in cubic time for any given w.

We first extend left-most derivations to 'dotted' derivations written as $S \overset{d}{\Rightarrow} w \bullet \alpha$. The dot indicates a position in the sentential form separating the known prefix w and a string α of grammar symbols together generating an unknown suffix v. No symbol to the right of the dot may be rewritten. The rationale is that this would lead to probability mass being included more than once in the theory that is to follow.

Formally, a dotted derivation can be either $A \overset{\epsilon}{\Rightarrow} \bullet A$, which represents the empty derivation, or it can be of the form $A \overset{d\pi}{\Rightarrow} wv \bullet \alpha\beta$, where $\pi = (B \rightarrow v\alpha)$ with $v \neq \epsilon$, to represent the (left-most) derivation $A \overset{d}{\Rightarrow} wB\beta \overset{\pi}{\Rightarrow} wv\alpha\beta$.

In the remainder of this section, we will assume that the PCFG is proper and consistent. This allows us to rewrite:

$$\mathit{Pref}(w) = \sum_{d,\alpha} p(S \overset{d}{\Rightarrow} w \bullet \alpha) \cdot \sum_{d',v} p(\alpha \overset{d'}{\Rightarrow} v) = \sum_{d,\alpha} p(S \overset{d}{\Rightarrow} w \bullet \alpha). \quad (7.37)$$

Note that derivations leading from any α in the above need not be considered individually, as the sum of their probabilities is 1 for proper and consistent PCFGs.

Example 7. We investigate the prefix probability of bb, for the following PCFG:

$$\begin{array}{lll} \pi_1 : S \rightarrow A\ a & (0.2) \\ \pi_2 : S \rightarrow b & (0.8) \\ \pi_3 : A \rightarrow S\ a & (0.4) \\ \pi_4 : A \rightarrow S\ b & (0.6) \end{array}$$

The set of derivations d such that $S \overset{d}{\Rightarrow} bb \bullet \alpha$, some α, can be described by the regular expression $(\pi_1(\pi_3 \cup \pi_4))^*\pi_1\pi_4\pi_2$. By summing the probabilities of these derivations, we get:

$$\begin{aligned} \mathit{Pref}(bb) &= \sum_{m \geq 0} \big(p(\pi_1) \cdot (p(\pi_3) + p(\pi_4))\big)^m \cdot p(\pi_1) \cdot p(\pi_4) \cdot p(\pi_2) \\ &= \sum_{m \geq 0} p(\pi_1)^m \cdot p(\pi_1) \cdot p(\pi_4) \cdot p(\pi_2). \end{aligned}$$

As $\sum_{m \geq 0} p(\pi_1)^m = \frac{1}{1-p(\pi_1)} = 1.25$, we obtain:

$$\mathit{Pref}(bb) = 1.25 \cdot 0.2 \cdot 0.6 \cdot 0.8 = 0.12.$$

The remainder of this section derives a practical solution for computing the value in (7.37), due to [30]. This requires that the underlying CFG is in Chomsky normal form, or more precisely that every rule has the form $A \rightarrow BC$ or $A \rightarrow a$. We will ignore rules $S \rightarrow \epsilon$ here.

We first distinguish two kinds of subderivation. For the first kind, the yield falls entirely within the known prefix $w = a_1 \cdots a_n$. For the second, the yield includes the boundary between known prefix w and unknown suffix v. We do not have to investigate the third kind of subderivation, whose yield falls entirely within the unknown suffix, because the factors involved are always 1, as explained before.

For subderivations within the known prefix we have values of the form:

$$\textstyle\sum_d p(A \overset{d}{\Rightarrow} a_{i+1} \cdots a_j), \quad (7.38)$$

with i and j between 0 and n. These values can be computed using techniques already discussed, in Section 7.2 for Z, and in Section 7.6.2 for *inner*. Here, the computation can be done in cubic time in the length of the prefix, since there are no cyclic dependencies.

Let us now look at derivations at the boundary between w and v. If the relevant part of w is empty, we have:

$$\sum_{d} p(A \overset{d}{\Rightarrow} \bullet\, A) = 1. \tag{7.39}$$

If the relevant part of w is only one symbol a_n, we have:

$$\sum_{d,\alpha} p(A \overset{d}{\Rightarrow} a_n \bullet \alpha) = \sum_{\pi=(B\to a_n)} \sum_{d,\alpha} p(A \overset{d}{\Rightarrow} B\alpha) \cdot p(\pi). \tag{7.40}$$

Here B plays the role of the last nonterminal in a path in a parse tree from A down to a_n, taking the left-most child at each step.

It is easy to see that:

$$\sum_{d,\alpha} p(A \overset{d}{\Rightarrow} B\alpha) = \delta(A = B) + \sum_{\pi=(A\to CD)} p(\pi) \cdot \sum_{d,\alpha} p(C \overset{d}{\Rightarrow} B\alpha). \tag{7.41}$$

If we replace expressions of the form $\sum_{d,\alpha} p(A \overset{d}{\Rightarrow} B\alpha)$ by variables $\mathit{chain}(A, B)$, then (7.41) represents a system of linear equations, for fixed B and different A. This system can be solved with a time complexity that is cubic in the number of nonterminals. Note that this is independent of the known prefix w, and is therefore an off-line computation.

If the derivation covers a larger portion of the prefix ($i + 1 < n$) we have:

$$\begin{aligned}
&\sum_{d,\alpha} p(A \overset{d}{\Rightarrow} a_{i+1} \cdots a_n \bullet \alpha) = \\
&\quad \sum_{\pi=(D\to BC)} \sum_{d,\alpha} p(A \overset{d}{\Rightarrow} D\alpha) \cdot p(\pi) \cdot \\
&\quad \sum_{k:i<k\leq n} \sum_{d_1} p(B \overset{d_1}{\Rightarrow} a_{i+1} \cdots a_k) \cdot \sum_{d_2,\beta} p(C \overset{d_2}{\Rightarrow} a_{k+1} \cdots a_n \bullet \beta).
\end{aligned} \tag{7.42}$$

The intuition is as follows. In a path in the parse tree from the indicated occurrence of A to the occurrence of a_{i+1}, there is a first node, labelled B, whose yield is entirely within the known prefix. The yield of its sibling, labelled C, includes the remainder of the prefix to the right as well as part of the unknown suffix.

We already know how to compute $\sum_{d,\alpha} p(A \overset{d}{\Rightarrow} D\alpha)$ and $\sum_{d_1} p(B \overset{d_1}{\Rightarrow} a_{i+1} \cdots a_k)$. If we now replace expressions of the form $\sum_{d,\alpha} p(A \overset{d}{\Rightarrow} a_{i+1} \cdots a_n \bullet \alpha)$ in (7.42) by variables *prefix_inside*(A, i), then we obtain a system of equations, which define values *prefix_inside*(A, i) in terms of *prefix_inside*(B, k)

with $k > i$. These values can be computed in quadratic time if the other values in (7.42) have already been obtained. The resulting value of $\mathit{prefix\_inside}(S, 0)$ is the required prefix probability of w.

In many applications, the prefix probabilities need to be computed for increasingly long strings. For example in real-time speech recognition, the input grows as the acoustic signal is processed and sequences of sounds are recognised as words. The algorithm above has the disadvantage that the values in (7.42) are specific to the length n of the prefix w. If w grows on the right by one more symbol, the computation of the values has to be done anew. The algorithm by [66], which is based on Earley's algorithm, suffers less from this problem. Most of the values it computes can be reused as the prefix grows. A very similar algorithm was described by [56]. It differs from [66] in that it does not explicitly isolate any off-line computations.

Prefix probabilities for tree adjoining and linear indexed grammars were investigated by [48, 47].

7.8 Probabilistic Push-Down Automata

A paper in a previous volume [51] argued that a parsing strategy can be formalised as a mapping from CFGs to push-down automata that preserves the described languages. In this section we investigate the extension of this notion to probabilistic parsing strategies [53], which are to preserve probability distributions over strings.

As in [51], our type of push-down automaton does not possess states. Hence, a *push-down automaton* (PDA) $\mathcal{M}$ is a 5-tuple (Σ, Γ, X_{init}, X_{final}, Δ), where Σ is a finite set of *input symbols*, Γ is a finite set of *stack symbols*, $X_{init} \in \Gamma$ is the *initial stack symbol*, $X_{final} \in \Gamma$ is the *final stack symbol*, and Δ is the set of *transitions*. Each transition can have one of the following three forms: $X \overset{\epsilon}{\mapsto} XY$ (a push transition), $YX \overset{\epsilon}{\mapsto} Z$ (a pop transition), or $X \overset{x}{\mapsto} Y$ (a swap transition); here $X, Y, Z \in \Gamma$, $x \in \Sigma \cup \{\epsilon\}$. Note that in our notation, stacks grow from left to right, i.e., the top-most stack symbol will be found at the right end.

Without loss of generality, we assume that any PDA is such that for a given stack symbol $X \neq X_{final}$, there are either one or more push transitions $X \overset{\epsilon}{\mapsto} XY$, or one or more pop transitions $YX \overset{\epsilon}{\mapsto} Z$, or one or more swap transitions $X \overset{x}{\mapsto} Y$, but no combinations of different kinds of transition. If a PDA does not satisfy this normal form, it can easily be brought in this form by introducing for each stack symbol $X \neq X_{final}$ three new stack symbols X_{push}, X_{pop} and X_{swap} and new swap transitions $X \overset{\epsilon}{\mapsto} X_{push}$, $X \overset{\epsilon}{\mapsto} X_{pop}$ and $X \overset{\epsilon}{\mapsto} X_{swap}$. In each existing transition that operates on top-of-stack X, we then replace X by one from X_{push}, X_{pop} or X_{swap}, depending on the type of that transition. We also assume that X_{final} does not occur in the left-hand side of any transition, again without loss of generality.

As usual, the process of recognition of a string w starts with a configuration consisting of the singleton stack X_{init}. If a list of transitions leads to singleton stack X_{final} when the entire input w has been scanned, then we say that w is recognised. Such a list of transitions is called a *computation*. The language accepted by the PDA is the set of all strings that can be recognised. We assume below that a PDA is always *reduced*, which means that each stack symbol can be used in some computation that recognises a string. For more precise definitions, we refer to [51].

A *weighted push-down automaton* (WPDA) $\mathcal{M}$ is a 6-tuple (Σ, Γ, X_{init}, X_{final}, Δ, μ), where (Σ, Γ, X_{init}, X_{final}, Δ) is a PDA, and μ is a mapping from transitions in Δ to positive real numbers. Thereby, a WPDA assigns weights to computations and strings, in the same way as WCFGs assign weights to derivations and strings.

A *probabilistic push-down automaton* (PPDA) is a WPDA with the restriction that the values assigned to transitions are no greater than 1 [61]. Consistency is defined as for WCFGs. We say a WPDA is *proper* if:

- $\Sigma_{\tau=(X \overset{\epsilon}{\mapsto} XY)}\ p(\tau) = 1$ for each $X \in \Gamma$ such that there is at least one transition of the form $X \overset{\epsilon}{\mapsto} XY$;
- $\Sigma_{\tau=(X \overset{x}{\mapsto} Y)}\ p(\tau) = 1$ for each $X \in \Gamma$ such that there is at least one transition of the form $X \overset{x}{\mapsto} Y$; and
- $\Sigma_{\tau=(YX \overset{\epsilon}{\mapsto} Z)}\ p(\tau) = 1$ for each $X, Y \in \Gamma$ such that there is at least one transition of the form $YX \overset{\epsilon}{\mapsto} Z$.

For each stack that may arise in the recognition of a string, exactly one of the above three clauses applies, depending on the symbol on top (provided this is not X_{final}). The conditions ensure that the sum of probabilities of next possible transitions is always 1.

An obvious question is whether parsing strategies, mapping CFGs to PDAs, preserve the capacity to describe probability distributions on strings. In many cases, PDAs are able to describe a wider range of probability distributions than the CFGs they were derived from by a parsing strategy.

Consider for example the parsing strategy of top-down parsing. The stack symbols of the constructed PDA are of the form $[A \to \alpha \bullet \beta]$, where $A \to \alpha\beta$ is a rule in the CFG. The transitions are given by:

- $[A \to \alpha \bullet a\beta] \overset{a}{\mapsto} [A \to \alpha a \bullet \beta]$ for each rule $A \to \alpha a \beta$;
- $[A \to \alpha \bullet B\beta] \overset{\epsilon}{\mapsto} [A \to \alpha \bullet B\beta]\ [B \to \bullet\, \gamma]$ for each pair of rules $A \to \alpha B \beta$ and $\pi = B \to \gamma$; and
- $[A \to \alpha \bullet B\beta]\ [B \to \gamma\, \bullet] \overset{\epsilon}{\mapsto} [A \to \alpha B \bullet \beta]$.

We assume without loss of generality that the start symbol has only one defining rule, say $S \to \sigma$. The initial stack symbol is then $[S \to \bullet\, \sigma]$ and the final stack symbol is $[S \to \sigma\, \bullet]$.

The probability distribution described by a proper and consistent PCFG can be carried over to a PPDA implementing the top-down parsing strategy if we let transitions of the first and third kind above have probability 1, and let those of the second kind have the same probability as the rule $B \to \gamma$ from the PCFG.

The reverse does not hold in general. In the PPDA we may assign different probabilities to two different transitions of the form:

- $[A \to \alpha \bullet B\beta] \overset{\epsilon}{\mapsto} [A \to \alpha \bullet B\beta]\ [B \to \bullet\ \gamma]$; and
- $[A' \to \alpha' \bullet B\beta'] \overset{\epsilon}{\mapsto} [A' \to \alpha' \bullet B\beta']\ [B \to \bullet\ \gamma]$.

Such a distinction between the different contexts for an occurrence of nonterminal B cannot normally be encoded into the original CFG. This observation is related to a technique from [17] that allows probability distributions more refined than those that can be expressed in terms of a given CFG. See also [33].

Example 8. Consider the CFG:

$$\begin{aligned}\pi_1 &: S \to A\\ \pi_2 &: A \to a\ B\\ \pi_3 &: A \to b\ B\\ \pi_4 &: B \to c\\ \pi_5 &: B \to d\end{aligned}$$

If p is the probability distribution over strings induced by a proper PCFG that extends the CFG above, then we must have:

$$\frac{p(ac)}{p(ad)} = \frac{p(bc)}{p(bd)}. \tag{7.43}$$

Another way of looking at this is that we have a 2-dimensional parameter space, as there are only two free parameters: once we choose $p(\pi_2)$ and $p(\pi_4)$, then $p(\pi_3)$ must be $1-p(\pi_2)$ and $p(\pi_5)$ must be $1-p(\pi_4)$. Naturally $p(\pi_1) = 1$.

Consider now the corresponding top-down PDA. If we are to turn this into a proper PPDA, all transitions must have probability 1, except the following six:

- $[S \to \bullet\ A] \overset{\epsilon}{\mapsto} [S \to \bullet\ A]\ [A \to \bullet\ aB]$,
- $[S \to \bullet\ A] \overset{\epsilon}{\mapsto} [S \to \bullet\ A]\ [A \to \bullet\ bB]$,
- $[A \to a \bullet B] \overset{\epsilon}{\mapsto} [A \to a \bullet B]\ [B \to \bullet\ c]$,
- $[A \to a \bullet B] \overset{\epsilon}{\mapsto} [A \to a \bullet B]\ [B \to \bullet\ d]$,
- $[A \to b \bullet B] \overset{\epsilon}{\mapsto} [A \to b \bullet B]\ [B \to \bullet\ c]$, and
- $[A \to b \bullet B] \overset{\epsilon}{\mapsto} [A \to b \bullet B]\ [B \to \bullet\ d]$.

Now (7.43) no longer restricts the space of available probability distributions. Seen in a different way, we have a 3-dimensional parameter space, as there are three free parameters; the probabilities of the first, third and fifth transitions above determine those of the others.

Another parsing strategy that preserves the allowable probability distributions is left-corner parsing [68]. This preservation does not hold for all parsing strategies however, as pointed out for bottom-up parsing by [1]. Another strategy for which it does not hold is LR parsing. In [53], an example is given of a PCFG with a probability distribution that cannot be expressed in terms of the corresponding PDA implementing the LR strategy. However, as shown by [50], this problem disappears if we abandon the requirement that the PPDA be proper.

7.9 Semirings

Let us compare the computation of Z (Section 7.2) with the computation of p_{max} (Section 7.5). An important similarity is that values coming from members in the right-hand side of a rule are multiplied, as can be witnessed in both (7.9) and (7.20). An important difference is that in (7.6) we add the values coming from alternative derivations, whereas in (7.20) these values are combined by maximisation. By allowing other operations in place of those mentioned above, possibly with another domain of weights, we obtain a general class of computations involving context-free grammars. The domain and operations are subject to a number of constraints, which can be expressed as an algebraic structure.

Formally, a *semiring* is a 5-tuple $(\mathbb{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, where $\mathbb{D}$ is a set, $\oplus$ and $\otimes$ are binary operations on $\mathbb{D}$, and $\mathbf{0}, \mathbf{1} \in \mathbb{D}$, with the following properties for all $a, b, c \in \mathbb{D}$:

additive identity $a \oplus \mathbf{0} = \mathbf{0} \oplus a = a$,
additive commutativity $a \oplus b = b \oplus a$,
additive associativity $(a \oplus b) \oplus c = a \oplus (b \oplus c)$,
multiplicative identity $a \otimes \mathbf{1} = \mathbf{1} \otimes a = a$,
annihilation $a \otimes \mathbf{0} = \mathbf{0} \otimes a = \mathbf{0}$,
multiplicative associativity $(a \otimes b) \otimes c = a \otimes (b \otimes c)$,
distributivity $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$.

The semiring $(\mathbb{R}^+, +, \cdot, 0, 1)$ underlies the computation of Z, where $\mathbb{R}^+$ stands for the non-negative real numbers, and $+$ and $\cdot$ stand for ordinary addition and multiplication. By replacing $+$ by max, we obtain the semiring $(\mathbb{R}^+, \max, \cdot, 0, 1)$, which underlies the computation of p_{max}. These and several other semirings were discussed in relation to context-free grammars by [25]. Semirings are firmly rooted in the theory of context-free grammars and other formalisms, often in connection with formal power series [36]. For applications of semirings with a focus on finite-state transducers, see [45].

In the remainder of this section, we investigate the semiring $(\mathbb{R}^+ \cup \{\infty\}, \min, +, \infty, 0)$. The domain includes the symbol ∞, which also acts as the 'zero' element. This means that $\min(a, \infty) = \min(\infty, a) = a$ and $a + \infty = \infty + a = \infty$ for all a. We will show that this semiring is useful for error correction of programming languages, which is closely related to the problem of computing the optimal parse on the basis of WCFGs, as is known from [67].

Assume a CFG $\mathcal{G}$, and assume two functions on terminals, called d and i, and a function s on pairs of terminals. These functions define the costs of correcting a string by deleting or inserting a terminal, or by substituting one terminal by another. Costs are non-negative real numbers. We assume that $s(a,b) \leq d(a) + i(b)$, or in words, it is at least as costly to delete a and insert b as to substitute a by b. Naturally, $s(a,a) = 0$ for all a, which means that leaving a terminal unaffected can be treated as substituting it by itself.

The *minimum edit distance* between two strings w and v, denoted by $dist(w,v)$, is defined as the minimum sum of costs of a list of deletions, insertions and substitutions needed to turn w into v [72]. We are now asked to solve the following problem. Given a string w, compute the string v in the language generated by $\mathcal{G}$ that minimises $dist(w,v)$. Similarly to our presentation in Section 7.5, the algorithm we will show computes this minimal value $dist(w,v)$, but not the relevant v itself nor the used edit operations. These can be computed by a simple extension of the basic mechanism.

Let us assume a PDA $\mathcal{A}$ instead of a CFG as representation of a context-free language, which slightly simplifies the discussion. The PDA may implement any parsing strategy. We now construct a WPDA $\mathcal{A}'$ as follows. For each stack symbol X in $\mathcal{A}$, $\mathcal{A}'$ has two distinct stack symbols X and X_{del}. A symbol of the form X_{del} will be on top of the stack immediately after a substitution, or at the beginning of the input. While it is on top of the stack, we allow an uninterrupted sequence of deletions, but no other actions. We thereby effectively force a canonical ordering on the edit operation and stack manipulations, placing deletions as early as possible. If the initial stack symbol of $\mathcal{A}$ is X, then that of $\mathcal{A}'$ is X_{del}. Further:

- Pop and push transitions are copied unchanged from $\mathcal{A}$ to $\mathcal{A}'$. Also swap transitions of the form $X \overset{\epsilon}{\mapsto} Y$ are copied unchanged. The weight of all these transitions is 0.
- For each transition $X \overset{a}{\mapsto} Y$ in $\mathcal{A}$, $\mathcal{A}'$ has the following transitions:
  - $X \overset{b}{\mapsto} Y_{del}$ with weight $s(b,a)$, for each b, and
  - $X \overset{\epsilon}{\mapsto} Y$ with weight $i(a)$.
- For each stack symbol X_{del}, $\mathcal{A}'$ has the following transitions:
  - $X_{del} \overset{a}{\mapsto} X_{del}$ with weight $d(a)$, for each a, and
  - $X_{del} \overset{\epsilon}{\mapsto} X$ with weight 0.

As dictated by the specified semiring, weights in a computation are added. In the presence of ambiguity, we take the minimum weight of all the computations that recognise a string.

Example 9. Consider the following grammar and the top-down PDA obtained from it:

$$
\begin{array}{ll}
[S \rightarrow \bullet\ aAa]_{del} \overset{\epsilon}{\mapsto} [S \rightarrow \bullet\ aAa] & (0) \\
[S \rightarrow \bullet\ aAa] \overset{\epsilon}{\mapsto} [S \rightarrow a \bullet Aa] & (i(a)) \\
[S \rightarrow a \bullet Aa] \overset{\epsilon}{\mapsto} [S \rightarrow a \bullet Aa]\ [A \rightarrow \bullet\ a] & (0) \\
[A \rightarrow \bullet\ a] \overset{a}{\mapsto} [A \rightarrow a\ \bullet]_{del} & (0) \\
[A \rightarrow a\ \bullet]_{del} \overset{\epsilon}{\mapsto} [A \rightarrow a\ \bullet] & (0) \\
[S \rightarrow a \bullet Aa]\ [A \rightarrow\ a\ \bullet] \overset{\epsilon}{\mapsto} [S \rightarrow aA \bullet a] & (0) \\
[S \rightarrow aA \bullet a] \overset{b}{\mapsto} [S \rightarrow aAa\ \bullet]_{del} & (s(b,a)) \\
[S \rightarrow aAa\ \bullet]_{del} \overset{b}{\mapsto} [S \rightarrow aAa\ \bullet]_{del} & (d(b)) \\
[S \rightarrow aAa\ \bullet]_{del} \overset{\epsilon}{\mapsto} [S \rightarrow aAa\ \bullet] & (0)
\end{array}
$$

Fig. 7.6. One possible list of transitions that can be applied in order to recognise string *abb* with error correction, in Example 9. The weights of these transitions are given between brackets.

$$
\begin{array}{l}
S \rightarrow a\ A\ a \\
A \rightarrow b\ A\ b \\
A \rightarrow a
\end{array}
$$

One of several ways to recognise the string *abb*, while allowing for error correction, is by application of the list of transitions in Figure 7.6. The total weight of the computation is $i(a) + s(b,a) + d(b)$. There are other computations recognising the same string, which may have lower weight, depending on the values of i, d and s.

In transforming a PDA to become an error-correcting WPDA, new nondeterminism is introduced. By tabulation however, all computations can be simulated in cubic time in the input length, in a way that allows extraction of the computation with the lowest weight [38]. Depending on the chosen parsing strategy, the result may be similar to Earley's algorithm [2, 42], to CYK parsing [67], or to tabular LR parsing [41, 40].

7.10 Further References

Some interpretations of probabilistic formalisms differ from what we have described above in that they define acceptance by *cut-point*. This means that a probabilistic grammar or automaton with probability assignment p is paired with a number c between 0 and 1. The language that is thereby defined consists of all strings w such that $p(w) > c$. For PCFGs this was investigated by [59], and for PPDAs by [28, 24, 62, 23].

Assume a PCFG with probability assignment p. If we let $\lambda_\pi = \log_e p(\pi)$ for each rule π, then the probability of a derivation can be rewritten as:

$$p(d) = \prod_{\pi} p(\pi)^{C(\pi,d)} = \prod_{\pi} e^{\lambda_\pi \cdot C(\pi,d)}. \qquad (7.44)$$

This equation stresses that the probability of a derivation is determined only by the frequencies of individual rules occurring in it. We cannot express, say, preference for combinations or patterns of rules. For this, we need to generalise the framework to *log-linear models* [7, 15]. Such a model allows us to specify a number of arbitrary features $c_1, \ldots, c_m$ on derivations. The features map derivations to non-negative numbers. Further, there is an equal number of weights $\lambda_1, \ldots, \lambda_m$. The model thereby defines a probability distribution on derivations as:

$$p(d) = \frac{1}{z} \prod_i e^{\lambda_i \cdot c_i(d)} = \frac{1}{z} e^{\sum_i \lambda_i \cdot c_i(d)}, \tag{7.45}$$

where z is a normalisation constant, that is, z is the sum of $e^{\sum_i \lambda_i \cdot c_i(d)}$ for all d that are valid left-most derivations.

References

1. S. Abney, D. McAllester, and F. Pereira. Relating probabilistic grammars and automata. In *37th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 542–549, Maryland, USA, 1999.
2. A.V. Aho and T.G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1:305–312, 1972.
3. A.V. Aho and J.D. Ullman. *Parsing*, volume 1 of *The Theory of Parsing, Translation and Compiling*. Prentice-Hall, Englewood Cliffs, N.J., 1972.
4. J.K. Baker. Trainable grammars for speech recognition. In J.J. Wolf and D.H. Klatt, editors, *Speech Communication Papers Presented at the 97th Meeting of the Acoustical Society of America*, pages 547–550, 1979.
5. Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. In Y. Bar-Hillel, editor, *Language and Information: Selected Essays on their Theory and Application*, chapter 9, pages 116–150. Addison-Wesley, Reading, Massachusetts, 1964.
6. L.E. Baum. An inequality and associated maximization technique in statistical estimation of probabilistic functions of a Markov process. *Inequalities*, 3:1–8, 1972.
7. A. Berger, S. Della Pietra, and V. Della Pietra. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22:39–71, 1996.
8. V.D. Blondel and C. Canterini. Undecidable problems for probabilistic automata of fixed dimension. *Theory of Computing Systems*, 36:231–245, 2003.
9. R. Bod, J. Hay, and S. Jannedy, editors. *Probabilistic Linguistics*. MIT Press, 2003.
10. T.L. Booth and R.A. Thompson. Applying probabilistic measures to abstract languages. *IEEE Transactions on Computers*, C-22:442–450, 1973.
11. F. Casacuberta and C. de la Higuera. Computational complexity of problems on probabilistic grammars and transducers. In A. Oliveira, editor, *Grammatical Inference: Algorithms and Applications*, volume 1891 of *Lecture Notes in Artificial Intelligence*, pages 15–24. Springer-Verlag, 2000.
12. E. Charniak. *Statistical Language Learning*. MIT Press, 1993.

13. R. Chaudhuri, S. Pham, and O.N. Garcia. Solution of an open problem on probabilistic grammars. *IEEE Transactions on Computers*, C-32:748–750, 1983.
14. S.F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. In *34th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 310–318, Santa Cruz, California, USA, 1996.
15. Z. Chi. Statistical properties of probabilistic context-free grammars. *Computational Linguistics*, 25:131–160, 1999.
16. Z. Chi and S. Geman. Estimation of probabilistic context-free grammars. *Computational Linguistics*, 24:299–305, 1998.
17. M.V. Chitrao and R. Grishman. Statistical parsing of messages. In *Speech and Natural Language, Proceedings*, pages 263–266, Hidden Valley, Pennsylvania, 1990.
18. M. Collins. Three generative, lexicalised models for statistical parsing. In *35th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 16–23, Madrid, Spain, 1997.
19. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
20. A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Statistical Society Series*, B 39:1–38, 1977.
21. J. Eisner and G. Satta. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *37th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 457–464, Maryland, USA, 1999.
22. K. Etessami and M. Yannakakis. Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. In *22nd International Symposium on Theoretical Aspects of Computer Science*, volume 3404 of *Lecture Notes in Computer Science*, pages 340–352, Stuttgart, Germany, 2005. Springer-Verlag.
23. R. Freivalds. Probabilistic machines can use less running time. In *Proceedings of IFIP Congress 77*, pages 839–842, Toronto, 1977.
24. K.S. Fu and T. Huang. Stochastic grammars and languages. *International Journal of Computer and Information Sciences*, 1:135–170, 1972.
25. J. Goodman. Semiring parsing. *Computational Linguistics*, 25:573–605, 1999.
26. S.L. Graham, M.A. Harrison, and W.L. Ruzzo. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2:415–462, 1980.
27. L. Huang and D. Chiang. Better k-best parsing. In *Proceedings of the Ninth International Workshop on Parsing Technologies*, pages 53–64, Vancouver, British Columbia, Canada, 2005.
28. T. Huang and K.S. Fu. On stochastic context-free languages. *Information Sciences*, 3:201–224, 1971.
29. S.E. Hutchins. Moments of strings and derivation lengths of stochastic context-free grammars. *Information Sciences*, 4:179–191, 1972.
30. F. Jelinek and J.D. Lafferty. Computation of the probability of initial substring generation by stochastic context-free grammars. *Computational Linguistics*, 17:315–323, 1991.
31. F. Jelinek, J.D. Lafferty, and R.L. Mercer. Basic methods of probabilistic context free grammars. In P. Laface and R. De Mori, editors, *Speech Recognition and*

Understanding — Recent Advances, Trends and Applications, pages 345–360. Springer-Verlag, 1992.

32. V.M. Jiménez and A. Marzal. Computation of the n best parse trees for weighted and stochastic context-free grammars. In *Advances in Pattern Recognition*, volume 1876 of *Lecture Notes in Computer Science*, pages 183–192, Alicante, Spain, 2000. Springer-Verlag.
33. M. Johnson. PCFG models of linguistic tree representations. *Computational Linguistics*, 24:613–632, 1998.
34. D. Klein and C.D. Manning. Parsing and hypergraphs. In *Proceedings of the Seventh International Workshop on Parsing Technologies*, Beijing, China, 2001.
35. D.E. Knuth. A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6:1–5, 1977.
36. W. Kuich. Semirings and formal power series: their relevance to formal languages and automata. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Vol. 1*, chapter 9, pages 609–677. Springer, Berlin, 1997.
37. J. Kupiec. Hidden Markov estimation for unrestricted stochastic context-free grammars. In *ICASSP'92*, volume I, pages 177–180, 1992.
38. B. Lang. A generative view of ill-formed input processing. In *ATR Symposium on Basic Research for Telephone Interpretation*, Kyoto, Japan, 1989.
39. K. Lari and S.J. Young. The estimation of stochastic context-free grammars using the Inside-Outside algorithm. *Computer Speech and Language*, 4:35–56, 1990.
40. A. Lavie. An integrated heuristic scheme for partial parse evaluation. In *32nd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 316–318, Las Cruces, New Mexico, USA, 1994.
41. A. Lavie and M. Tomita. GLR* – an efficient noise-skipping parsing algorithm for context free grammars. In *Third International Workshop on Parsing Technologies*, pages 123–134, Tilburg (The Netherlands) and Durbuy (Belgium), 1993.
42. G. Lyon. Syntax-directed least-errors analysis for context-free languages: A practical approach. *Communications of the ACM*, 17:3–14, 1974.
43. C.D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.
44. A. Martelli and U. Montanari. Optimizing decision trees through heuristically guided search. *Communications of the ACM*, 21:1025–1039, 1978.
45. M. Mohri. Statistical natural language processing. In M. Lothaire, editor, *Applied Combinatorics on Words*, chapter 4, pages 210–240. Cambridge University Press, 2005.
46. M.-J. Nederhof. Weighted deductive parsing and Knuth's algorithm. *Computational Linguistics*, 29:135–143, 2003.
47. M.-J. Nederhof, A. Sarkar, and G. Satta. Prefix probabilities for linear indexed grammars. In *Fourth International Workshop on Tree Adjoining Grammars and Related Frameworks*, pages 116–119. Institute for Research in Cognitive Science, University of Pennsylvania, 1998.
48. M.-J. Nederhof, A. Sarkar, and G. Satta. Prefix probabilities from stochastic tree adjoining grammars. In *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics*, volume 2, pages 953–959, Montreal, Quebec, Canada, 1998.

49. M.-J. Nederhof and G. Satta. Probabilistic parsing as intersection. In *8th International Workshop on Parsing Technologies*, pages 137–148, LORIA, Nancy, France, 2003.
50. M.-J. Nederhof and G. Satta. An alternative method of training probabilistic LR parsers. In *42nd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 551–558, Barcelona, Spain, 2004.
51. M.-J. Nederhof and G. Satta. Tabular parsing. In C. Martín-Vide, V. Mitrana, and G. Păun, editors, *Formal Languages and Applications*, pages 529–549. Springer, 2004.
52. M.-J. Nederhof and G. Satta. Estimation of consistent probabilistic context-free grammars. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 343–350, New York, USA, 2006.
53. M.-J. Nederhof and G. Satta. Probabilistic parsing strategies. *Journal of the ACM*, 53:406–436, 2006.
54. L.R. Nielsen, D. Pretolani, and K.A. Andersen. Finding the k shortest hyperpaths using reoptimization. *Operations Research Letters*, 34:155–164, 2006.
55. A. Paz. *Introduction to Probabilistic Automata*. Academic Press, New York, 1971.
56. E. Persoon and K.S. Fu. Sequential classification of strings generated by SCFG's. *International Journal of Computer and Information Sciences*, 4:205–217, 1975.
57. D. Prescher. Inside-outside estimation meets dynamic EM. In *Proceedings of the Seventh International Workshop on Parsing Technologies*, Beijing, China, 2001.
58. P. Resnik. Probabilistic tree-adjoining grammar as a framework for statistical natural language processing. In *Proceedings of the fifteenth International Conference on Computational Linguistics*, volume 2, pages 418–424, Nantes, 1992.
59. A. Salomaa. Probabilistic and weighted grammars. *Information and Control*, 15:529–544, 1969.
60. J.-A. Sánchez and J.-M. Benedí. Consistency of stochastic context-free grammars from probabilistic estimation based on growth transformations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:1052–1055, 1997.
61. E.S. Santos. Probabilistic grammars and automata. *Information and Control*, 21:27–47, 1972.
62. E.S. Santos. Probabilistic pushdown automata. *Journal of Cybernetics*, 6:173–187, 1976.
63. Y. Schabes. Stochastic lexicalized tree-adjoining grammars. In *Proceedings of the fifteenth International Conference on Computational Linguistics*, volume 2, pages 426–432, Nantes, 1992.
64. K. Sima'an. Computational complexity of probabilistic disambiguation. *Grammars*, 5:125–151, 2002.
65. S. Sippu and E. Soisalon-Soininen. *Parsing Theory, Vol. I: Languages and Parsing*, volume 15 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1988.
66. A. Stolcke. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21:167–201, 1995.
67. R. Teitelbaum. Context-free error analysis by evaluation of algebraic power series. In *Conference Record of the Fifth Annual ACM Symposium on Theory of Computing*, pages 196–199, 1973.

68. F. Tendeau. Stochastic parse-tree recognition by a pushdown automaton. In *Fourth International Workshop on Parsing Technologies*, pages 234–249, Prague and Karlovy Vary, Czech Republic, 1995.
69. R.A. Thompson. Determination of probabilistic grammars for functionally specified probability-measure languages. *IEEE Transactions on Computers*, C-23:603–614, 1974.
70. E. Vidal, F. Thollard, C. de la Higuera, F. Casacuberta, and R.C. Carrasco. Probabilistic finite-state machines — part I. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27:1013–1025, 2005.
71. A.J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, IT-13:260–269, 1967.
72. R.A. Wagner and M.J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21:168–173, 1974.
73. C.S. Wetherell. Probabilistic languages: A review and some open questions. *Computing Surveys*, 12:361–379, 1980.

DNA-Based Memories: A Survey

Andrew J. Neel and Max H. Garzon

Department of Computer Science, The University of Memphis
209 Dunn Hall, Memphis, TN 38152 3240
{aneel,mgarzon}@memphis.edu

Summary. DNA-based computers have been made possible by biotechnology developed in the last two decades. They can make advances on challenges caused by limiting features of conventional silicon computers. General and application specific DNA-based computers both require memory systems for DNA computers capable of either sophisticated processing capabilities or the storage of massive amounts of data and, more importantly, effective methods to extract information meaningful to human brains from massive corpora of data. We survey the challenges and methods to build such memories, as well as some applications where they offer very good potential. The DNA memories discussed here do not require an intelligent, outside "brain" to extract the relevant features from given data. Hybridization affinity naturally selects the relevant features in the input and display them on a DNA chip signature as a 2D graphical and semantic representation, by relatively simple parallel procedures that would take forbidding amounts of time to operate on equivalent massive amount of data in digital form by conventional computers.

Keywords: Noncrosshybridizing oligonculeotides, PCR, PCR Selection, DNA-based memories, information retrieval, semantic retrieval, DNA chips, textual entailment.

8.1 Introduction

The original motivation for DNA Computing was the real feasibility, demonstrated by [1], to utilize Deoxyribonucleic Acid (DNA) molecules as data processors capable of solving problems intractable with conventional solutions. The original vision was to ultimately replace conventional computers with biological computers made of DNA. As such, DNA computers would then require input-output protocols, processors, and memory systems to process information for difficult general-purpose computational applications. This vision has substantially evolved in the last decade. Research has expanded from building DNA Computers for general solutions toward the use of DNA as special-purpose computers for specific applications. Here scientists apply lessons learned in biology to solve difficult computational problems by taking advantage of the massively parallel nature of DNA molecules. A survey of the basic biological ideas and biotechnology that made DNA Computing a reality can be found in [15, 21, 22].

Both the original and new visions still require memory systems for DNA computers capable of either sophisticated processing capabilities (such as self-assembly of DNA into useful molecular structures [31, 32], or capable of storing, in principle, large amounts of data (order of terabytes and larger) for information retrieval [2]. In addition to its computational role, Eric Baum [2] suggested that DNA is capable of storing data more compactly than is possible using the best technologies conventional expertise can create. A terabyte of data can, in principle, fit well within a gram of DNA material. The sheer capacity best conventional counterparts of laser media such as CD / DVD which store about 8 Gigs max on 120mm disk, solid state media which stores about the same per cm^2, and magnetic hard disks which stores up to one Terabyte over several 3.5in platters [4]. Given the current state of biotechnology, it seems conceivable that large caches of data can be represented in so-called *DNA-based memories* in small volumes. Further, by taking advantage of massive parallelism naturally occurring in DNA interactions, it may be possible not only to store terabytes of data compactly, but also *mine data* from it in just a few hours. The most striking of possibilities is the potential to apply what is known about DNA computing to create so-called memories capable of retrieving *information* where only data is stored. This idea is best illustrated by a hypothetical memory that would store results of a national survey as *data* but retrieves *information* in the form of line graphs/charts, or answers to complex questions which require aggregation of data or even reasoning about the contents of the memory.

The realization of this vision poses a number of challenges. The critical one is "Can such a memory be created to store even genetic data which is native to the source media?" A second question can be framed as "Can one realistically design a DNA based memory to store and mine a terabyte of data in only a few hours?" Third, "Can it be done with the same degree of accuracy and reliability standard on conventional computers?" Finally, with the potential quantity of data so vast and the memory sizes so minuscule, "How will the results become specifically useful to humans?"

The next section presents two of these challenges (specifically, "How to test DNA memories?" and "How to build DNA memories?"). The following sections continue by summarizing solutions to many of the critical challenges facings DNA memories. The final section concludes by summarizing some of the known applications and some other potential applications of DNA memories.

8.2 Challenges to DNA Memory Implementations

This section introduces several very critical challenges in developing and testing DNA Memories. The first of these challenges is technological in nature and must be met on two fronts: biotechnology and computer technology. Biotechnologies have advanced enough to retrieve and sequence results from DNA memories and can provide a means to create DNA of any desirable species. However, no efficient bio-process or bio-technology standard is readily available to systematically and automatically encode text and data into DNA form. As such, years may be required to bring DNA memories to the commercial world. Computer technologies need to advance in order to enable faster development and lower cost testing for memory protocols. Alternatives to testing *in vitro* provide a platform capable of testing principle implementations of protocols *in silico* but are bound to relatively small memories given computational boundaries of conventional hardware. Until these issues are addressed, researchers will be challenged to constantly replace and update hardware for development and testing purposes. The final analysis is that the technologies are available for DNA-based memories but they are not always efficient. The next section shows how Baum's proposed memory could be implemented with extant biotechnologies and briefly analyzes the efficiency of the technologies that enable implementation.

The largest class of challenges to overcome is centered on the very chemistry that makes DNA-based memories so intriguing. First in this class is the need for input protocols that encode data into DNA such that undesirable hybridizations do not occur. For example, consider the disastrous result of an input protocol that encodes two very different inputs into two very chemically similar DNA structures. The resulting memory would have two data elements capable of hybridizing together and to queries. Similarly, if two queries were encoded as Watson-Crick complements of each other, the retrieval protocol certainly fail to retrieve. Again, consider the input protocol that encodes queries as WC complements of DNA structures that represent very different data in the memory. The retrieval protocol could provide none or very confusing results. This challenge is best expressed as the problem of finding input protocols that store data into DNA reliably to enable reliable retrieval and to prevent data loss or confusion of data.

Still another challenge in this class is to overcome the effects of kinetic forces that negatively influence motion and chemical reactions within the test tube. The kinetic effect on the motion of DNA may prevent queries from reaching a particular target DNA structure in the memory. By analogy, a single person in a dense crowd of people may see his date across the room but not be able to reach her because he is continually buffeted by many a people moving in different directions. Even if two DNA molecules get close enough to hybridize, it may be that the crowd of DNA prevents a desirable fold or bend in the DNA or causes partial hybridization to more than one molecule (e.g. a 3-way knot). The root cause of this problem is too many people (i.e., too much DNA) in too small a room (or test tube holding the DNA memory). As a result, this challenge is to find a concentration ideal for retrieval protocols to operate.

Another class of challenges is found as a result of the capacity potential. Specifically, it may be that so much data is stored that retrieval will eventually fail to return anything useful to humans (typical internet seacrhes comes to mind.) Even with the potential for DNA to act as a computer and create information where data is stored, it is also conceivable such protocols will not scale to the maximum data storage capacity of DNA-based memories. As such, too much data would result in far too little information. The challenge thus remains to invent scalable retrieval protocols and DNA structures that capture exactly the data and information expected by a human querying the system. Even more challenging is to create these structures and protocols to provide *very* informative and *very* specific results. This challenge is best expressed as one of retrieving (getting what humans want) by the semantics of the query constructed from human language that expresses what is desired. More succinctly, this challenge is one of **semantic retrieval**.

The remainder of this article demonstrates solutions to two of these challenges. Section 8.3 tackles the critical challenge of overcoming the time and expense of *in vitro* experimentation, a challenge made even more difficult due to a glass wall that separates the computer scientist, who is interested in DNA memories as solutions to information storage and retrieval problems, and the chemist who has the education and experience to perform experiments *in vitro* and produce actual results. This solution uses a computer software, called *EdnaCo* [14], that is capable of reliable simulation of *in vitro* experiments. Section 8.4 continues with a second critical problem of producing the raw material needed to encode such volumes of data into DNA. The ideal solution to this challenge, known as the encoding solution, is a DNA library that is resistant to self hybridization and whose complement library is also resistant to self-hybridization. The remaining challenges of the capacity potential and kinetic effects are discussed elsewhere [14].

8.3 Virtual Test Tubes

In this section we summarize the experimental tool used to test and benchmark the memory protocols. Full details of virtual test tubes can be found in several sources [14, 13].

8.3.1 Test Tubes in Silico

A virtual test tube (VTT) is "any type of (simulated) biomolecular reactions in electronic media that captures fairly closely the environment and kinetics of the molecular interactions, while making minimal assumptions about the global behavior of molecular populations." (as defined by [14]). *EdnaCo* [14] is a distributed VTT implementation. The computational framework of *EdnaCo* is a complex of interacting data structures distributed over several processing nodes which are interconnected, transparently to the user, in order to produce a single test tube in each run.

The VTT of *EdnaCo* consists of an organized network space of a cellular automaton [16] arranged in grid formation. The nodes (cells) represent quanta of 2D or 3D space that may be empty or occupied by nucleotides, molecules, or other reactants. Each cell can also be characterized by associated parameters that render the tube conditions in a realistic way, such as temperature, salinity, covalent bonds, etc. The entire tube is distributed over a cluster of processors in such a way that each local processor holds only a segment of the entire tube. A segment is itself a copy of Edna, so that one can check the contents of the tube and manage deletions (when a strand leaves the local tube segment) and additions (incoming strands, strand additions at the outset of the simulation, and hybridization events). No strand is split between two different nodes, but their Brownian motion may include migration to any other different node. Further details on the performance and implementation of *EdnaCo* can be found in [14].

Real nucleotides are replaced in *EdnaCo* with virtual ones implemented as C++ objects. Polymers, called strands when implemented in simulation, are represented as complex structural combinations of these nucleotide objects (e.g., linked lists in a software implementation.) Strands carry context information (meta-information, such as position, velocity, direction, etc.), in addition to their internal structure, which may include even morphological information. Strand interaction *in silico* occurs similarly to how it would occur if the interaction were to occur *in vitro*. Two strands encounter each other when they come into close proximity to each other. At this point of the encounter, the tube attempts to hybridize one strand to hybridize to the other, according to some pre-specified criterion for local interactions, for example, some approximation of the Gibbs Energy released by the real molecules. One approximation is the Hamming distance [30] with provides an error count or count of mismatches between two DNA sequences of equal length perfectly lined up to each another. A second approximation is the h-measure developed by [16] that computes the Hamming distance with frame shifts but still assumes the strands are rigid and do not form buldges, in particular. A third approximation is the simplified dynamic programming algorithm of [9].

Each strand is moved about by a motion engine that mimics random-like Brownian motion of the real test tube with actually random motion. The motion engine tracks when any molecule moves beyond the border of a cell. When a border is crossed, the motion model transfers the molecule to the migration engine, which moves it from cell to cell (processor to processor). Motion occurs in discrete steps, called *iterations*. Each iteration corresponds to about 1 ms of real time in the real test tube, roughly equivalent to the time it takes two molecules to settle a hybridization event.

8.3.2 Validation

How can a naturally suspecting biologist or chemist give any credence to the outcome of a simulation or any conclusions based on their analyses? To validate the simulation to suspecting chemists, it is necessary to develop controlled experiments that will quantify the degree of reliability and fidelity of the test tube experiments. For validation, Adleman's experiment was successfully recreated [13] inside a virtual test tube, except at a larger scale that well illustrates the power and scope of the simulations. Random graph configurations [28] were chosen as instances by selecting varying edge densities depending on a fixed probability (0.2, 0.4, and 0.6) of including an edge from the set of possible edges. The size of the graph (number of vertices) varied from 5-9. Each graph was a positive instance, where one witness Hamiltonian path was placed randomly, connecting source to destination. Vertex strands were selected as polymers of 20 bases. Edges were constructed from two vertices by taking the last 10 bases of one vertex and ligating it to first 10 bases of another (but in Watson-Crick complementary form.) The experiment was done under conditions that capture a mildly stringent hybridization criterion where hybridization occurs only if the strands perfectly match in all but two or three places, but were otherwise unconstrained [19], including frame-shifts. The experiment was performed 30 times for 3000 iterations. The results were averaged to provide accurate results. Over 99.4% of nearly 500 total instances of the problem systematically returned the correct answer and solution.

The result of this simulation not only validates the results of the real test tube but also illustrates the power of *EdnaCo's* VTT's to provide very realistic results. Quantifying the success rate by performing the same number of experiments would be very costly in the wet lab where each experiment would cost at least several hundreds of dollars. Here, DNA computers *in silico* provided the solution to Adleman's initial experiment in about 1200 iterations of the simulation. Moreover, [13] were able to show how to improve the efficiency of Adleman's technique (which is essentially brute force and blindly attempts to build all possible paths (most of which will fail to be Hamiltonian) by introducing the concept of a *fitness function*. The fitness functions were later used to improve the efficiency of the simulation and suggest that protocols may exist to achieve similar improvements to Adleman's experiment in vitro. More information on this experiment or the enhancements that derived from this validation can be found in [13]. In [25], a similar second validation, which uses the PCR protocol to *selectively* increase the concentration of *specific* DNA species, is reported. *EdnaCo* has proven itself equally useful and reliable in implementing PCR [20], DNA Chips [27], and Baum's associative memory [2, 24].

8.4 Noncrosshybridizing Bases and PCR Selection

The encoding problem E [15, 16] is defined as the problem of representing, as DNA structures, the data set D in a humanly understood language (such as English), or in the bits of conventional computers (although English will continue to be the running example.) An ideal representation would be **unambiguous** and fully **available** for retrieval. To be considered truly unambiguous, two requirements must be met. First, D must be **encoded unambiguously** to represent English words $W_{English}$ (i.e. every $W_{English}$ in $D_{English}$ must have exactly one counterpart DNA codeword W_{DNA} in D_{DNA} and, likewise, every W_{DNA} in D_{DNA} must have exactly one counterpart $W_{English}$ in $D_{English}$). Second, D_{DNA} must remain **unambiguous over time** and persist through any protocols (e.g. PCR). Encoding solutions are fully available when all strands in D can be queried and retrieved. Of specific concern are encoding solutions that cross-hybridize and thus would create ambiguity in the encoding (because queries may match similar DNA) or may hide information (two memory strands may hybridize and prevent retrieval.) Ideally, every W_{DNA} in D_{DNA} will not hybridize to itself or any other W_{DNA} in D_{DNA} and complementary W_{DNA} in the complementary set of D_{DNA} will not hybridize to itself or any other complementary W_{DNA} in the complementary set of D_{DNA}.

An ideal solution to the encoding problem will produce an ideal data representation efficiently in a manner that is both scalable and reversible. First, encoding solutions are expected to **represent data** with all the characteristics of D_{DNA} expected of an ideal solution (as described in Section 8.2). The encoding process itself is expected to be **reversible** (i.e. a data set $D_{English}$ is encoded as D_{DNA} such that $D_{English}$ can be *reconstructed* exactly from the reverse encoding D_{DNA} to $D_{English}$). The complexity of the encoding algorithm must be low and ideally **linear** to the size of D. For example, $D_{English}$ or D_{Bytes} with n elements should require at most n steps to produce D_{DNA}. The solution E must be **scalable** to encode very large amounts of data and could ideally encode D_{DNA} from any size D).

The solution that is generally considered to be ideal is substitution of words and phrases with a **codeword** that captures the significance of that word or phrase. In DNA memories, a **codeword** is a ssDNA (single-stranded DNA) that unambiguously represents bytes or a language construct (e.g. word, phrase or concept). For example, Adleman hashed graph vertices and edges unambiguously into DNA codewords and demonstrated that his model was feasible *in vitro* by producing a solution to the Hamiltonian Path Problem. This demonstration essentially proved that his representation was fully available over the full length of the experiment and unambiguous in its representation.

This encoding solution requires a single pass over the text to substitute codewords with language constructs that completes in linear time. For example, n words require n substitutions to produce to D_{DNA} and m phases require m substitutions to produce D_{DNA} In more general terms, any set of abstract concepts of size n can be represented in DNA after n substitutions of DNA for concept. By similar substitution of the language constructs with DNA words, the process can be reversed. The encoding solution is scalable to the size of the number of the codewords available.

This solution is common and generally considered adequate. The critical shortcoming of this approach is the lack of available DNA to encode the words of the English language (currently estimated to be around a million by http://www.languagemonitor.com/) or word concepts expressed in WordNet as approximately 207,000 different meanings used worldwide and expressed by different words (http://wordnet.princeton.edu/man/wnstats.7WN). Finding DNA sets of sufficient quantity and quality for even subsets of these databases is the so-called *word design problem.* Its solution has motivated a decade long search for an optimal set of codewords [5, 20, 15, 18, 3, 13, 10, 16]. As pointed out in [20], relatively small DNA strands of 20-mers could easily represent 1 terabyte of data if just one byte corresponded to one 20-mer. If representing abiotic data in the form of words and phrases, or more conceptual ideas in the form of word meanings or word relationships, the potential exists to realize Baum's [2] first estimates of exceeding the capacity of the human brain. Because of the importance of the encoding problem for the entire field, there has been many efforts to find good codeword designs since early days. Surveys can be found in [15, 11, 12, 16]

The ideal approach is to use DNA-based computing to solve the problem because the results is largely independent of Gibbs energy models and likely to yield optimal performance *in vitro.* The resulting PCR Selection (PCRS) [3, 10] protocol was designed to take advantage of PCR's capabilities to select, amplify DNA, and obtain noninteracting (referred to below as *noncrosshybridizing*, for short *nxh*) codewords. The protocol uses PCR to *selectively* amplify DNA in a test tube and then *selectively* separate the amplified DNA from the rest of the test tube. Step one of PCRS is initialized with a seed set of dsDNA D_1 that is placed into a test tube at a low temperature. Each D_1 is initialized as a pair of DNA with universal, unchanging primers attached such that P_1 attached at the $5'$ end and primer P_2' attached at $3'$ end. This protocol begins by heating the test tube to a temperature warm enough to melt *less* all DNA, then quickly cooling it to allow *more* complementary DNA to re-hybridized. The protocol continues by amplifying the melted ssDNA (the more nxh DNA in the test tube) by PCR. Amplification [23] of the nxh DNA only occurs because the primers, inserted to initiate PCR extension, will not hybridize to the dsDNA (the more stable and more complementary DNA in the test tube.)

The capability of PCRS to identify nxh subsets was evaluated experimentally in [6, 7]. The validation began with an initial DNA set of DNA seeded with the full set of random 20-mers. The primers, P_1 and P_2, were nxh 20-mers and were excluded from the seed set. Figure 1.1 in cite14 shows the templates (red) in the top row (centered above each gel) with each primer (black) attached at the 5′ and 3′ ends. The first template was fully complementary while the last was nxh. Two other templates were selected as intermediate steps between fully complementary and nxh. The template sequences were designed using an *in silico* software tool [10] that selects nxh DNA from an initially random pool.

In a first round of experiments, PCR was performed on each of the four templates representative of the spectrum of conditions between fully crosshybridizing and nxh extremes. The test tube was heated to 52^{o}C, 58^{o}C, 64^{o}C, 70^{o}C, and 74^{o}C (centigrade) and PCR extension was allowed to run for 1 round that lasted one hour. Each template was incubated in a PCR buffer of 50 mM KCI, 10 mM Tris-HCI, 0.1% Triton X-100, 2.5mM MgCl_2, 0.4 nM 4 dNTP, and 4 U Taq DNA polymerase in total 10ul volume. The results of the 20 experiments (5 temperatures for each of the four species) were then placed into a denature gel at 400V. Denaturing was allowed to run for one hour before being captured by autoradiography. In the resulting gels, amplification occurred at 52^{o}C, 58^{o}C, and 64^{o}C for all species. At temperatures at or above 70^{o}C, very little amplification occurred for the three nearly crosshybridizing templates. However, amplification for the nxh DNA templates occurred at all temperatures with maximum yield at 52^{o}C. This result proves that PCRS can selectively amplify nxh DNA from a seed set and eventually extract a maximal subset.

This experiment was repeated a second time to determine the ideal conditions for nxh amplification. In this experiment, only the maximally similar and maximally dissimilar templates were used. The range of temperatures included 37^{o}C, 40^{o}C, 43^{o}C, 46^{o}C, 48^{o}C, 50^{o}C, 56^{o}C, 62^{o}C, 68^{o}C, and 72^{o}C. It was determined that no amplification occurs at 43^{o}C when the templates are complementary. However, plenty of amplification occurs when the temperature was 43^{o}C and below. This range of temperatures allows PCRS to operate efficiently.

The results of PCRS were evaluated and the nxh quality of each set was confirmed to be very high in [7]. PCRS was again performed to produce a set of template species. The nxh quality was then evaluated by spectrophotometric quantification, a method that measures optical density by spectrography. The critical property being exploited is that UV light absorption at 260nm is less for ssDNA than for dsDNA. By melting the DNA results fully, a spectrophotometer can measure the amount of light absorbed by the ssDNA. Thus, a rough census of nxh to crosshybridizing DNA can be taken over time. As the test tube cools, hybridization will occur naturally only if the DNA can form energetically stable bonds. The measurement of this concentration of dsDNA over time curve is called the *CoT curve.* (for *C*oncentration-*T*ime.) A steep decline in the CoT indicates the DNA is very crosshybridzing while a flatter CoT curve indicates the DNA is more nxh.

PCRS is ideal for creating nxh sets. Because PCRS is massively parallel in nature, it is maximally efficient *in vitro*. As a result, the time of completion for a single round of PCRS is in the order of minutes to hours. The product set is maximally nxh according to the natural process of DNA hybridization and not an approximation of hybridization applied in other approaches. The characterization of several product sets has been given in [7]. The remaining challenge thus becomes to discover a scalable solution that can extract the codeword sequences from these nxh product sets. Even if one is willing to bear the cost, in terms of money, of sequencing a sample of the code set *in vitro*, it remains forbiddingly infeasible in terms of time to sequence the whole product set of 20-mers, estimated to be 10,000 - 50,000 species [6].

In order to get useful information on the composition of such nxh set, PCRS has been implemented *in silico* with *EdnaCo*. In simulation, a model is required to of decide whether two strands hybridize. An approximation of Gibbs energy that has proved to provide very good results up to 60-mers can be used [8]. Scalability is still a challenge due to the sheer size of input sets and the number of rounds needed to refine the codeword set. The advantage is that the codeword set will be fully sequenced and of comparable quality. We will discuss this solution in full details below.

In [20], PCR selection (PCRS) was implemented *in silico* on *EdnaCo* to identify quality codeword sets. PCRS protocol was implemented exactly as it would be *in vitro* with one exception: all copies of a codeword were removed from the virtual test tube at the moment *any* copy of the codeword formed a duplex according to Gibbs energy. This variation significantly reduced the overhead and helped the apparent kinetic problems that emerged *in vitro* required to converge to an ideal nxh codeword set.

The codeword product sets were evaluated by both size and quality. Quality was evaluated by three methods to assess the application of each set *in vitro*. The first method performed an $O(n^2)$ search of $R = \{\Sigma r\}$ to determine if any r would hybridize to any member of R and if any r' (the complement of r) would hybridize to any member of R' (the complement set of R). Because Gibbs energy evaluation is the primary criterion for hybridization *in silico* and is an excellent predictor of hybridization *in vitro*, this first method is an excellent measure of the nxh quality of R. The second method estimates the Gibbs energy of each pair of codewords. The third method characterizes each r in R as blocks of 2, 3, and 4-mers.

PCRS was performed with various input sets of uniform length n where $n = 5, 6, 7, 8, 10, 13, 16,$ and 20 [20]. No matter how efficiently represented in simulation, a seed set of length 20 contains at most 4^{20} strands. The resources needed to perform PCRS with that much input are too high a computational and memory requirement even for supercomputers of today. This challenge motivated a pre-processing filtering that reduces the complexity and search space of the input DNA. The inputs to PCRS, called S, were filtered to minimize wasting simulation cycles. Each S was filtered to remove *a priori* those obvious candidate DNA species likely to be removed by PCR Selection. In every full n-mer DNA set, a sizable number of them is immediately available to be discarded as they are palindromes or contain hairpins (i.e., reverse WC complements of self or other strands in the set), contain k-mer runs ($k > 3$) of

G/Cs. The above filters were implemented on a full seed input of all n-mers. Where it was feasible, the input set was scanned to remove any hybridizing DNA that would crosshybridize even with very stringent hybridization thresholds according to the h-distance [17] which is less accurate but executes faster than the Gibbs approximation used by *EdnaCo*. Because scanning for crosshybridization was only possible over several months for 12-mers and 13-mers, a different pre-processing step was used for $n > 13$. The serialized filter was replaced by a distributed filter to do the work to PCRS running on *EdnaCo* (a parallel application). By removing these strands from the input sets, were able to reduce the complexity of these problems by orders of magnitude [20]. As shown in Figure 8.1, the net gain is a filtered set (middle bars) that is at least one order of magnitude smaller in size than the full set (left bars). The size of n-mer set seems to grow exponentially with a power law $4^{0.72n+1.66}$.

The results and capabilities of PCRS to scale and produce large sets of nxh were first discussed in [20]. Below is a recap of the evidence presented in that paper that proves PCRS as a scalable solution capable of produce fully sequenced nxh codewords efficiently in a matter of hours for small examples and months for very large codeword sets. The next section addresses the issue of scalability and is followed by two evaluations of the nxh quality of the codeword product sets.

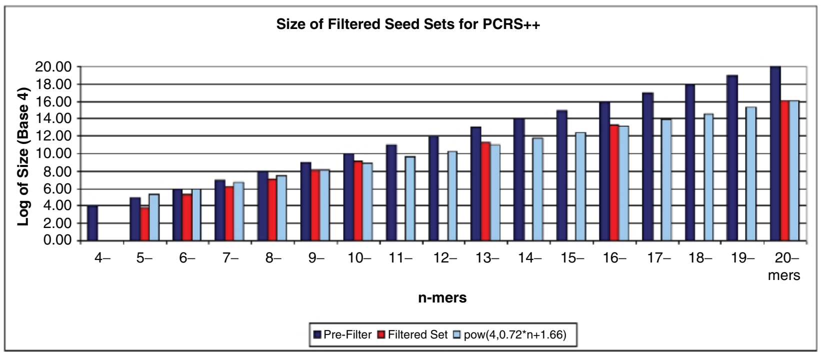


Fig. 8.1. Pre-filtering of the space of n-mers reduces the full set by at least an order of magnitude. The filtering process removes from the full input set (left bars) the DNA species that contain hairpins or short runs of the same base and those species that are palindromes or are too much like other strands already passed by the filter. The size of the resulting filtered set (middle bars) can be estimated by the power law $4^{0.72n+1.66}$ (right bars). [20]

Figure 8.2 compares the size of filtered n-mer input set (dark blue) is compared with the PCRS product set (red) and an ideal PCRS product set (black). Due to the sheer complexity of the task, the experiments for 20-mers were not completed after 10 months into the simulation. The growth in size for n-mer sets in range from 5-10 was calculated to be $4.19^{sqrt((n)+0.3)}$. By extrapolation, the sizes of 20-mer nxh sets are estimated to be around 4^{16} or 4.7 billion. The set of 16-mers began with $4^{13.23474901}$ or about 93 million DNA polymers. PCRS produced a set of about 4^8, which was reduced to about 47 or `16K`DNA polymers by additional exhaustive filtering.

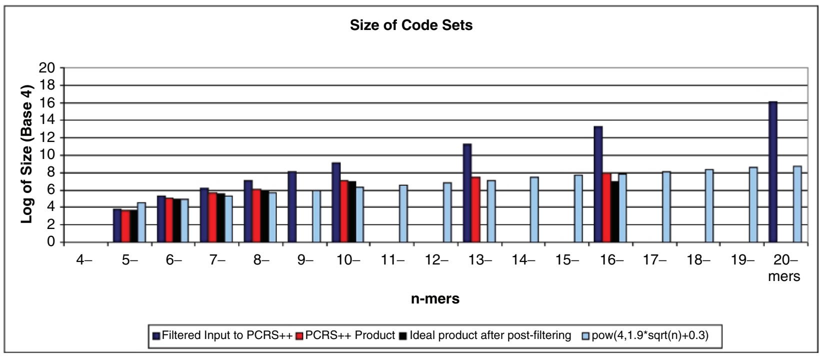


Fig. 8.2. Size of the PCR Selection product obtained from the filtered input sets (left bars). The size of the resulting set (middle bars) can be estimated by the sub-power law $4^{1.9pn+0.6}$ (light bars on the right). [20]

[20] obtained very large sets of very good nxh quality that are guaranteed to perform well under a variety of reaction conditions in wet test tubes. The method is constructive, not in the sense that it selects the actual wet strands as the PCR Selection protocol does *in vitro* [10], but in the complementary and more advantageous sense that the sequence and composition of the actual codewords is known, thereby bypassing a costly or impossible sequencing procedure. The method is also nearly optimal, in the sense that the size of the codeset obtained is in the order of magnitude of the theoretical maximum size of a set for the given hybridization stringency under which the simulation was conducted.

The net result is that there is in hand a number of codeword sets of n-mers for values of n = 4-10, 13-,16- and 20mers. More importantly, we have the protocol that is capable of producing far larger codeword sets. The analysis for some of these sets has been presented, for example, G/C-content analysis and a bias analysis of the most frequent k-mer-blocks for k = 2, 3, 4. These results suggest further analyses that could be conducted with the products of the protocol *in vitro* to obtain further characterization of its products (so far only conducted in vitro for 20-mers [7]). These PCRS experiments (including the analysis) have lasted over 10 months and are expected to bear full fruit up to 20-mers (once resources become available). PCRS can easily be scaled to sets of longer polymers with feasible (but long) run times up to about 60-mers when the Gibbs energy model used begins to fail. At that point, optimality will be out of range. Given the very encouraging results of PCRS [20], tensor products [15], and shuffle codes [29], it may be necessary to apply tensor product and shuffle codes as filters to PCRS as the size of the n-mers increase much beyond 20-mers.

[20] argued that the size of these codeword sets may be in the order of magnitude of maximal sets and provided a good practical estimate of the capacity of a DNA memory [20] based on Baum's construction [2]. With these codes, we have the support required to implement virtually any kind of application of biomolecular computing, and in particular, the primary ingredient for DNA memories [25, 18, 25, 33].

8.5 Some Applications of DNA Memories

In this Section, we summarize a few applications of DNA memories capable of retrieving both biotic and abiotic data under development or consideration.

Given a nxh basis B of N oligos of length n, arbitrary data sets can be represented by a so-called *signature.* [18, 5] define the representation as a *signature* as follows. Without loss of generality, it will be assumed that strings x to be encoded are written in a four letter alphabet {a,c,g,t}. The signature of a string x with respect to B is a vector V of size N obtained by shredding. x to fragments of size n or less and allowing the fragments to hybridize under saturation conditions to the oligos in B. The vector V can be visualized as a 2D matrix by arranging the basis strands in a 2D DNA-chip, with a fixed common number of basis strand per spot. The vector V may appear not to be well-defined, since it is clear that its calculation depends on the concentration of basis as well a strands x used in an experiment to compute it. To avoid these difficulties, this situation is idealized to the case where only the same number of copies (say, a fixed large number of copies) of each basis strand is present in the tube, that the target input x is poured in saturation concentrations over the chip, and that the tube is small enough that all possible hybridization occur within reasonable time. These idealizations are supported by a number experiments performed on *EdnaCo* that essentially normalize the representation and make it unique to within small variation in intensity for each pixel, but with basically the same pattern over the entire chip. Precise details can be found in [18]

An obvious and very intriguing application of the type of DNA memories described above is in storage and retrieval of genomic data of biological organisms. One cannot help but wonder what type of signatures given organisms may produce and how they compare to one another. These issues have been explored in several cases, of which we mention three, two with genomic data and one with abiotic/textual data. Precise details can be found in [20, 26, 27, 25]

In the first application, a number of plasmids of lengths varying between 3Kbp (kilobase pairs) and 4.5Kbps were shredded to fragments of size 40 or less and thrown into a tube containing a basis B of three different nxh quality consisting of 36-mers. The each member of the basis B was permanently fixed to a spot on a thin piece of glass. Their signature were approximated by simulation on *EdnaCo* by a simple process described above, i.e. measuring the degree in which the shredded target DNA interacted with the DNA on the chip. The variability of the signatures was telling of the nature and origin of the plasmids. More nxh bases produced signatures with less variability and noise. As the noise decreased, the size of the basis could be reduced so that the same result could be achieved with far fewer nxh DNA and far smaller DNA chips. On a high-quality nxh basis, the plasmid signatures appeared were just as different as the organisms that originated them. This application shows how genetic data could be captured by merely measuring hybridization affinity to an nxh set of DNA. In a second application, the DNA chip can reveal emergent information from a set of shredded DNA that the shredded DNA itself became the DNA memory. The shredded DNA is placed into a test tube. PCR selection protocol is applied to identify an nxh subset, and the resulting species become the DNA memory placed onto a DNA chip. When the shredded DNA is allowed to interact with the DNA chip, the resulting signature is far sharper and crisper. By performing this process on many such species, it is possible to create a single, re-usable DNA chip capable of identifying the origin species of any DNA sample. Precise details can be found in [20].

Furthermore, this same protocol can be extended to very different data sets. For example, the shredded DNA may be replaced with English words from a Shakespearian play encoded in DNA structures. PCR selection protocol could then produce a meaning subset of words (arguably the most meaningful subset of words) which could then be placed on a DNA chip. The application of this chip could assist in natural language processing problems of question answering and information summarization. Precise details of this protocol can be found [26, 27] (for genetic data) and [25] (for abiotic data).

In the third application, DNA memories are used to reveal the meaning of human language in applications such as recognizing textual entailment (RTE). Here, the capacity potential of DNA to retrieve fine-grained information from vast amounts of data could be truly leveraged. Data representing the full complement of word-phrase meanings is stored into a DNA memory. This potential has been documented in the case of textual entailment challenge (RTE), A typical instance of RTE consists of two paragraphs (one called the *text* and the other called the *hypothesis*). The problem is to decide whether the hypothesis is entailed by the text. This is not a problem about logical inference, but rather a problem that requires bringing into play background knowledge about the world, usually left implicit in the text. Using DNA memory representations of semantic networks such as WordNet, entailment can be solved automatically by disambiguating the words and phrases of each paragraph to meanings in the DNA memory and then examining the overlap of the corresponding semantic concepts contained in both text and hypothesis. This application shows how DNA memories might be able to capture semantical knowledge in superior ways to that standard syntactic and lexical representations. More details can be found in [25, 27].

In conclusion, we mention an important advantage on this type of DNA memory. Information retrieval normally requires the identification of the relevant data, retrieval of the data from the DNA-based memory, and further processing of the desired information. The critical issue is how to "identify the correct data to retrieve". The simplest and most frequent conventional approach is to "let the human decide" which data to retrieve. For example, conventional computers ask the user to search for information by opening a series of files on a file system, reading each file, and deciding from the content which file is correct. The underlying system may provide tools to help the human search by iterating through a list of files searching for keywords. This "key-word" search approach will miss files and content that express similar concepts with words different than the keywords. Ideally, data retrieval tools could not only retrieve lexically (by matching key words); but also retrieved by matching *meanings* of words captured in the query to *meanings* of words captured in the data in the memory. By contrast, the DNA memories discussed here do not require an intelligent, outside "brain" to extract the relevant features from the given data. Hybridization affinity naturally selects the relevant features in the input and display them on the DNA chip signature as a 2D graphical and semantic representation.

References

1. L.M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021, 1994.
2. E. Baum. Building an associative memory vastly larger than the brain. *Science*, 268:583–585, 1995.
3. H. Bi, J. Chen, R. Deaton, M. Garzon, H. Rubin, and D. Wood. A PCR based protocol for in vitro selection of non-crosshybridizing oligonucleotides. *Journal of Natural Computing*, 2(4):461–477, 2003.
4. BES Computers Blog. Hitachi 1tb hard drive scores popular mechanics editors choice award. http://blog.bescomputers.net/2007/01/20/hitachi-1tb-hard-drive-scores-popular-mechanics-editors-choice-award-ces/, 2007.
5. K. Bobba, A. Neel, V. Phan, and M. Garzon. Reasoning and talking DNA: Can DNA understand English? In *Proceedings of the 12th International Meeting on DNA Computing*. Springer.
6. J. Chen, R. Deaton, M. Garzon, J.W. Kim, D.H. Wood, H. Bi, D. Carpenter, J.S. Lee, and Y.Z. Wang. Sequence complexity of large libraries of DNA oligonucleotides. In N. Pierce and A. Carbone, editors, *DNA Computing: Preliminary Proceedings of the 11th International Workshop on DNA-Based Computers.*
7. J. Chen, R. Deaton, M. Garzon, J.W. Kim, D.H. Wood, H. Bi, D. Carpenter, and Y.Z. Wang. Characterization of noncrosshybridizing DNA oligonucleotides manufactured in vitro. *Journal of Natural Computing*, 1567.
8. R. Deaton, J. Chen, H. Bi, M. Garzon, H. Rubin, and D.H. Wood. A PCR-based protocol for in vitro selection of non-crosshybridzing oligonucleotides. In M. Hagiya and A. Ohuchi, editors, *DNA Computing VIII.*

9. R. Deaton, R. Murphy, M. Garzon, D. Franceschetti, and Jr.S. Stevens. Good encodings for DNA-based solutions to combinatorial problems. In *Proceeedings of the Second Annual DIMACS Workshop on DNA Based Computers*, 1996.
10. R.J. Deaton, J. Chen, H. Bi, and J.A. Rose. A software tool for generating non-crosshybridizing libraries of DNA oligonucleotides. In M. Hagiya and A. Ohuchi, editors, *Proceedings of the 8th International Meeting of on DNA-Based Computers*.
11. R.M. Dirks, M. Lin, E. Winfree, and A.N. Pierce. Paradigms for nucleic acid design. *Nucleic Acids Res.*, 32:1392–1403, 2004.
12. U. Feldkamp and H. Ruahue W. Banzhaf. Software tools for sequence design. *Genetic Programming and Evolvable Machines*, 4:153–171, 2003.
13. M. Garzon, D. Blain, K. Bobba, A. Neel, and M. West. Self-assembly of DNA-like structures in silico. *Journal of Genetic Programming and Evolvable Machines*, 4:185–200, 2003.
14. M. Garzon, D. Blain, and A. Neel. Virtual test tubes. *Journal of Natural Computing*, 3(4):461–477, 2004.
15. M. Garzon and R. Deaton. Codeword design and information encoding in DNA ensembles. *Journal of Natural Computing*, 3(33):253–292, 2004.
16. M. Garzon, R. Deaton, P. Neathery, D.R. Franceschetti, and R.C. Murphy. A new metric for DNA computing. In *Proceedings of the 2nd Genetic Programming Conference*. Morgan Kaufman.
17. M. Garzon, R. Deaton, P. Neathery, R.C. Murphy, D.R. Franceschetti, and Jr.E. Stevens. On the encoding problem for DNA computing. In *Preliminary Proceedings of the Third DIMACS Workshop on DNA-based Computing*, pages 230–237, University of Pennsylvania, 1997.
18. M. Garzon, A. Neel, and K. Bobba. Efficiency and reliability of semantic retrieval in DNA-based memories. In *Proceedings of DNA Based Computing 2003*. Spinger.
19. M. Garzon and C. Oehmen. Biomolecular computation on virtual test tubes. In N. Jonoska and N. Seeman, editors, *Proceedings of the 7th International Workshop on DNA-based Computers, 2001*.
20. M. Garzon, V. Phan, S. Roy, and A. Neel. In search of optimal codes for DNA computing. In *Proceedings of DNA Computing, 12th International Meeting on DNA Computing*. Springer.
21. M.H. Garzon and R.J. Deaton. Biomolecular computing and programming: A definition. *Kunstliche Intelligenz*, 1.
22. M.H. Garzon and R.J. Deaton. Biomolecular computing and programming. *IEEE Transactions on Evolutionary Computation*, 3(3):36–50, 1999.
23. K.B. Mullis. The unusual origin of the polymerase chain reaction. *Scientific American*, 262(4):56–61, 64–65, 2001.
24. A. Neel and M. Garzon. Efficiency and reliability of genomic information storage and retrieval in DNA-based memories with compaction. In *IEEE Congress for Evolutionary Computation 2003*, pages 2733–2739, 2003.
25. A. Neel and M. Garzon. Semantic retrieval in DNA memories with gibbs energy models. *Biotechnology Progress*, 22(1):86–90, 2006.
26. A. Neel, M. Garzon, and P. Penumatsa. Improving the quality of semantic retrieval in DNA-based memories with learning. In *Knowledge-Based Intelligent Information & Engineering Systems. KES-2004*. Springer.

27. A. Neel, M. Garzon, and P. Penumatsa. Soundness and quality of semantic retrieval in DNA based memories with abiotic data. In G. Greenwood and G.Fogel, editors, *Proceedings of the IEEE Conference on Evolutionary Computation, CEC 2004*.
28. E.M. Palmer. *Graphical Evolution*. Wiley, New York, 1985.
29. V. Phan and M. Garzon. Information encoding using DNA. In *Proceedings of the 10th International Conference on DNA Computing*. Springer.
30. J. Roman. *The Theory of Error-Correcting Codes*. Springer, Berlin, 1995.
31. N.C. Seeman. DNA engineering and its application to nanotechnology. *Trends in Biotechnology*, 17:437–443, 1999.
32. E. Winfree, F. Liu, L.A. Wenzler, and N.C. Seeman. Design and self-assembly of two dimensional DNA crystals. *Nature*, 394:539–544, 1998.
33. B. Yurke and A.P. Mills. Using DNA to power nanostructures. *Genetic Programming and Evolvable Machines*, 4:111–112, 2003.