Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering

Two-dimensional Pattern Matching Using Automata Approach

by

Jan Žďárek

A thesis submitted to
the Faculty of Electrical Engineering, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

February 2010

Thesis Supervisor:

Prof. Ing. Bořivoj Melichar, DrSc.
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo náměstí 13
121 35 Praha 2
Czech Republic

Abstract and contributions

This work focuses on using finite and pushdown automata for multi-dimensional pattern matching. All methods are presented in detail for the two-dimensional case. Presented algorithms use the idea of a dimensional reduction and thus can be extended to solve pattern matching problems of more than two dimensions.

First contribution of this work takes inspiration in results concerning formal description of all known one-dimensional pattern matching problems and their corresponding finite automata for pattern matching. The generic algorithm of two-dimensional pattern matching using finite automata is presented. This algorithm is then used for construction of particular exact and approximate pattern matching algorithms using finite automata. Each step of particular two-dimensional pattern matching algorithm involves different types of finite automata for pattern matching. Types of finite automata used for preprocessing and pattern matching steps of the two-dimensional pattern matching algorithm are presented.

For a two-dimensional text of size $(n \times n)$ and a two-dimensional pattern of size $(m \times m)$ the exact pattern matching version of the generic algorithm works in linear time with respect to the size of the two-dimensional text, that is $\mathcal{O}(n^2)$. This is a variant of the Baker and Bird result, using solely finite automata. As an improvement of the Baker and Bird result, we present an implementation with significantly lower space complexity, $\mathcal{O}\left(m^2 + n\right)$. The two-dimensional approximate pattern matching version of the generic algorithm uses the two-dimensional Hamming distance, an analogy to the distance used in one-dimensional pattern matching. Implementation of this algorithm is independent of the number of errors allowed and works in linear time, not counting the required determinisation of involved finite automata. For this method we developed a space-efficient version, too, $\mathcal{O}\left(nm^2\right)$.

The second contribution of this work has multiple sources of inspiration. These are the pushdown automata for tree pattern matching and, specifically, their usage for subject tree indexing, allowing to perform the matching in time proportional to the size of the tree pattern and not the tree itself. This approach is equivalent to the one-dimensional text indexing. Another source of inspiration are particular methods in bio-informatics, namely representation of genetical relations among species in form of phylogenetical trees. There are certain situations, where additional relations among genomes of species prevent to construct the graph of relations in the form of the tree. The additional relations, represented as additional transitions in the graph, turn the tree into the directed acyclic graph.

These points led us to a new approach to the two-dimensional pattern matching, specifically to the two-dimensional text indexing. We present the transformation of two-dimensional structures into the form of a tree, preserving the context of each element of the structure. The tree can be linearised using the prefix notation into the form of one-dimensional text (string) and we do the pattern matching in this text. In this part of our work, pushdown automata indexing the two-dimensional text are presented. They allow to search for two-dimensional prefixes, suffixes, or factors of the two-dimensional text in time proportional to the size of the two-dimensional pattern representation. The two-dimensional pattern matching is then performed in time proportional to the length of the prefix notation of the tree, representing the two-dimensional pattern. This result achieves the properties analogous to the results obtained in tree pattern matching and one-dimensional text indexing. Although the two-dimensional prefix pushdown automaton is deterministic, there is still an open question how to construct deterministic versions of two-dimensional suffix and factor pushdown automata.

Keywords:
Two-dimensional exact pattern matching; two-dimensional approximate pattern matching; two-dimensional index; pattern matching automaton; classification of pattern matching automata; simulation of finite automata; dynamic programming; edit distance; two-dimensional Hamming distance; pushdown automaton; prefix notation; tree; tree pattern matching.

Acknowledgements

During the time I worked on this thesis, a number of people were very important in providing guidance, support and encouragement to me. The first one of them is my thesis supervisor, Prof. Bořivoj Melichar. I am indebted to him for everything. He deserves my greatest thanks. It was many years ago during my Master studies when we have discussed the *LZ77* and *LZ78* algorithms and the style of presentation the original papers are written in. Looking back, it was a natural step that he became the supervisor of my master's thesis shortly after our first discussion. And, ultimately, he offered me the opportunity to become his doctoral student. Listed below is (among others) the grant project he offered me to collaborate with him on. The project provided both the unique opportunity to address interesting topics as well as some financial support. Further, he invited me to participate on *Advanced Technology Higher Education Network/SOCRATES* (*ATHENS* courses) of European Union, where our research topics were presented. These courses for students of the selected top European technical universities provide very interesting feedback.

I am unable to count the number of discussions we held, the number of drafts he patiently marked up in red, or the number of ideas and recommendations he gave me. His comments, corrections and suggestions to numerous drafts of the thesis improved it significantly. Maybe even more importantly, his patient encouragement, leadership and clear vision allowed and forced me to ever this work. He provides a feeling of freedom to his students as a supervisor and to his staff as a manager. He asserts his will by discussion rather than by force. Many could learn from him. Moreover, he introduced me into the art of grant writing, a priceless skill. Thank you, Bořek.

Prof. Jan Holub gets my thanks for official support during the final months of work on this thesis and for providing financial support from some of the grants and projects listed below. However, he offered me more than material support. The offer to be the co-editor of several conference proceedings, including the Proceedings of the Conference on Implementation and Application of Automata 2007, was a challenge and honour. Further, I really enjoyed our (rather infrequent) scientific disputes as well as our (more frequent) altercations over correct TEXing of things. Inspiration in his work, his attitude and encouragement, provided a great help to me.

During the years of my doctoral studies, I led bachelor and master theses of several students who helped with the practical evaluation of the algorithms and explored many dead ends. In alphabetical order these students were: Petr Chytil, Marie Kalbáčová, Vlastimil Menšík and Jakub Šlesarík.

My friends, members of our research group the Prague Stringology Club, deserve my gratitude. I have already acknowledged Prof. Bořivoj Melichar and Prof. Jan Holub. That does not lessen the merits of the other members. They have provided me with numerous suggestions, comments and valuable spiritual support. In alphabetical order they are: Jan Antoš, Miroslav Balík, Martin Bloch, Tomáš Flouri, Jan Janoušek, Jan Lahoda, Zdeněk Troníček, Jan Šupol, Michal Voráček and Ladislav Vagner.

Among them, very special thanks deserves Jan Lahoda. His suggestions and comments to numerous drafts of this thesis improved it significantly. He provided a different point of view on the topics presented. It was also a great pleasure to cooperate with him in developing his ideas in the field of compressed pattern matching. I have learned much.

I would like to thank all heads of the Department of Computer Science and Engineering I worked for and the whole staff of the department for such an inspiring environment and for

the possibility to use the departmental facilities.

I am very grateful to Prof. Maxime Crochemore for his time and support while I was at *le Laboratoire d'informatique Gaspard-Monge, Université de Marne-la-Vallée*. I would also like to thank the head of the department and all his staff for allowing me to work there, as well as the Socrates program of the European Union and *la Caisse nationale des Allocations Familiales* for their financial support.

I wish to thank the following for the financial support: the Ministry of Education, Youth, and Sports of the Czech Republic (MŠMT/FRVŠ), the Czech Science Foundation (GAČR), and Czech Technical University in Prague (ČVUT). I also wish to express my gratitude to all of the heads of the Department of Computer Science and Engineering who provided me with funding from the research program of Ministry of Education, Youth, and Sports of the Czech Republic.

Projects I participated in or I led, in chronological ordering:
GAČR grant No. 201/01/1433 – "Pattern matching in text",
ČVUT grant No. 0409413 – "Two-dimensional pattern matching using finite automata",
FRVŠ grant No. 2060/2004 – "Two-dimensional pattern matching",
MŠMT research program J04/98:212300014 – "Research in the area of information technologies and communications",
MŠMT research program MSM6840770014 – "Research in the area of the prospective information and navigation technologies",
GAČR grant No. 201/01/0807 – "Pattern matching in text and trees".

❧

Long were my studies, and long are the Acknowledgements. Over the years, I have realised the truth in the statement that the world we live in is composed of more than just work and research. Family is and always has been a safe harbour from the world's turbulences. I feel these last words shall belong to our family.

I am grateful to my brother Václav: despite working in a quite different branch of studies, your achievements are a continuing source of motivation for me.

Míšenko, you doubted you would understand any sentence of this work. Bearing this in mind, the following Czech sentence expresses much more than its literal meaning: *děkuji za lásku, podporu a trpělivost s mojí prací.*

Final thanks are reserved for my parents Jan and Ludmila, without whom and their unceasing support and care I would never have reached this point. I am deeply grateful that you have always supported me in my decisions and provided me with real parental guidance and a model of life. As a small repayment for all I owe you, I dedicate my work to you.

To my parents

Contents

Introduction

1 Introduction

1.1 Motivation

PATTERN matching algorithms can be found in almost all fields of informatics. They represent one of the main data manipulation operation, comparison, which can be found everywhere. The pattern matching problem in strings, where it is to find all exact or approximate occurrences of pattern P of length m in text T of length n, is a classical problem of informatics studied for a long years [Gal85].

Although data are usually stored in linear files, i.e. as strings, their interpretation can be very heterogeneous. In many situations arising in digital data processing we encounter strings of symbols organized into multidimensional structures, especially two-dimensional. These structures are in fact very common, almost every computer user have encountered examples of such structures: pictures. The usage of multidimensional structures is widespread, multidimensional pattern matching is applied in various fields ranging from image recognition to computational biology and medicine.

1.2 Related work

The first linear-time (on fixed alphabets) two-dimensional exact pattern matching algorithm has been found independently by Bird [Bir77] and Baker [Bak78]. The first attempt to solve the two-dimensional approximate pattern matching is due to Krithivasan and Sitalakshmi [KS87] but their solution considers presence of errors along one dimension only. Since then, many papers describing various methods of the two-dimensional exact and approximate pattern matching have appeared. More results related to our approach will be mentioned later.

The classification of one-dimensional pattern matching problems is due to Melichar and Holub [MH97], finite automata based solutions to these problems are presented in several works, the largest part e.g. in [MHP05].

Suffix and factor automata for one-dimensional text indexing originated in several works by Blumer *et al.* and Crochemore [BBE$^+$83, BBE$^+$85, Cro86]. Other indexing tools, such are suffix trees and suffix arrays are due to Weiner [Wei73], McCreight [McC76], Ukkonen [Ukk95], Farach [Far97], Manber and Myers [MM93].

There are different tree algorithms due to many authors, their survey can be found for example in Comon *et al.* [CDG$^+$07] and in Cleophas [Cle08]. Tree pattern matching using pushdown automata is due to Janoušek and Melichar [JM09, Jan09], alternative construction is due to Flouri, Janoušek and Melichar [FJM09].

The first indexing data structure for two dimensional arrays called a PAT-tree has been proposed by Gonnet [Gon88]. (The name PAT-tree refers to Morrison [Mor68] and his one-dimensional structure.) Another variant has appeared in Amir and Farach [AF92]. Since then, two-dimensional suffix trees were addressed by Giancarlo and Grossi [GG97b], Kim, Kim and Park [KKP03], and Na, Giancarlo and Park [NGP07].

1.3 Problem statement

Our aim is to find out ways how to reuse the results concerning one-dimensional pattern matching using finite automata in the pattern matching in multiple dimensions and show some examples of this approach.

Next, we want to explore the possibilities of application of tree pattern matching algorithms based on pushdown automata to the two-dimensional pattern matching. We want to present a way how a two-dimensional text can be indexed using the linearised tree representation in the form of pushdown automata.

1.4 Contributions of the thesis

This work has the following main contributions.

The first contribution of this work is the generic algorithm of the two-dimensional pattern matching using finite automata. This algorithm is then used in a construction of particular exact and approximate pattern matching algorithms using finite automata. Each step of the particular two-dimensional pattern matching algorithm involves different types of finite automata for pattern matching. Types of finite automata used for preprocessing and pattern matching steps of the two-dimensional pattern matching algorithm are presented.

The main advantage of the use of finite automata is that they by definition process the text in time proportional to its length and this time does depend neither on the size of a pattern nor on the number of allowed errors. Furthermore, there are known simulation methods of (nondeterministic) finite automata or direct construction methods of deterministic finite automata.

The second contribution of this work presents pushdown automata that serve as a pattern matching and indexing tool for linearised two-dimensional texts. We present the transformation of two-dimensional structures into the form of a tree, preserving the context of each element of the structure. The tree can be linearised using the prefix notation into the form of one-dimensional text (string) and we do the pattern matching in this text. Pushdown automata indexing the two-dimensional text are presented. They allow to search for two-dimensional prefixes, suffixes, or factors of the two-dimensional text in time proportional to the size of the two-dimensional pattern representation. The two-dimensional pattern matching is then performed in time proportional to the length of the prefix notation of the tree, representing the two-dimensional pattern. This result achieves the properties analogous to the results obtained in tree pattern matching and one-dimensional text indexing.

Finally, the methods presented are applicable to problems of pattern matching in more than two dimensions.

1.5 Organisation of the thesis

This work has following major parts: after this short introduction we introduce notions and definitions used in the rest of the text (chapter 2, Definitions and Terminology, begins at

page 4). This chapter contains basic string, graph and tree-related definitions, automata related notions and at its end it presents a definition of the problem of the two-dimensional pattern matching and related notions (p. 14). The next chapter summarizes previous results of one- and two-dimensional pattern matching algorithms, tree and graph algorithms, that are somehow related to our work (chapter 3, Previous Results and Related Work, p. 19). The chapter that follows (chapter 4, Two-dimensional Pattern Matching Using Finite Automata, p. 39) is devoted to automata-based models of the two-dimensional pattern matching. At first, there is presented general model for the exact and approximate two-dimensional pattern matching. In the rest of this chapter several applications of this general principle are presented and some of their properties are studied. The next chapter (chapter 5, Dagology/Operations Over Linear Representation of Acyclic Graphs, p. 58) opens new topic of pushdown automata processing a linearised representation of multidimensional texts. Two-dimensional prefix, suffix and factor indexing pushdown automata construction algorithms are presented and some of their properties are studied. Finally, a summary of results and conclusions are contained in the last chapter beginning at page 75.

Background and survey of previous results

2 Definitions and Terminology

Let us start with a chapter devoted to summarize the basic notions and notations that are used in this work. The chapter is divided into several sections: first, the notions of sets and strings domain are presented, followed by definitions of pattern matching in strings. Next, finite and pushdown automata are mentioned, followed by notions of the theory of graphs. The final part of this chapter contains notions of multi-dimensional pattern matching.

2.1 Sets and strings

Definition 2.1 (Alphabet)
An *alphabet* A is a finite non-empty set of *symbols*. $|A|$ denotes its cardinality, i.e. the number of symbols in alphabet A.

Definition 2.2 (Complement of symbol)
A *complement* of symbol a over alphabet A, where $a \in A$, is a set $A \setminus \{a\}$ and is denoted by $\bar{a}$.

Definition 2.3 (String)
A *string* w over an alphabet A is any sequence of symbols from A.

Definition 2.4 (Empty string)
The *empty string* is denoted by ε and $|\varepsilon| = 0$.

Definition 2.5 (Length of string)
The *length* of a string w is the number of symbols of A in the string w and is denoted by $|w|$.

Definition 2.6 (Set of all strings)
The *set of all strings* over alphabet A is denoted by A^*, $\varepsilon \in A^*$. The set of all non-empty strings over A is denoted by A^+. The set of all strings of length m over A is denoted by A^m, where $0 \leq m$.

Definition 2.7 (Dictionary)
A *dictionary* is a finite set of strings over some alphabet.

Definition 2.8 (i^{th} element)
Let w over alphabet A be a string and let $w = a_1a_2a_3 \cdots a_{i-1}a_ia_{i+1} \cdots a_{|w|}$, where $a_j \in A$, $1 \leq j \leq |w|$. The i^{th} *element* of string w is the symbol a_i. It is denoted by $w[i]$ or w_i, $1 \leq i \leq |w|$.

Remark 2.9
Elements in strings in this work are numbered from one, the first element of string w is $w[1]$.

Definition 2.10 (Reduced alphabet)
Let P be a string over alphabet A and let its length be m, $m \geq 0$, $P \in A^m$. Let X be a subset of alphabet A, such that X consists of all symbols not present in the string P. Formally, either $X = \{x : \forall x \in A \wedge x \neq P[i], i = 1, \ldots, m, m > 0\}$, or $m = 0$ and $X = A$. Then subset X may be replaced by some $x \in X$. A *reduced alphabet* is an alphabet A', where $A' = \{\text{all distinct symbols of } P\} \cup \{x\}$. $|A'| \leq m + 1$.

Definition 2.11 (Ordering)
An *ordering* is a binary relation on some set. This relation is transitive and antisymmetric, for total ordering is also total.

Definition 2.12 (Ordered alphabet)
An *ordered alphabet* A is a finite ordered non-empty set of symbols.

Property 2.13
Let set A be totally ordered under $\leq$, then the following statements hold for all symbols $a, b, c \in A$:
1. Antisymmetry: If $a \leq b$ and $b \leq a$ then $a = b$.
2. Transitivity: If $a \leq b$ and $b \leq c$ then $a \leq c$.
3. Totality: $a \leq b$ or $b \leq a$.

Definition 2.14 (Ranked alphabet)
A *ranked alphabet* is a finite non-empty set of symbols, each symbol has a unique non-negative arity (rank). Given a ranked alphabet $A_{\#}$, the arity of symbol a is denoted by $\text{arity}(a)$. The set of symbols of arity p is denoted by $A_{\#p}$. Elements of arity $0, 1, 2, \ldots, p$ are respectively called nullary, unary, binary, ..., p-ary symbols. Nullary symbols are also called constants. It is assumed $A_{\#}$ contains at least one constant.

Note 2.15: In this work, symbols with arity are used only in parts dealing with ranked trees. Every symbol $a \in A$ used in these trees has $\text{arity}(a) = 2$. The only constant defined is the symbol #, $\text{arity}(\#) = 0$.

Definition 2.16 (Substring, factor)
String x is a *substring* or *factor* of string y if $y = uxv$ for some strings u, v, where $x, y, u, v \in A^*$.

Definition 2.17 (Prefix, suffix)
Substring x of string y, where $y = uxv$, is said to be a *prefix* of y if $u = \varepsilon$ and a *suffix* of y if $v = \varepsilon$.

Definition 2.18 (Powerset)
For any set S the set of all subsets of S is called *powerset* of S and is denoted by $\mathcal{P}(S)$.

2.2 Pattern matching in strings

For one-dimensional pattern matching we distinguish several *edit operations* ([MH97]). They are used to define edit distances for pattern matching.

Definition 2.19 (Replace)
Edit operation *replace* converts string uav to string ubv, where $u, v \in A^*$, $a, b \in A$, $a \neq b$. (One symbol is replaced by another and thus the operation preserves the length of the string.)

Definition 2.20 (Insert)
Edit operation *insert* converts string uv to string uav, where $u, v \in A^*$, $a \in A$. (One symbol is inserted into the string.)

Definition 2.21 (Delete)
Edit operation *delete* converts string uav to string uv, where $u, v \in A^*$, $a \in A$. (One symbol is removed from the string).

Definition 2.22 (Transpose)
Edit operation *transpose* converts string $uabv$ to string $ubav$, where $u, v \in A^*$, $a, b \in A$. (Two adjacent symbols are exchanged.)

Definition 2.23 (Hamming distance)
Hamming distance [Ham50] $D_H(v, w)$ between two strings $v, w \in A^*$, $|v| = |w|$ is the minimum number of edit operations *replace* (change of symbol), needed to convert string v to w.

Definition 2.24 (Levenshtein distance)
Levenshtein distance [Lev66] $D_L(v, w)$ between two strings $v, w \in A^*$, is the minimum number of edit operations *replace*, *insert* and *delete*, needed to convert string v to w.

Definition 2.25 (Damerau distance)
Damerau distance [Dam64] $D_D(v, w)$ between two strings $v, w \in A^*$, is the minimum number of edit operations *replace*, *insert*, *delete* and *transpose*, needed to convert v to w. Each symbol of string v can participate at most in one edit operation *transpose*. Sometimes this distance is called also the *generalized Levenshtein distance*.

For following definitions 2.26, 2.27 and 2.28 let A be an ordered alphabet [CCI+99].

Definition 2.26 (Δ distance)
Let $a_1, a_2, a_3, \ldots, a_{|A|}$ be symbols of ordered alphabet A. Then Δ distance of two symbols a_i, a_j, $i \leq j$, $a_i, a_j \in A$, is $\Delta(a_i, a_j) = |i - j|$.
Δ *distance* $D_\Delta(v, w)$ between two strings $v, w \in A^*$, $|v| = |w|$, is $\max_{1 \leq i \leq |v|} \Delta(v_i, w_i)$.

Definition 2.27 (Γ distance)
Γ *distance* $D_\Gamma(v, w)$ between two strings $v, w \in A^*$, $|v| = |w|$, is $\sum_{i=1}^{|v|} \Delta(v_i, w_i)$.

Definition 2.28 (Γ, Δ distance)
Γ, Δ *distance* $D_{\Gamma,\Delta}(v, w)$ is a combination of the previous two. $D_{\Gamma,\Delta}(v, w)$ between two strings $v, w \in A^*$, $|v| = |w|$, is a pair (k, l) such that $D_\Gamma(v, w) \leq k \wedge D_\Delta(v, w) \leq l$, where $k, l \in \mathbb{N}, l \leq k$.

Hamming distance D_H between two strings of equal length is the number of positions with mismatching symbols in these two strings. Levenshtein and Damerau distances D_L and D_D, respectively, between two strings P and R, not necessarily with equal length, are the minimal numbers of relevant operations required to modify original string P to R. Δ distance D_Δ between two strings of equal length is the maximum of Δ distances between symbols on the same positions in these two strings. Γ distance D_Γ between two strings of equal length is the sum of Δ distances between mismatching symbols on the same positions in these two strings.

Definition 2.29 (Text and Pattern)
Let P and T be two strings, $|P| = m$, $|T| = n$, and let $m \leq n$. P denotes a string called *pattern* that is searched for. T denotes a string called *text* in which the pattern is to be searched.

Definition 2.30 (Occurrence)
An *occurrence* of pattern $P = p_1p_2\cdots p_m$ in text $T = t_1t_2\cdots t_n$ is a position i in the text, $1 \leq i \leq n$, such that $p_j = t_{i+j-1}$ for all $j = 1, \ldots, m$.

An occurrence is
- exact, when P is a substring of T according to Def. 2.16,
- approximate (with k errors), when some string R is a substring of T and relevant edit distance $D(P, R) \leq k$.

Following definitions will introduce a notion of *matching*.

Definition 2.31 (Approximate pattern matching)
An *approximate pattern matching* is defined as a searching for all occurrences of pattern $P = p_1p_2\cdots p_m$ in text $T = t_1t_2\cdots t_n$ with at most k errors allowed, $k \in \mathrm{N}$. That is to verify whether pattern X occurs in text T so that the distance $D(P, X) \leq k$.

Types of errors allowed in a found substring are determined by the edit distance used for pattern matching, e.g. the Hamming distance (Def. 2.23).

Note 2.32: The approximate pattern matching using the Hamming distance is called *approximate pattern matching with k mismatches*.

Definition 2.33 (Exact pattern matching)
An *exact pattern matching* is the approximate pattern matching, where $k = 0$.

Definition 2.34 (Formal language)
A *formal language* L over alphabet A is an arbitrary subset of A^*, i.e. $L \subseteq A^*$.

Definition 2.35 (Languages for pattern matching)
Let $w \in A^*$ be a string (pattern) and $k \in \mathrm{N}$ be the number of allowed errors. Then we can define languages $E(w)$, $H_k(w)$, $\Gamma_k(w)$ as follows:
- $E(w) = \{w\}$ is a language for the exact pattern matching of w ($k = 0$).
- $H_k(w) = \{v;\ v \in A^*, D_H(v, w) \leq k\}$ is a language for the approximate pattern matching of w with the Hamming distance.
- $\Gamma_k(w) = \{v;\ v \in A^*, D_\Gamma(v, w) \leq k\}$ is a language for the approximate string matching of w with the Γ distance.

All these languages can be extended easily to operate over a set of strings X. Let $X = \{w_1, w_2, \ldots, w_{|X|}\}$, where $w_i \in A^*$, $1 \leq i \leq |X|$. In this work we need the following languages:
- $E(X) = \{w;\ w \in X\}$ is a language for the exact string matching of the set of strings X.
- $H_k(X) = \{v;\ v \in A^*, D_H(v, w) \leq k \wedge w \in X\}$ is a language for the approximate pattern matching of set of strings X using the Hamming distance.

Note that the exact matching of a set of strings can find more than one string at a time, the strings may be overlapping. For example, let a set of strings be $\{v, w\}$ and let v be a proper suffix of w, $w = uv$, $u \in A^+$. Then the exact matching will report two occurrences of w and v at the same time after reading w.

Remark 2.36
Languages for other distances are defined similarly.

2.3 Finite automata

Definition 2.37 (Finite automaton)
A *finite automaton*, *FA*, is a quintuple (Q, A, δ, I, F), where

- Q is a finite set of states,
- A is a finite input alphabet,
- δ is a mapping $Q \times (A \cup \{\varepsilon\}) \mapsto \mathcal{P}(Q)$, $\mathcal{P}(Q)$ is the powerset of set Q (Def. 2.18),
- $I \subseteq Q$ is a set of initial states,
- $F \subseteq Q$ is a set of final states.

Note 2.38: A finite automaton is called a nondeterministic finite automaton (NFA) in some literature [MHP05, Hol00].

Definition 2.39 (Configuration of FA)
A *configuration of FA* $M = (Q, A, \delta, I, F)$ is a pair $(q, x) \in Q \times A^*$. The *initial configuration of FA* is a pair (q_0, x), $q_0 \in I$, and a *final (accepting) configuration of FA* is a pair (q_f, ε), where $q_f \in F$.

Definition 2.40 (Transition of FA)
A *transition* of FA $M = (Q, A, \delta, I, F)$ is a relation $\vdash_M \subseteq (Q \times A^*) \times (Q \times A^*)$ defined as $(q_1, aw) \vdash_M (q_2, w)$, where $q_2 \in \delta(q_1, a)$, $a \in A \cup \{\varepsilon\}$, $w \in A^*$, $q_1, q_2 \in Q$. The symbol $\vdash_M^*$ denotes a reflexive and transitive closure of the relation $\vdash_M$.

Definition 2.41 (Finite transducer)
A *finite transducer* is a sextuple (Q, A, A', δ, I, F), where
- Q is a finite set of states,
- A is a finite input alphabet,
- A' is a finite output alphabet,
- δ is a mapping $Q \times (A \cup \{\varepsilon\}) \mapsto \mathcal{P}(Q \times A'^*)$,
- $I \subseteq Q$ is a set of initial states,
- $F \subseteq Q$ is a set of final states.

Definition 2.42 (Deterministic finite automaton)
A *deterministic finite automaton*, *DFA*, is a quintuple (Q, A, δ, q_0, F), where
- Q is a finite set of states,
- A is a finite input alphabet,
- δ is a mapping $Q \times A \mapsto Q$,
- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states.

Definition 2.43 (Configuration of DFA)
A *configuration of DFA* $M = (Q, A, \delta, q_0, F)$ is a pair $(q, x) \in (Q \times A^*)$. The *initial configuration of DFA* is a pair (q_0, x) and a *final (accepting) configuration of DFA* is a pair (q_f, ε), where $q_f \in F$.

Definition 2.44 (Transition of DFA)
A *transition of DFA* $M = (Q, A, \delta, q_0, F)$ is a relation $\vdash_M \subseteq (Q \times A^+) \times (Q \times A^*)$ defined as $(q_1, aw) \vdash_M (q_2, w)$, where $\delta(q_1, a) = q_2$, $a \in A$, $w \in A^*$, $q_1, q_2 \in Q$. The symbol $\vdash_M^*$ denotes a reflexive and transitive closure of the relation $\vdash_M$.

Definition 2.45 (Run of finite automaton)
A *run of a finite automaton* (FA or DFA) M is a sequence of configurations starting from the initial configuration that are used by M as a response to the input word.

Definition 2.46 (Pattern matching automaton)
A *pattern matching automaton* is a finite automaton that may have its transition mapping modified compared to the definition of the finite automaton (FA or DFA), e.g. the AC automaton (or machine) uses "forward" δ function and "backward" *fail*-function [AC75]; or it may be able to do some additional operations needed in applications, e.g. it may have some "actions" assigned to some or all of its transitions or states. The latter possibility is a generalization of the finite transducer (Def. 2.41).

Note 2.47: In this work, the notion "finite automaton" is sometimes used to denote a pattern matching automaton. It is possible, because the core of all automata-based algorithms for the pattern matching is the run (or its simulation) of either finite automaton or deterministic finite automaton.

Definition 2.48 (Transition diagram)
A *transition diagram* is an oriented labeled graph representing a finite automaton. The nodes of the graph are labeled by the names of its states, the edges represent its transitions. Each edge (transition) is labeled by a symbol which has been read during a transition corresponding to this edge or by ε if no input is read. An initial state is denoted by an arrow without origin, a final state by two concentric circles.

Note 2.49: In examples and figures, there may appear some transitions of finite automaton seemingly labeled by a set of symbols. This represents a set of transitions each labeled by one symbol of the set of symbols. Another transitions of a finite automaton may be labeled by "negated" symbol (e.g. $\bar{x}$), representing a complement of symbol x in alphabet A, (see Def. 2.2). Corresponding transitions are then labeled by all symbols of alphabet A except symbol x.

Definition 2.50 (Language accepted by finite automaton)
A *language accepted* by finite automaton $M = (Q, A, \delta, I, F)$ is the set $L(M) = \{w;\ w \in A^*, (q_0, w) \vdash_M^* (q, \varepsilon)$ for some $q_0 \in I$ and $q \in F\}$. Backwards, the automaton accepting language L is denoted $M(L)$.

Definition 2.51 (Equivalence of finite automata)
Two finite automata M and M' are *equivalent* if $L(M) = L(M')$.

Definition 2.52 (Active state)
An *active state* of finite automaton $M = (Q, A, \delta, I, F)$ after reading input string $w \in A^*$ is each state q, $q \in Q$, such that $(q_0, w) \vdash_M^* (q, \varepsilon)$, $q_0 \in I$.

Definition 2.53 (Simulation of FA)
Algorithm B *simulates* a run of FA $M = (Q, A, \delta, I, F)$, if $\forall w, w \in A^*$, it holds that B with given w at the input reports all information associated with each final state $q_f, q_f \in F$, after processing w, if and only if there exists a run (sequence of configurations) $(q_0, w) \vdash_M^* (q_f, \varepsilon)$, where $q_0 \in I$.

2.4 Pushdown automata

Definition 2.54 (Extended Pushdown Automaton)
An *extended pushdown automaton* (*extended PDA*), is a septuple $(Q, A, G, \delta, q_0, Z_0, F)$, where
- Q is a finite set of states,
- A is a finite input alphabet,
- G is a finite pushdown store alphabet,
- δ is a mapping $Q \times (A \cup \{\varepsilon\}) \times G^* \mapsto \mathcal{P}(Q \times G^*)$,
- $q_0 \in Q$ is an initial state,
- $Z_0 \in G$ is the initial pushdown store symbol,
- $F \subseteq Q$ is a set of final states.

Definition 2.55 (Extended Pushdown Store Operation)
An *extended pushdown store operation* of an extended pushdown automaton M, $M = (Q, A, G, \delta, q_0, Z_0, F)$, is a relation $(A \cup \{\varepsilon\}) \times G^* \mapsto G^*$. An extended pushdown store operation produces new contents on the top of the pushdown store by taking one input symbol or the empty string from the input and the current contents on the top of the pushdown store or the empty string from the top of the pushdown store.

Remark 2.56
The extension of an extended pushdown automaton is in its transition function definition and specifically in definition of its pushdown operations. Pushdown automaton has its transition mapping δ defined as follows: $\delta : Q \times (A \cup \{\varepsilon\}) \times G \mapsto \mathcal{P}(Q \times G^*)$, allowing only one pushdown store symbol to be read at every pushdown store operation. The basic pushdown store operation is then $(A \cup \{\varepsilon\}) \times G \mapsto G^*$.

Note 2.57: If there is no risk of ambiguity, the word "extended" from "extended pushdown store operation" may be omitted in the following text.

Definition 2.58 (Configuration of PDA)
A *configuration of PDA* and extended PDA $M = (Q, A, G, \delta, q_0, Z_0, F)$ is a triplet $(q, w, x) \in Q \times A^* \times G^*$. The *initial configuration* of a pushdown automaton is a triplet (q_0, w, Z_0) for the input string $w \in A^*$.

Definition 2.59 (Transition of extended PDA)
A *transition* of an extended pushdown automaton is the relation $\vdash_M \subseteq (Q \times A^* \times G^*) \times (Q \times A^* \times G^*)$. It holds that $(q, aw, \alpha\beta) \vdash_M (p, w, \gamma\beta)$ if $(p, \gamma) \in \delta(q, a, \alpha)$. The k-th power, transitive closure, and transitive and reflexive closure of the relation $\vdash_M$ is denoted $\vdash_M^k$, $\vdash_M^+$, $\vdash_M^*$, respectively.

Definition 2.60 (Extended deterministic pushdown automaton)
A pushdown automaton is an *extended deterministic pushdown automaton* if it holds:
1. $|\delta(q, a, \gamma)| \leq 1$ for all $q \in Q$, $a \in A \cup \{\varepsilon\}$, $\gamma \in G^*$.
2. If $\delta(q, a, \alpha) \neq \emptyset$, $\delta(q, a, \beta) \neq \emptyset$ and $\alpha \neq \beta$ then α is not a suffix of β and β is not a suffix of α.
3. If $\delta(q, a, \alpha) \neq \emptyset$, $\delta(q, \varepsilon, \beta) \neq \emptyset$, then α is not a suffix of β and β is not a suffix of α.

Definition 2.61 (Input driven pushdown automaton)
A pushdown automaton is *input-driven* if each of its pushdown store operations at particular state q, $q \in Q$, is determined only by the input symbol.

Formally, let M be an input driven pushdown automaton, $M = (Q, A, G, \delta, q_0, Z_0, F)$. Then $|\delta(q, a, \gamma)| \leq 1$ for all $q \in Q$, $a \in A \cup \{\varepsilon\}$, $\gamma \in G^*$.

Definition 2.62 (Language accepted by pushdown automaton)
A *language L accepted by pushdown automaton M* is a set of words over finite alphabet A. It is defined in two distinct ways:
1. *Accepting by final state*:

$$L(M) = \{x;\ \delta(q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \gamma), x \in A^*, \gamma \in G^*, q \in F\}.$$

2. *Accepting by empty pushdown store*:

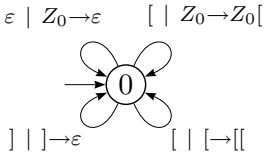
$$L_\varepsilon(M) = \{x;\ (q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \varepsilon), x \in A^*, q \in Q\}.$$

If the pushdown automaton accepts the language by empty pushdown store then the set F of its final states is the empty set.

Example 2.63
We are given context-free grammar G, $G = (N, T, P, S)$. Set of production rules P of grammar G is $P = \{S \to [S]S, S \to \varepsilon\}$; set of nonterminal symbols $N = \{S\}$, set of terminal symbols $T = \{[,]\}$. $L(G)$ is a language whose words are sequences of paired square brackets, brackets can be nested. Clearly, $L(G)$ is a context-free language.

Since $L(G)$ is a context-free language, there exists a pushdown automaton $M = (\{q_0\}, A, G, \delta, q_0, Z_0, F)$ such that $L(M) = L(G)$. Let M be accepting the input by an empty pushdown store. Set of transitions δ of automaton M is

$$\begin{aligned}\delta(q_0, [, Z_0) &= \{(q_0, Z_0[)\}\\ \delta(q_0, [, [) &= \{(q_0, [[)\ \}\\ \delta(q_0,], [) &= \{(q_0, \varepsilon)\ \}\\ \delta(q_0, \varepsilon, Z_0) &= \{(q_0, \varepsilon)\ \}\end{aligned}$$



δ	[	]	ε
Z_0	$Z_0($	*reject*	*accept*
]	[[	ε	

This example demonstrates an input-driven pushdown automaton where all pushdown store operations are determined by the input symbol.

Transitions performed by automaton M on input text [ ] [[ ]] are as follows, the pushdown store grows to the right:

$$\begin{aligned}(q_0, [\,][\,[\,]\,], Z_0) &\vdash (\,q_0,]\,][\,[\,]\,], Z_0[\ )\\ &\vdash (\,q_0,\ [\,[\,]\,], Z_0\ \ )\\ &\vdash (\,q_0,\ \ [\,]\,], Z_0[\ )\\ &\vdash (\,q_0,\ \ \ ]\,], Z_0[\,[\,)\\ &\vdash (\,q_0,\ \ \ \ ], Z_0[\ )\\ &\vdash (\,q_0,\ \ \ \ \varepsilon, Z_0\ \ )\\ &\vdash (\,q_0,\ \ \ \ \varepsilon, \varepsilon\ \ \ )\,(\mathit{accept})\end{aligned}$$

Remark 2.64
In this work, the top of the pushdown store is always on the righthand side of the particular string showing its elements, i.e. initial pushdown store symbol Z_0 is written first and the pushdown store grows to the right.
If a pushdown store operation $(a, \alpha, \beta\gamma)$, $a \in A$, $\alpha, \beta, \gamma \in G$, is executed by a PDA, it removes (pops) α from the top of the pushdown store and stores (pushes) β and γ onto the top of the pushdown store, in this order.

The following definitions introduce a special type of pushdown automata and languages they accept. This theory is due to Alur and Madhusudan [AM04]. Alur and Madhusudan starts definitions introducing a *pushdown alphabet*. In fact it is the input alphabet of the pushdown automaton, hence we name it a *visible alphabet* to distinguish it unambiguously from the pushdown store alphabet.

Definition 2.65 (Visible alphabet)
A *visible alphabet* $\tilde{A}$ is a triple $\tilde{A} = (A_c, A_r, A_{int})$. $\tilde{A}$ comprises three categories of symbols, three disjoint finite alphabets: A_c is a finite set of calls (pushdown store grows), A_r is a finite set of returns (pushdown store shrinks), A_{int} is a finite set of internal actions that do not use the pushdown store.

In [AM04], they define A_{int} as a finite set of internal actions that do not use the pushdown store, this is possible using extended pushdown automaton (Def. 2.60). Not to read from the top of the pushdown store is not possible for pushdown automaton in general, however. The authors state the fact that it is possible to keep track of the top of the pushdown store in the internal states of a pushdown automaton. Since we do not need to use any internal actions in visibly pushdown automata used for our purposes, we will not elaborate it further.

Definition 2.66 (Visibly pushdown automaton)
A *visibly pushdown automaton* (VPA) is a septuple $(Q, \tilde{A}, G, \delta, Q_{in}, Z_0, F)$, where
- Q is a finite set of states,
- $\tilde{A}$ is a finite visible alphabet,
- G is a finite pushdown store alphabet,
- δ is a mapping: $(Q \times A_c \times \varepsilon) \mapsto (Q \times (G \setminus \{Z_0\})) \cup$
  $(Q \times A_r \times G) \mapsto (Q \times \varepsilon) \cup$
  $(Q \times A_{int} \times \varepsilon) \mapsto Q \times \varepsilon)$,
- $Q_{in} \subseteq Q$ is a set of initial states,
- $Z_0 \in G$ is the initial pushdown store symbol,
- $F \subseteq Q$ is a set of final states.

All notions related to extended pushdown automata hold for visibly pushdown automata as well. Specifically, the language accepted by visibly pushdown automaton (cf. Def. 2.62).

Definition 2.67 (Language accepted by visibly pushdown automaton)
A *language* $L(M)$ *accepted by visibly pushdown automaton* M is the set of words accepted by M.

Definition 2.68 (Deterministic visibly pushdown automaton)
A *deterministic visibly pushdown automaton* is a VPA $M = (Q, \tilde{A}, G, \delta, Q_{in}, Z_0, F)$, where $|Q_{in}| = 1$ and for every $q \in Q$ it holds:
- for every $a \in A_{int}$, there is at most one transition of the form $(q, a, \varepsilon, q', \varepsilon) \in \delta$,
- for every $a \in A_c$, $\gamma \in G$, there is at most one transition of the form $(q, a, \varepsilon, q', \gamma) \in \delta$,
- for every $a \in A_r$, $\gamma \in G$, there is at most one transition of the form $(q, a, \gamma, q', \varepsilon) \in \delta$.

Definition 2.69 (Visibly pushdown language)
A *visibly pushdown language* is a set of words over some finite alphabet A, $L \subseteq A^*$ with respect to $\tilde{A}$ ($\tilde{A}$-VPL) if there exists a visibly pushdown automaton M over $\tilde{A}$ such that $L(M) = L$.

The visibly pushdown automata and visibly pushdown languages are primarily used in program analysis. We note that visibly pushdown automata are related to input-driven pushdown automata in the sense of classification of pushdown operations based on the input symbol (Def. 2.61). However, they are weaker in the particular aspect of low cost determinisation process, since they do not have the limit of one pushdown operation per one input symbol at each of its states. That is, $|\delta(q, a, \gamma)| \leq 1$ for all $q \in Q$, $a \in A \cup \{\varepsilon\}$, $\gamma \in G^*$, does not hold for them.

Visibly pushdown automata can be determinised.

Theorem 2.70 ([AM04], Theorem 2)
For any visibly pushdown automaton M over $\tilde{A}$, there exists a deterministic VPA M' over $\tilde{A}$ such that $L(M') = L(M)$. Moreover, if M has $|Q|$ states, it is possible to construct M' with $\mathcal{O}\left(2^{|Q|^2}\right)$ states and with pushdown store alphabet of size $\mathcal{O}\left(2^{|Q|^2}|A_c|\right)$.

Proof: *ibidem*.

2.5 Notions of graph theory

For more informations on graphs and trees see Valiente's book [Val02].

2.5.1 Directed graphs

Definition 2.71 ((Directed) graph)
A *(directed) graph* G is a pair (V, E), where V is a finite non-empty set of nodes (vertices) and E is a finite set of (directed) edges (arcs), $E \subseteq V \times V$. The order of a graph is the number of its vertices, the size of a graph is the number of its edges. Edge $e = (u, v)$, $e \in E$, $u, v \in V$, is said to be incident with vertices u and v, u is the source, v is the target of edge e. u and v are said to be adjacent. The in-degree of vertex v is the number of edges in G whose target is v, $\text{indeg}(v) = |(u, v); (u, v) \in E|$. Analogously, the out-degree of vertex v is the number of edges in G whose source is v, $\text{outdeg}(v) = |(v, w); (v, w) \in E|$. The degree of v is the sum of the in-degree and out-degree of the vertex, $\deg(v) = \text{indeg}(v) + \text{outdeg}(v)$.

Definition 2.72 (Undirected graph)
A graph $G = (V, E)$ is *undirected* if $(u, v) \in E$ implies $(v, u) \in E$, for all $u, v \in V$.

Definition 2.73 (Walk, Trail, Path)
A *walk* from vertex v_i to vertex v_j in a graph is an alternating sequence of vertices and edges $[v_i, e_{i+1}, v_{i+1}, e_{i+2}, \ldots, e_{j-1}, v_{j-1}, e_j, v_j]$, such that $e_k = (v_{k-1}, v_k)$ for $k = i + 1, \ldots, j$.
A *trail* is a walk with no repeated edges.
A *path* is a trail with no repeated vertices, the initial and final vertices may be the only exception. The length of a walk, trail or path is the number of edges in the sequence.

Definition 2.74 (Cycle)
A walk, trail or path $[v_i, e_{i+1}, v_{i+1}, e_{i+2}, \ldots, e_{j-1}, v_{j-1}, e_j, v_j]$ is said to be closed if $v_i = v_j$. A *cycle* in graph is a closed path of length at least one.

Note 2.75: If there are no alternative edges between any two vertices in the alternating sequence of vertices and edges $[v_i, e_{i+1}, v_{i+1}, e_{i+2}, \ldots, e_{j-1}, v_{j-1}, e_j, v_j]$, a walk, trail or path in graph can be uniquely represented by a sequence of its edges $[e_{i+1}, e_{i+2}, \ldots, e_j]$ or vertices $[v_i, v_{i+1}, v_{i+2}, \ldots, v_j]$ in the alternating sequence of vertices and edges.

Definition 2.76 (Directed acyclic graph)
A *directed acyclic graph* (DAG) is a directed graph that has no cycle.

Definition 2.77 (Ordered directed graph)
An *ordered directed graph* G is a pair (V, R), where V is a set of vertices and R is a set of linearly ordered lists of edges such that each element of R is of the form $((f, g_1), (f, g_2), \ldots, (f, g_n))$, where $f, g_1, g_2, \ldots, g_n \in V$, $n \geq 0$. Such an element indicates that for node f, there are n edges leaving f, the first edge connects f with g_1, the second connects f with g_2, and so forth.

Definition 2.78 (Connected directed graph)
A *connected directed graph* G, $G = (V, E)$, is a graph in which for every pair of vertices there exists an undirected walk between them, that is a walk where the edge orientation is not considered.

2.5.2 Trees

Note 2.79: The vertices of a tree are called *nodes*.

Definition 2.80 (Tree (Rooted directed tree))
A *rooted directed tree* is an acyclic directed connected graph T, $T = (V, E)$, with distinguished node $r \in V$, called the root of the tree, where it holds that the undirected graph created from T by removing orientation of its edges is acyclic (cf. Def. 2.76). For all nodes $v \in V$, there exists a path in G from root r to node v. Node r has in-degree $\text{indeg}(r) = 0$, all nodes except r have in-degree 1. Any node v with $\text{outdeg}(v) = 0$ is called a leaf.

Definition 2.81 (Rooted ordered labeled directed tree)
A *rooted ordered labeled directed tree* T, $T = (V, E)$, is a rooted directed tree where every node $v \in V$ is labeled by symbol $a \in A$ and its out-degree is $\text{arity}(a)$. Analogously to rooted directed tree, nodes labeled by nullary symbols (constants) are called *leaves*.

The hierarchical structure of trees allows to measure depth and height of its nodes and of the tree itself. A depth is the distance measured as the number of edges from the top of the tree (from the root). A height counts the number of edges from leaves to particular node in the tree.

Definition 2.82 (Height of node and tree)
Let $T = (V, E)$ be a tree. The *height* of node $u \in V$, denoted $\text{height}(u)$, is the length of a longest path from node u to any node in the subtree of T rooted at node u, for all nodes $u \in V$. The *height of tree* T, denoted $\text{height}(T)$, is the maximum among the heights of all nodes $u \in V$.

Definition 2.83 (Depth)
Let $T = (V, E)$ be a tree. The *depth* of node $u \in V$, denoted $\text{depth}(u)$, is the length of the unique path from the root node to node u, for all nodes $u \in V$. The *depth of tree* T is the maximum among the depths of all nodes $u \in V$.

The following definitions are useful for matching in trees and are taken from [JM10] where the details can be found.

Definition 2.84 (Tree pattern, Tree template)
Let A be a ranked alphabet. Let S be a special nullary symbol not in A, which serves as a placeholder for any subtree. A *tree pattern* is an ordered labeled ranked tree over ranked alphabet $A \cup \{S\}$. A tree pattern consisting of single S is not allowed.
A tree pattern containing at least one symbol S is called a *tree template*.
A tree pattern in prefix notation is an ordered labeled ranked tree over ranked alphabet $A \cup \{S\}$ in prefix notation.

Definition 2.85 (Tree matching, Treetop matching)
A *tree matching* of tree pattern p in subject tree t (a tree in which the matching is done) is to locate all occurrences of p in t. A tree pattern p with $k \leq 0$ occurrences of symbol S matches object tree t at node n if there exist subtrees $t_1, t_2, \ldots, t_k$ (not necessarily the same) of the tree t such that the tree p', obtained from p by substitution of subtree t_i for the i-th occurrence of S in p, $i = 1, 2, \ldots, k$, is equal the the subtree of t rooted at node n.
If all occurrences $t_1, t_2, \ldots, t_k$ must have its root identical to the root of tree t, the matching is called a *treetop matching*.

2.6 Notions of multi-dimensional pattern matching

In the previous sections some necessary definitions of the one-dimensional pattern matching were mentioned. In the area of two-dimensional pattern matching the situation is a little

bit more complicated, because the available literature sometimes define the same terms in a different manner or do not define them at all (cf. [CR94, GR97, and others]). The purpose of this section is to extend some of the above presented terms from 1D to 2D space in accordance with the literature available. Whenever some newly defined notion has its 1D equivalent, it is formulated as analogously as possible.

Definition 2.86 (Shape)
A *shape* is a geometrical formation in n-dimensional space, $n \in \mathbf{N}$.

Note 2.87: The meaning of the term *shape* vary in literature. For example, Giancarlo and Grossi [GG97b] define special 2D characters over an ordered alphabet, $\{\mathcal{IN}, \mathcal{SW}, \mathcal{NW}, \mathcal{SE}, \mathcal{NE}\}$. Shape $\mathcal{IN}$ is the 1×1 array, the other shapes denote paired combinations of top ($\mathcal{N}$)/bottom ($\mathcal{S}$)/left ($\mathcal{W}$)/right ($\mathcal{E}$) edges of a rectangular array.

Definition 2.88 (Picture)
A *picture* (multidimensional string) is a multidimensional shape consisting of elements, symbols over some alphabet A. If not stated otherwise, a picture in this work has two-dimensions. Formally, (two-dimensional) picture P is an element of set A^{**}, $P \in A^{**}$.

Definition 2.89 (Set of all pictures)
The *set of all pictures* over alphabet A is denoted A^{**}. (A two-dimensional language over A (a picture language) is thus a subset of A^{**}.)

The set of all pictures of size $(n \times n')$ over A, where $n, n' > 0$, is denoted $A^{(n \times n')}$.

Definition 2.90 (Element of a picture)
An *element of a picture* P, denoted $P[i, j]$, is a symbol over alphabet A, $P[i, j] \in A$.

Definition 2.91 (Two-dimensional array)
An *array* is a rectangular picture P of symbols taken from finite alphabet A. Formally, let P be a two-dimensional array, $P \subseteq A^{(n \times n')}$. $\forall i, j$, $1 \leq i \leq n$, $1 \leq j \leq n'$; $P[i, j] \in A$.

Remark 2.92
The following special symbols are used in this text: $\emptyset$ denotes the empty set, $\oslash$ is a special "filling" symbol for the two-dimensional arrays, and symbol $\varnothing$ is a "don't care" symbol. The "don't care" symbol is a special universal symbol that matches any other symbol including itself [FP74, MH97, CR02]. The "don't care" symbol and its use in the one-dimensional pattern matching was first studied by Fischer and Paterson [FP74].

Definition 2.93 (Bounding array)
Let P be a picture, $\oslash$ be a special (filling) symbol, A be an alphabet, $\oslash \notin A$. Let picture P be completed using symbol $\oslash$ so that it forms a 2D array R (Def. 2.91).
R is a *bounding array* of P, all elements of bounding array R undefined by picture P be defined as $\oslash$. Formally, let P be a picture over A, $P \in A^{(n \times n')}$, and R its bounding array of size $(x \times y)$ (Def. 2.94). Then

$$\forall i, j;\ 1 \leq i \leq n \leq x, 1 \leq j \leq n' \leq y : R[i, j] = \begin{cases} P[i, j]; & \text{if } P[i, j] \in A, \\ \oslash \quad ; & \text{otherwise}. \end{cases}$$

Furthermore, it holds that $P \sqsubseteq R$ (Def. 2.99).

Definition 2.94 (Picture size)
Let P be a picture and R be a bounding array of P, $R \in A^{(x \times y)}$. A *size* of a picture is the size of its bounding array, denoted by $|P|$ or $(x \times y)$, its numerical value is the product of its components, $|P| = xy$.

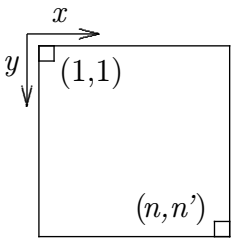
Definition 2.95 (Minimal bounding array)
Let P be a picture, A be an alphabet, symbol $\oslash \notin A$, $P \in A^{(n \times n')}$. A *minimal bounding array* R_{min} of picture P is a bounding array of the minimal size, that is $(n \times n')$.

Obviously, for every picture P there exists its bounding array R.

Note 2.96: In order to be consistent with those parts of this work, where non-rectangular pictures are considered, the rectangular shapes will be called *arrays* instead of in literature also used *pictures*.

Definition 2.97 (Picture origin)
The *origin* of a picture is the element at position $(1, 1)$.

The size of the figured array is $(n \times n')$. Geometrical axes have their source at the upper left corner.
$\vec{x}$-axis therefore points from left to right, $\vec{y}$-axis points from top to bottom of the array. (Axes oriented in this manner are used in the Computer Graphics.)



Definition 2.98 (Empty picture)
The *empty picture* is a picture denoted by λ and its size is $|\lambda| = (0 \times 0)$.

The picture of size $(0 \times n)$ and $(n \times 0)$, where $n > 0$ are not defined.

Definition 2.99 (Sub-picture)
Let $R \in A^{(n \times n')}$ be a picture of size $(n \times n')$ and $P \in A^{(m \times m')}$ be a picture of size $(m \times m')$. Picture P is a *sub-picture* (a *block*, a *sub-array*) of R if and only if $m \leq n \wedge m' \leq n'$ $(|P| \leq |R|)$ and every element of P occurs on the appropriate position in R.
Formally, $\exists k, l;\ k \leq n - m,\ l \leq n' - m'$ such that $P[i, j] = R[i + k, j + l];\ \forall i, j,\ 1 \leq i \leq m,\ 1 \leq j \leq m'$, where $P[i, j] \neq \oslash$ (picture P is included in R as a sub-picture, $P \sqsubseteq R$). A sub-picture is called a sub-array if it has the rectangular shape.

Definition 2.100 (Picture inclusion)
Let $R \in A^{(n \times n')}$ be a picture of size $(n \times n')$ and $P \in A^{(m \times m')}$ be a picture of size $(m \times m')$, $1 \leq m \leq n, 1 \leq m' \leq n'$, and P be a sub-picture of R. The *inclusion of* some *picture* P in R will be denoted by $\sqsubseteq$, $P \sqsubseteq R$.
$P \sqsubset R$, if it additionally holds either $1 \leq m < n \vee 1 \leq m' < n'$, $|P| < |R|$, or $1 \leq m = n \wedge 1 \leq m' = n'$ and there exists element $R[i, j]$ such that $P[i, j]$ does not exist (there is no element defined at i, j in P).

Property 2.101
Relation $\sqsubseteq$ is reflexive, transitive and antisymmetric.

Proof
1. Reflexive: It holds that $P \sqsubseteq P$ for all $P \in A^{**}$ (Def. 2.99).
2. Transitive: Let $P \sqsubseteq R$ (Def. 2.100). Let S be $S \in A^{(i \times j)}$, $1 \leq i \leq m, 1 \leq j \leq m'$, $S \sqsubseteq P$. Since $|S| \leq |P| \leq |R|$, then also $S \sqsubseteq R$.
3. Antisymmetric: Let $P \sqsubseteq R$ and $R \sqsubseteq P$. Using Def. 2.99 and Def. 2.100, it holds that $|P| \leq |R|$. If $|P| < |R|$, then $R \not\sqsubseteq P$, hence the only possibility is $|P| = |R|$ and thus $P = R$. □

Property 2.102
Relation $\sqsubset$ is transitive.

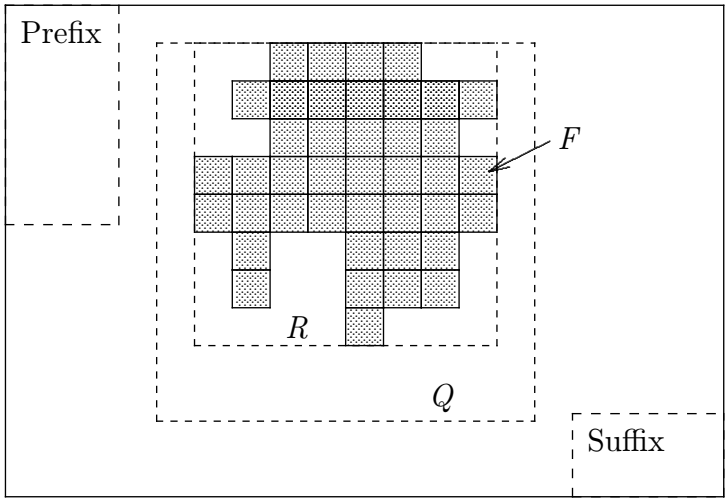


Figure 2.1. 2D prefix, suffix, and factor F with its bounding arrays Q and (minimal) R

Proof
Let $1 \leq m < n$, $1 \leq m' < n'$, then $P \sqsubset R$ (Def. 2.100).
Transitive: Let S be $S \in A^{(i \times j)}$, $1 \leq i \leq m, 1 \leq j \leq m'$, $S \sqsubseteq P$. From the assumption, it holds $|S| < |R|$. Since $|P| < |R|$ and $|S| \leq |P| < |R|$, then $S \sqsubset R$. □

Definition 2.103 (Two-dimensional factor)
A *two-dimensional factor* of two-dimensional array T is a picture F such that $F \sqsubseteq T$.

Definition 2.104 (Two-dimensional prefix)
A *two-dimensional prefix* of two-dimensional array T is a picture F_p, $F_p \sqsubseteq T$ and there exist at least two elements $e_l, e_k \in F_p$, $e_l, e_k \neq \oslash$, such that $e_l = F_p[1, j]$, $e_k = F_p[i, 1]$, where $1 \leq i \leq n$, $1 \leq j \leq n'$.

Definition 2.105 (Two-dimensional suffix)
A *two-dimensional suffix* of two-dimensional array T is a picture F_s, $F_s \sqsubseteq T$ and there exist at least two elements $e_l, e_k \in F_s$, $e_l, e_k \neq \oslash$, such that $e_l = F_p[n, j]$, $e_k = F_p[i, n']$, where $1 \leq i \leq n$, $1 \leq j \leq n'$.

Remark 2.106
Rectangular two-dimensional factors, prefixes and suffixes can be defined directly from definitions 2.103, 2.104, 2.105 above, using the bounding rectangle R of F, F_p, F_s, respectively; $R \sqsubseteq T$.

Figure 2.1 depicts 2D text T with one of its 2D prefixes, one of its 2D suffixes, and with picture F, which is a 2D factor of T. 2D factor F has its bounding arrays Q and (minimal) R, $F \sqsubset R \sqsubset Q$. All these are also 2D factors of T.

In the rest of the text two terms will be frequently used: two-dimensional *pattern array* (PA) and *text array* (TA).

Definition 2.107 (Two-dimensional pattern matching problem)
A *two-dimensional pattern matching* problem is to locate $(m \times m')$ pattern array PA inside $(n \times n')$ text array TA (Fig. 2.2).

Definition 2.108 (Two-dimensional occurrence)
A *two-dimensional occurrence* of pattern array PA in text array TA is

– exact, when PA is included in TA as a sub-array,
– approximate (with at most k errors), if for some sub-array X of TA and some 2D edit distance *2D-dist*(PA, X) is minimal or *2D-dist*$(PA, X) \leq k$.

Note 2.109: When there is no risk of ambiguity, we may refer to a 2D occurrence simply as an occurrence.

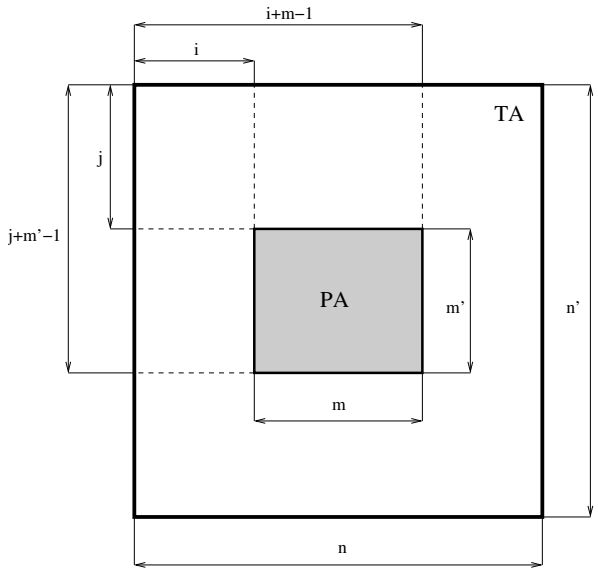


Figure 2.2. Pattern array PA occurs at position (i, j) in text array TA

The definition of 2D operation *replace* is straightforward and is similar to Def. 2.19.

Definition 2.110 (Two-dimensional replace)
An edit operation *two-dimensional replace* converts array P to array R of the same size, $|P| = |R|$. Symbol $P[i, j] = x$ is replaced by another symbol $y = R[i, j]$, $y \neq x$, $x, y \in A$.

Definition 2.111 (Two-dimensional Hamming distance)
A *two-dimensional Hamming distance* $D_{2H}(P, R)$ between two arrays P and R, $P, R \in A^{**}$, $|P| = |R|$, is the minimum number of edit operations *2D replace* needed to convert array P to R.

Definition 2.112 (Two-dimensional approximate pattern matching)
A *two-dimensional approximate pattern matching* is searching for all occurrences of pattern array PA in text array TA (Fig. 2.2) with at most k errors allowed, $k \in \mathrm{N}$. The maximum number of edit operations (errors) and theirs types allowed in a found sub-array are determined by the distance used (e.g. 2D Hamming).

Definition 2.113 (2D approximate pattern matching using the 2D Hamming distance)
A two-dimensional pattern matching using the 2D Hamming distance k, $k \in \mathrm{N}$, means to find all occurrences of PA, $PA \in A^{(m \times m')}$, in TA with equal or less than k mismatching symbols, $k < mm'$.

Note 2.114: The 2D approximate pattern matching using the 2D Hamming distance is called also the *2D approximate pattern matching with k mismatches.*

Definition 2.115 (Two-dimensional exact pattern matching)
A *two-dimensional exact pattern matching* is two-dimensional approximate pattern matching, where errors are not allowed, i.e. $k = 0$.

3 Previous Results and Related Work

In this chapter, an overview of selected topics from the broad area of one and two-dimensional pattern matching is presented. The chapter starts with algorithms of one-dimensional pattern matching, tree algorithms and directed acyclic graphs. Next are two-dimensional exact and approximate pattern matching algorithms and structures for multidimensional indexing.

3.1 One-dimensional pattern matching

3.1.1 Automata based pattern matching principles

It has been shown that many string matching algorithms, both exact and approximate, in fact simulate a run of a (possibly nondeterministic) finite automaton [MHP05]. In this section, the principles, classification and selected pattern matching automata and their construction methods are presented.

3.1.1.1 *Types of 1D pattern matching automata*

In 1997 Melichar and Holub showed in [MH97] that all 1D pattern matching problems are sequential problems and therefore it is possible to solve them using finite automata. Moreover, they presented *six-dimensional classification* of all (1D) pattern matching problems for an alphabet of finite size.

One-dimensional pattern matching problems for a finite size alphabet can be classified according to several criteria. The authors of [MH97] presented six criteria leading to 6D space in which each point corresponds to the particular pattern matching problem. The classification criteria and types of problems included in them (see Fig. 3.1) are:

1. a nature of a pattern (String, seQuence);
2. an integrity of a pattern (Full pattern, Subpattern);
3. a number of patterns (One, Finite number, Infinite number);
4. a way of matching: exact (E), approximate using Hamming (R), Levenshtein (D) and Damerau (generalized Levenshtein) (T) distance, approximate matching using Δ (G), Γ (L) or combined Γ, Δ (H) distance;
5. an "importance" of symbols in a pattern (take Care of all symbols, Don't care of some symbols);
6. number of instances of a pattern (One, finite Sequence).

In order to make references to the particular pattern matching problem easy, abbreviations for all problems are used. They are summarized in Tab. 3.1.

The original model has been updated in its fourth dimension with distances used in the area of musicology (Δ, Γ) [CCI$^+$99]. The update appeared in [ŽM04]. (An evaluation of applicability of the stringological approach to musicology, including these distances, is described in [BB01].)

Δ and Γ distances were not known at the time of publication of [MH97]. However, their discovery did not influenced validity of the 6D classification, the only required action was to update the appropriate dimension with them. As a consequence, the number of problems described by this classification have risen from 192 to $2 \cdot 2 \cdot 3 \cdot 7 \cdot 2 \cdot 2 = 336$.

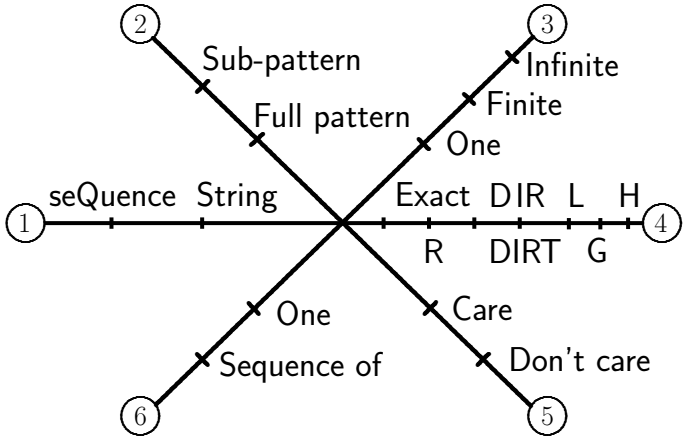


Figure 3.1. Updated classification of one-dimensional pattern matching problems

The fact that this classification was able to adapt in spite of new discoveries confirms it is valid and useful. We think that the most likely updates in the future will be done right in this "distances" dimension.

Together, those criteria describe all now known subtypes of pattern matching and the new distances (e.g. the Γ distance) allow us to classify conveniently all pattern matching automata used for the 2D pattern matching.

Example 3.1
For example, a common problem in text editors is to find a (possibly) misspelled word. That means to use the approximate string matching of one pattern using the Levenshtein distance. This problem can be simply referred to as a *SFODCO* problem.

Instead of single pattern matching problem we can use the notion of family of pattern matching problems. In this case symbol "?" is used instead of a particular letter. For example *SFF*??? describes the family of all problems concerning full pattern matching of multiple strings.

3.1.2 Selected 1D pattern matching automata

In the exact string matching (according to Def. 2.33) we search for all occurrences of pattern $P = p_1 p_2 \cdots p_m$ in the text $T = t_1 t_2 \cdots t_n$, where P is a suffix of the prefix of T (of the text currently read). Let us recall that in this text we use following convention for letters m and n: letter n denotes the length of a text, $n = |T|$, and letter m denotes the length of a pattern, $m = |P|$, and $n \geq m$. More types of finite automata for 1D pattern matching together with methods of their simulation are described thoroughly in Holub's work [Hol00].

Criterion	1	2	3	4	5	6
Abbrev.	S Q	F S	O F I	E R D T G L H	C D	O S

Table 3.1. Pattern matching criteria

3.1.2.1 *Exact string matching of one pattern – SFOECO automaton*

For the exact string matching we construct finite automaton M that accepts language $L(M) = \{wP;\ w \in A^*\}$. In this FA, each state q_i represents a prefix of length i of the pattern that has been found in the text T. (The initial state q_0 represents the trivial empty prefix.) To skip the symbols in front of an occurrence of the pattern P we add a selfloop into the initial state for each symbol of input alphabet A. The construction of such FA is shown in Alg. 3.1, example of its transition diagram is in Fig. 4.3. The resulting automaton has $m + 1$ states.

Algorithm 3.1: Construction of FA for the exact string matching
Input: Pattern $P = p_1p_2 \cdots p_m$, $P \in A^m$, $m \geq 1$.
Output: FA M, $M = (\{q_0, q_1, \ldots, q_m\}, A, \delta, \{q_0\}, \{q_m\})$, accepting language $L(M) = A^*E(P) = \{wP;\ w \in A^*\}$. The mapping δ is constructed as follows.
Method:
$\delta(q_0, a) = \{q_0\}, \forall a \in A$ { the selfloop of the initial state }
$\delta(q_0, p_1) = \{q_0, q_1\}$
for $i = 1$ **to** $m - 1$ **do**
  $\delta(q_i, p_{i+1}) = \{q_{i+1}\}$ { constructs forward transitions for all non-final states }
end

3.1.3 Approximate string matching of one pattern

In the approximate string matching (according to Def. 2.31) we search for all occurrences of pattern $P = p_1p_2 \cdots p_m$ in text $T = t_1t_2 \cdots t_n$, $P \in A^m$, $m > 0$, $T \in A^n$, $n > 0$, considering number k, $k < m$, as the maximum number of errors allowed.

3.1.3.1 *Approximate string matching using the Hamming distance – SFORCO automaton*

Considering Def. 2.35 we construct finite automaton M accepting language $L(M) = \{uv;\ u, v \in A^*, D_H(P, v) \leq k\}$ for the approximate pattern matching using the Hamming distance [Mel95].

Construction of such an automaton consists of $k + 1$ copies of FA for the exact pattern matching, $M_0, M_1, \ldots, M_k$. M_0 represents matching pattern without errors, M_1 with exactly one error and so on up to M_k matching with exactly k errors. These $k+1$ automata are connected by the transitions representing edit operation *replace* (Def. 2.19). Each such transition is labeled by symbol $\bar{p}_{j+1}$, that is complement of symbol p_{j+1} in P (Def. 2.2). This transition increases the minimum number of errors in potential occurrence of P in T by one and leads from state q_j of automaton M_i to state q_{j+1} of automaton M_{i+1}, $0 \leq i < k$, $0 \leq j < m$. The initial state of automaton M_0 is the initial state of the resulting automaton. The transition diagram of FA for the approximate pattern matching using the Hamming distance after removing all inaccessible states is in Fig. 4.4.

Lemma 3.2 (Lemma 3.7 in [Hol00])
The finite automaton for the approximate string matching using the Hamming distance constructed by Alg. 3.2 for a pattern of length m and maximum number of errors k has $(k+1)(m+1-\frac{k}{2})$ states.

Proof
For some number k, $k < |P|$ we have built $k+1$ copies of the FA for the exact pattern matching, constructed by Alg. 3.1. Let i be $0 \leq i \leq k$. From i^{th} FA first i states are removed, because they are inaccessible. Therefore the number of states is determined as follows $(m + 1) + (m + 1 - 1) + (m + 1 - 2) + \cdots + (m + 1 - k) = \sum_{i=0}^{k}(m + 1 - i) = (k + 1)(m + 1 - \frac{k}{2})$. □

Algorithm 3.2: Construction of FA for the approximate string matching using the Hamming distance
Input: Pattern $P = p_1 p_2 \cdots p_m$, $P \in A^m$, $m \geq 1$; and the maximum number of errors allowed $k \in \mathrm{N}$, $k < m$.
Output: FA M, $M = (Q, A, \delta, \{q_0\}, F)$, accepting language $L(M) = \{uv;\ u, v \in A^*, D_H(P, v) \leq k\}$. Sets Q, F and the mapping δ are constructed in the following way:
Method:
$Q = \{q_0, q_1, \ldots, q_{|Q|-1}\},\ |Q| = (k+1)(m+1-\frac{k}{2})$
$l = 0$ { "level" = number of errors }
$r = 1$ { order of symbol in P = "depth" of state }
for $i = 0$ **to** $|Q| - 1$ **do**
  if $r > m$ **then**
    $\delta(q_i, a) = \emptyset,\ \forall a \in A$ { a final state }
    $F = F \cup \{q_i\}$
    $l = l + 1$; $r = l + 1$ { go to the first state of the next level (one more error) }
  else
    $\delta(q_i, p_r) = \{q_{i+1}\}$ { a matching transition }
    if $l < k$ **then**
      $s = i + m + 1 - l$ { s = number of right lower neighbour }
      $\delta(q_i, a) = \{q_s\},\ \forall a \in A \setminus \{p_r\}$ { *replace* transition }
    end
    $r = r + 1$ { go to the next symbol in current level }
  end
end
$\delta(q_0, a) = \delta(q_0, a) \cup \{q_0\},\ \forall a \in A$ { the selfloop of the initial state }

3.1.3.2 *Approximate dictionary matching*

Let Π denotes a dictionary of s strings (patterns), $\Pi = \{p_1, p_2, \ldots, p_s\}$. Analogously to the case of approximate one pattern matching and Def. 2.31, in case of approximate matching of a set of patterns we search for all occurrences of strings from dictionary Π in text T with the maximum number of errors allowed k, $k < m$, in each occurrence of p_i in T, $1 \leq i \leq s$. The limit m denotes the length of the shortest of the patterns to be searched for. Obviously, if m is greater or equal to the length of at least one string in the dictionary, its occurrences will appear at all positions of text T, using any of the distances defined in chapter 2.

3.1.3.3 *Approximate dictionary matching – SFFRCO automaton*

Considering Def. 2.35 we construct s finite automata M_i, $1 \leq i \leq s$, accepting language $L(M_i) = \{uv;\ u, v \in A^*, D_H(p_i, v) \leq k, p_i \in \Pi\}$ for the approximate pattern matching using the Hamming distance, Alg. 3.2.

Approximate dictionary matching with mismatches using finite automaton M is analogous to matching of one string with SFORCO automaton. $M = (Q, A, \delta, \{q_0\}, F)$ is constructed as a union of SFORCO automata M_i, $1 \leq i \leq s$, by Alg. 3.3. The construction is the same for other distances, the only difference is in the distance finite automata M_i search with.

3.1.3.4 *Approximate dictionary matching – automata simulation*

There exists an automata based filter for the approximate dictionary matching. It is described by Baeza-Yates and Navarro in [BYN99] and extends the FA simulation method of [WM92].

The principle of their method is to search for a superposition (*superimposition* in the original paper) of the patterns from a dictionary. The patterns are of the same length, however, they may be aligned using some non-matching symbol, as pointed out in [Smy03]. Since false matches may appear as a result of such a searching, occurrence verification is required as the second step for all patterns involved. Essentially, SFFRCO automaton, with multiple matching

Algorithm 3.3: Construction of a union of finite automata [MHP05, HMU01]
Input: Finite automata M_i, $M_i = (Q_i, A_i, \delta_i, \{q_{0_i}\}, F_i)$, $i \in \mathbf{N}$. Let each M_i accepts language $L(M_i)$. Let all sets of states Q_i be pairwise disjunct.
Output: FA M, $M = (Q, A, \delta, \{q_0\}, F)$, accepting language $L(M) = \bigcup_i L(M_i)$. Sets Q, F and the mapping δ are constructed in the following way:
Method:
$Q = \{q_0\}$
$Q = \bigcup_i Q_i$
$A = \bigcup_i A_i$
$\delta = \bigcup_i \delta_i$
$F = \bigcup_i F_i$
foreach i **do**
  $\delta(q_0, \varepsilon) = \delta(q_0, \varepsilon) \cup \{q_{0_i}\}$
end

transitions, is simulated. That is, the first matching character is one of $p_i[1]$, the second one of $p_i[2]$, and so on, $1 \leq i \leq s$, where s is the cardinality of the dictionary.

The authors described also the limitations of the superposition method. Problems are caused for example by the length of strings (over the computer word length) and by growing number of errors. The proposed solutions to some of the problems are: A division of the patterns to substrings that fit into the computer word. The number of false matches can be reduced by division of the dictionary into several sub-dictionaries and to make individual runs for the strings from them. The criterion for such division is that patterns in one sub-dictionaries share as much of its factors as possible.

The asymptotical time complexity of the filtering and verification is $\mathcal{O}(kmn)$, that is no improvement over the simulation of SFFRCO automaton. Under clearly defined conditions it performs much better, however. The filtering (preprocessing) itself has $\mathcal{O}\left(k\frac{m}{|p_i|-k}\frac{n}{\log n}\right)$. Simplified original results are in [Smy03], as well as the mentioned complexities, where Ukkonen and Myers algorithm [Ukk85a, Mye86] is used for the verification.

3.1.4 Algorithms not based on automata

The previous section devoted to pattern matching using finite automata has described preliminaries of our work. However, there are numerous (one is tempted to say countless) other results in both exact and approximate pattern matching areas. There are books where an interested reader may find some of the vast numbers of results, at least three of them have to be mentioned: Gusfield [Gus97], Crochemore and Rytter [CR02] and Smyth [Smy03].

In this section, only one of many methods based on a many different principles is presented. It is the exact string matching algorithm of Karp and Rabin [KR87]. This algorithm is presented here not only to show an interesting principle, it has a multidimensional variant by the original authors, described briefly in Sec. 3.4.3.2. It is also a base Zhu and Takaoka 2D exact matching algorithm is built upon, as described in Sec. 3.4.3.3.

3.1.4.1 *Exact pattern matching – algorithm of Karp and Rabin*

The algorithm of Karp and Rabin utilizes a fingerprinting (hashing) to search for occurrences of a pattern in a text. Its authors have not been the first to consider the use of fingerprints for string matching. The authors mention the methods based on check sums and hashes have appeared many times in literature, they do not mention any particular, however. In the textbook [CR02], Crochemore and Rytter pointed out Harrison in this context [Har71]. Harrison describes not only the method, he refers in his paper to implementation of the method in an editor that speeded up text searching. He extends his previous work treating a speed-up of operations over ordered sets by hashing techniques [Har70].

The paper deals with pattern matching algorithms based on short representation of preferably long pattern and a substring of a text in which the pattern is searched for. The longer the pattern, the greater the efficiency in comparison to pattern matching algorithms using character to character comparison. (The limit mentioned in the original paper is $|P| = 200$.) Furthermore, the presented approach is on-line, parallelisable, and under reasonable assumptions (constant time of arithmetic operations) also real-time. The authors also describe its extension to more than one dimension, even for non-rectangular shapes. We will return to this extension in Sec. 3.4.3.2.

The following properties are declared as the advantages of the algorithm:

1. The fingerprints are much shorter than the original pattern and text.
2. Is is easy to compute new fingerprint from the old one for updated original string.
3. The probability of a false match is small.

All these properties depend on the choice of appropriate fingerprinting function and this question is studied in detail in the original paper.

In Alg. 3.4 a fingerprint value for each possible m characters long substring of the text is computed. Then this value is compared to the fingerprint value of the pattern.

The function *modulo* is used to compute fingerprints. Hence, hashes and hashing is used to name fingerprints and fingerprinting in the following algorithm description.

The algorithm starts comparing hash value of the pattern with the hash value of substring $T[1..m]$ of T. Then the substring is shifted one symbol to the right, $T[2..m+1]$, its hash is updated, compared to hash of the pattern, and so on.

The hashing function is $h(x)$:

$$h(x) = x \pmod{q}\,, \tag{3.1}$$

q be a large prime number. Clearly, there exist more strings with the same hash value h. Using a large value of prime q makes collisions less probable, the probability of a random collision is $\mathcal{O}\left(\frac{1}{q}\right)$. To avoid false matches, the authors of [KR87] allowed one version (no-real time) of the algorithm to have a buffer of length m of last text characters read. Using this buffer, the algorithm is able to check whether there is an occurrence of P in T at the current position or not. This version is depicted in Alg. 3.4, with modification for alphabets with cardinality ≥ 2.

Both [KR87] and [CR02] assume the binary strings for the sake of simplicity. Let *textHash* be the current hash value of a substring of the text $T[i..i+m-1]$, $T[i+1..i+m]$ be the substring of T shifted by one symbol to the right, and $b = 2^{m-1} \pmod{q}$. The following formula evaluates $h(x)$ from formula 3.1 above for shifted $T[i+1..i+m]$.

$$2(\mathit{textHash} - T[i]b) + T[i+m] \tag{3.2}$$

Initialisation is done by evaluation of $\mathit{patternHash} = P[1..m] \pmod{q}$, $\mathit{textHash} = T[1..m] \pmod{q}$.

In [ZT89], the hash computation for strings over alphabets with cardinality ≥ 2 is presented. The symbols are packed in a computer word and treated as an integer. It means to write the symbols as numbers in a radix-d number system, where $d = |A|$ denotes the cardinality of ordered alphabet A and is the maximal value its symbols can have. The formula for evaluating x in the formula 3.1 is:

$$x = \mathrm{ord}(T[i])d^{m-1} + \mathrm{ord}(T[i+1])d^{m-2} + \cdots + \mathrm{ord}(T[i+m-1])\,.$$

Hash update is of course analogous to the version for binary strings given in formula 3.2:

$$d(\mathit{textHash} - \mathrm{ord}(T[i]d^{m-1})) + \mathrm{ord}(T[i+m])\,.$$

Algorithm 3.4: Algorithm of Karp and Rabin for 1D pattern matching
Input: Pattern $P \in A^m$, text $T \in A^n$, $1 \leq m \leq n$, d be the maximal value of all symbols of the alphabet, q be a large prime.
Output: Report of positions where pattern P is found.
Method:
$patternHash = 0$
$textHash = 0$
for $i = 1$ **to** m **do**
  $patternHash = (d \cdot patternHash + \text{ord}(P[i])) \pmod q$
end
for $i = 1$ **to** m **do**
  $textHash = (d \cdot textHash + \text{ord}(T[i])) \pmod q$
end
$b = 1$
for $i = 1$ **to** $m - 1$ **do**
  $b = (d \cdot b) \pmod q$
end
for $i = 1$ **to** $n - m$ **do**
  if $textHash = patternHash$ **then**
    if $T[i..i + m - 1] = P$ **then print** "P occurs at position i"
  end
  { Added $d \cdot q$ makes the result always positive. }
  $textHash = (textHash + d \cdot q - \text{ord}(T[i]) \cdot b) \pmod q$
  $textHash = (d \cdot textHash + \text{ord}(T[i + m])) \pmod q$
end

The algorithm works in $\mathcal{O}(n + m)$ expected time, the worst case $\mathcal{O}(nm)$ is reached in case potential occurrences need to be checked in each step (e.g. $P = a^m$, $T = a^n$).

3.1.5 Text indexing

In the previous sections selected pattern matching methods were presented. All of them are based on matching of a pattern in a text. There exists another principle of solving such a problem. The idea is to preprocess the text and construct an index for it. Then the pattern is to be searched in the index. Let us recall two definitions used in descriptions of indexing methods, suffix (Def. 2.17) and factor (Def. 2.16).

Text indexing is commonly used in situation where many patterns shall be tested against a fixed text (e.g. in biological or other database oriented applications).

Overview of structures used for (one-dimensional) text indexing
Different kinds of index data structures have been presented, these are:
- suffix trees [Wei73, McC76, Ukk95, Far97],
- suffix arrays [MM93],
- suffix automata [BBE$^+$83, BBE$^+$85, Cro86],
- factor automata [Cro86].

A survey of all these structures except the factor automata can be found in Smyth [Smy03], similarly in Gusfield [Gus97]. In Crochemore and Rytter [CR02, Chapters 4–6], there are treated suffix trees and DAWG, and there is a reference to [Cro86] and factor automata. Suffix and factor automata are covered in detail in [MHP05].

The oldest powerful text indexing structure appears to be a PATRICIA tree [Mor68]. It is a binary tree that stores a set of distinct strings over binary alphabet. Each of its internal nodes is labeled with so called *skip value*, an integer indicating a position of the branching symbol while descending towards leafs of the tree. The left arcs are implicitly labeled with

one symbol of the binary alphabet, the right arcs are implicitly labeled with the other symbol. Searching for a pattern of length m takes $\mathcal{O}(m)$ time and retrieves only the suffix pointer in two leaves, i.e. the leaf reached by branching with the skip values, and the leaf corresponding to an occurrence. Returning a pointer to a suffix in the text requires $\mathcal{O}(1)$ suffix lookups in the worst case. A PATRICIA tree storing s suffixes needs $\mathcal{O}(s \log n)$ bits. Suffix trees are often implemented by building a PATRICIA tree on the suffixes of the text [GBYS92].

A suffix tree is a compacted trie representing all suffixes of a text string, it is a space efficient version of Weiner's position tree [Wei73]. A suffix tree for text T of length n over alphabet A can be built in $\mathcal{O}(n \log |A|)$ time [McC76, Ukk95]. A pattern of length m can be then searched in $\mathcal{O}(m \log |A|)$ time. Applications of suffix trees are not limited to pattern matching, they are used in many string processing applications, as described in surveys in [CR02, Gus97, Smy03].

A suffix array has been described by Manber and Myers [MM93]. It is a sorted list of all suffixes of a text string and can be constructed in $\mathcal{O}(n \log n)$ time [MM93, Smy03]. These methods require $\mathcal{O}(n \log n)$ space to store a suffix array. Works [FM00, GV00, Sad00] have reduced this space to $\mathcal{O}(n)$ while a searching time has been increased by $\mathcal{O}(\log n)$ factor. (Journal version of [GV00] is [GV05].) For very long texts such are DNA sequences not only the resulting space complexity is important. The methods of a suffix array construction already mentioned require at least $\mathcal{O}(n \log n)$ space during their construction phases. Lam *et al.* in [LSSY02] considered a space complexity of the construction phase and they presented an algorithm that works in $\mathcal{O}(n \log n)$ time as the other methods and uses only $\mathcal{O}(n)$ space during construction.

A suffix automaton accepting all suffixes of a text is from Blumer *et al.* [BBE$^+$85] and Crochemore [Cro85, Cro86]. Blumer *et al.* have extended their previous finding, the Directed Acyclic Word Graph, achieved in [BBE$^+$83]. A survey can be found in [CR02, Smy03].

A factor automaton, the minimal automaton accepting the set of all factors of a given text and its construction is from [Cro85].

The suffix or the factor automaton can be constructed for the text in time linear in n, where n is the size of a text. The constructed suffix or factor automaton represents a complete index of the text for all possible suffixes or factors, respectively. It can find all occurrences of a text suffix or a text factor, respectively, and indicate their positions in the text.

The main advantages of this type of finite automata are: For given input text suffix or text factor of size m (the pattern), the suffix or the factor automaton, respectively, performs its search phase in time linear in m and not depending on n.

Although the number of possible factors in the text can be quadratic in n, the total size of the suffix or the factor automaton is linear in n.

There also exists a compacted version, CDAWG, with its transitions labeled by strings and not only by individual symbols, mentioned e.g. in [CR02]. Comparison of the DAWG and the suffix array from the practical performance point of view is e.g. in [Bal03].

3.2 Tree Algorithms

Many authors addressed problems related to trees and tree and subtree pattern matching up to this time. In the following, we will describe only those parts closely matching our needs or related to our approach. For a complete information on the topic an interested reader may see a monography [CDG$^+$07], or the dissertation thesis [Cle08] for a thorough survey and a different, taxonomy oriented, point of view.

3.2.1 Tree Pattern Matching

Tree pattern matching is a problem of finding all occurrences and their positions of tree patterns in a subject tree. Numerous methods, described in [CDG+07] and [Cle08, chapters 5 and 6], and referenced in [JM09, Jan09, FJM09, JM10] use an approach that preprocesses tree patterns and the search phase is performed in time linear to the size of a tree at best. Some of the methods mentioned also preprocess the subject tree, none of them does the preprocessing analogous to the string indexing, that is preprocessing of the tree itself.

There are some important properties of trees in the prefix notation.

Lemma 3.3 ([JM10])
Given an ordered tree t and its prefix notation $\mathrm{pref}(t)$, all subtrees of t in the prefix notation are substrings of $\mathrm{pref}(t)$.

Proof
By induction on the height of the subtree:

1. If subtree t' has only one node a, $\mathrm{arity}(a) = 0$, then $\mathrm{height}(t') = 0$, $\mathrm{pref}(t') = a$ and the statement holds for such subtree.
2. Assume the statement holds for subtrees $t_1, t_2, \ldots, t_p$, where $p \geq 1$, $\mathrm{height}(t_1) \leq m$, $\mathrm{height}(t_2) \leq m$, …, $\mathrm{height}(t_p) \leq m$, $m \geq 0$.
   It is to be proven that the statement also holds for each subtree $t' = at_1t_2 \cdots t_p$, where $\mathrm{arity}(a) = p$, $\mathrm{height}(t') = m + 1$.
   As $\mathrm{pref}(t') = a\,\mathrm{pref}(t_1)\,\mathrm{pref}(t_2) \cdots \mathrm{pref}(t_p)$, the statement holds for subtree t'.

The lemma holds. □

Not every substring of a tree in the prefix notation is a prefix notation of its subtree. This can be seen from the fact that for a given tree with n nodes there can be $\mathcal{O}(n^2)$ distinct substrings, but there are just n subtrees. Each node of the tree is the root of just one subtree. Only those substrings which themselves are trees in the prefix notation are those which are the subtrees in the prefix notation. This property is formalised in [JM10, Theorem 2].

Algorithm 3.5: Arity checksum of a string over a ranked alphabet
Input: Let $w = a_1a_2 \cdots a_m$, $a_i \in A$, $i = 1, \ldots, m$, $m \geq 1$, be a string over ranked alphabet A.
Output: Arity checksum $\mathrm{ac}(w)$ of string w.
Method:
$\mathrm{ac}(w) = \mathrm{arity}(a_1) + \mathrm{arity}(a_2) + \cdots + \mathrm{arity}(a_m) - m + 1 = \sum_{i=1}^{m} \mathrm{arity}(a_i) - m + 1\,.$

Lemma 3.4 ([JM10], Theorem 2)
Let $\mathrm{pref}(t)$ be the prefix notation of tree t, w be a substring of $\mathrm{pref}(t)$. Then, w is the prefix notation of a subtree of t if and only if $\mathrm{ac}(w) = 0$, and $\mathrm{ac}(w_1) \geq 1$ for each w_1 where $w = w_1x$, $x \neq \varepsilon$.

Proof
Let st_1 and st_2 be two subtrees of tree t. It holds that $\mathrm{pref}(st_1)$ and $\mathrm{pref}(st_2)$ are either two different strings or one is a substring of the other. The former case occurs if the subtrees st_1 and st_2 are two different trees with no shared part and the latter case occurs if one tree is a subtree of the other tree. No partial overlapping of subtrees is possible in ranked ordered trees. Moreover, it holds for any two subtrees which are adjacent ancestors of the same node (adjacent siblings) that their prefix notations are two adjacent substrings in the prefix notation of the tree.
If: By induction on the height of a subtree st, where $w = \mathrm{pref}(st)$.

1. Assume that $\mathrm{height}(st) = 0$. It means that $w = a$, where $\mathrm{arity}(a) = 0$, a is a nullary symbol. Then $\mathrm{ac}(w) = 0$. Thus the statement holds for the case where $\mathrm{height}(st) = 0$.

2. Assume the statement holds for subtrees $st_1, st_2, \ldots, st_p$, where $p \geq 1$, $\text{height}(st_1) \leq n, \text{height}(st_2) \leq n, \ldots, \text{height}(st_p) \leq n$, and $\text{ac}(\text{pref}(st_1)) = 0, \text{ac}(\text{pref}(st_2)) = 0, \ldots, \text{ac}(\text{pref}(st_p)) = 0$.
   It is to be proven that the statement also holds for subtree w of height $n+1$.

   Assume $w = a\,\text{pref}(st_1)\,\text{pref}(st_2)\cdots\text{pref}(st_p)$, where $\text{arity}(a) = p$. Then $\text{ac}(w) = p + \text{ac}(\text{pref}(st_1)) + \text{ac}(\text{pref}(st_2)) + \cdots + \text{ac}(\text{pref}(st_p)) - (p+1) + 1 = 0$ and $\text{ac}(w_1) \geq 1$ for each w_1, where $w = w_1 x$, $x \neq \varepsilon$.
   Thus the statement holds for the case where $\text{height}(st) = n+1$.

Only if: Assume $\text{ac}(w) = 0$ and $w = a_1 a_2 \cdots a_m$, where $m \geq 1$, $\text{arity}(a_1) = p$. Since $\text{ac}(w_1) \geq 1$ for each w_1, where $w = w_1 x$, $x \neq \varepsilon$, none of the substrings w_1 can be a subtree in prefix notation. This means that the only possibility for $\text{ac}(w) = 0$ is that w is of the form $w = a\,\text{pref}(t_1)\,\text{pref}(t_2)\cdots\text{pref}(t_p)$, where $p \geq 0$ and $t_1, t_2, \ldots, t_p$ are subtrees which are adjacent ancestors of the same node (adjacent siblings). In such a case $\text{ac}(w) = p + 0 - (p+1) + 1 = 0$. No other possibility of the form of w for $\text{ac}(w) = 0$ is possible and the claim holds.

Therefore the statement of the lemma holds. □

In tree pattern and subtree pushdown automata the arity checksum is computed by pushdown operations. The content of the pushdown store represents the corresponding arity checksum. The empty pushdown store means that the corresponding arity checksum is equal to zero. Such pushdown operations correspond to the pushdown operations of the standard top-down parsing algorithm (Aho and Ullman [AU71]) for a context-free grammar with rules of the form $S \to aS^{\text{arity}(a)}$. Moreover, these pushdown operations make the corresponding pushdown automaton input-driven (Def. 2.61), e.g. PDA accepting subtrees in the prefix notation [Jan09].

3.2.2 Prefix/Suffix/Euler notation

The first two algorithms for prefix and suffix notation are commonly used. The construction of the Euler notation has no direct use in this work and is listed for completeness.

Algorithm 3.6: Prefix notation of a tree
Input: Let $T = (V, E)$ be a rooted labeled ordered ranked directed tree. Let r, $r \in V$, be a root of T. Let descendant(v) be a function returning an ordered array of direct descendants of $v \in V$.
Output: Tree T in prefix notation.
Description: Algorithm begins processing of T with $node = r$.
Method:

```
print (getlabel(node))
if outdeg(node) ≠ 0 then                          { node is not a leaf }
   for i = 1 to |descendant(node)| do
      node = descendant(node)[i]
      call Alg. 3.6
   end
end
return
```

Algorithm 3.7: Postfix notation of a tree
Input: Let $T = (V, E)$ be a rooted labeled ordered ranked directed tree. Let r, $r \in V$, be a root of T. Let descendant(v) be a function returning an ordered array of direct descendants of $v \in V$.
Output: Tree T in postfix notation.
Description: Algorithm begins processing of T with $node = r$.
Method:

```
if outdeg(node) ≠ 0 then                                  { node is not a leaf }
   for i = 1 to |descendant(node)| do
      node = descendant(node)[i]
      call Alg. 3.7
   end
end
print (getlabel(node))
return
```

Algorithm 3.8: Euler notation of a tree
Input: Let $T = (V, E)$ be a rooted labeled ordered ranked directed tree. Let r, $r \in V$, be a root of T. Let descendant(v) be a function returning an ordered array of direct descendants of $v \in V$.
Output: Tree T in Euler notation.
Description: Algorithm begins processing of T with $node = r$.
Method:

```
print (getlabel(node))
if outdeg(node) ≠ 0 then                                  { node is not a leaf }
   for i = 1 to |descendant(node)| do
      node = descendant(node)[i]
      call Alg. 3.8
      print (getlabel(node))
   end
end
return
```

3.3 Directed acyclic graph (DAG), Newick notation

The biological applications of trees encountered difficulties with extra edges between nodes of two subtrees of a tree. The purpose of such edges is to describe some extra biological relations between two subtrees in question, e.g. a transfer of genes between species, denoted as nodes of a phylogenetic tree, as described below. This serves as motivation to move on from description of trees to directed acyclic graphs.

Phylogenetics deals with evolutionary relationships among nucleotide sequences, genes, chromosomes, genomes or species [CRV08]. For this purpose, it uses phylogenetic trees to describe and represent evolutionary histories under mutation. Phylogenies reveal the history of evolutionary events of a group of species, and they are central to comparative analysis methods for testing hypotheses in evolutionary biology [Pag99, CLRV08].

The standard textual description of phylogenetic trees is the Newick notation [Ols90, Fel04], adopted June 26, 1986 by an informal committee meeting at Newick's seafood restaurant in Dover, New Hampshire, USA, during the Society for the Study of Evolution meeting in Durham, New Hampshire. The Newick format is the de facto standard for representing phylogenetic trees. It is quite convenient as it allows to describe a whole phylogenetic tree in linear form in a unique

way, once the phylogenetic tree is drawn or the ordering among children nodes is fixed. The Newick description of a phylogenetic tree is a string of nested parentheses annotated with taxa names and possibly also with branch lengths or other values, obtained by traversing the phylogenetic tree in postorder (Alg. 3.7) and following some simple rules that allow parsing a Newick string into the corresponding phylogenetic tree, and vice versa. As noticed in [CRV08], almost every phylogenetic software tool includes an option to export phylogenetic trees in Newick format, and open-source code for parsing Newick strings is readily available in a number of software toolkits, e.g. BioPerl [SBB+02].

In the Newick format, results of molecular phylogenetic analysis are available from specialised databases [Mor96].

It has been shown that the existence of genetic recombinations, hybridisation and lateral gene transfer makes species evolve more in a reticulate way than in a simple, arborescent way [Doo99, CLRV08, CRV09]. The presence of such reticulate evolutionary events in phylogenies turn phylogenetic trees into phylogenetic networks. These events imply that there may exist multiple evolutionary paths from a non-extant species to an extant one, and this multiplicity makes the comparison of phylogenetic networks much more difficult than the comparison of phylogenetic trees [CLRV08].

Phylogenetic networks differ from phylogenetic trees in explicit modeling of reticulate evolutionary events such as already mentioned recombinations, hybridisation or lateral gene transfer [CRV08]. This is achieved by means of hybrid nodes which differ from tree nodes by in-degree greater than 1.

Adaptation of Newick description to special relations that turn phylogenetic trees into phylogenetic networks is possible. Two versions were proposed: a single Newick string (where each hybrid node is splitted once for each parent) or a set of Newick strings (one for each hybrid node plus another one for the phylogenetic network) [CRV09]. [CRV08] states the former one has been adopted as a standard extended Newick format, which describes a whole phylogenetic network with k hybrid nodes as a single Newick string with k repeated nodes. The authors of [CRV09] also note that such a representation is unique once the phylogenetic network is drawn or the ordering among children nodes is fixed.

We were unable to find any notice of gene backpropagation in the phylogenetic tree in the biological literature available to us, so we consider these phylogenetic networks to be *dags* (Def. 2.76), i.e. they do not contain cycles.

3.4 Two-dimensional pattern matching

The interest in the two-dimensional pattern matching can be traced back to the seventies of the past century. First results concerning 2D exact pattern matching problems were found independently by Bird and Baker in 1977–8 [Bir77, Bak78].

Many types of algorithms have appeared during the years of intensive research in this area: those based on reduction of the problem into one dimension [Bir77, Bak78], convolution [FP74] and even pure 2D approach based on the 2D periodicity [ABF92, GP92].

Amir, Benson and Farach [ABF92] (with preprocessing given by Galil and Park [GP92]) proposed a linear worst-case time algorithm that is alphabet-independent in the same sense as the KMP algorithm [KMP77] in 1D pattern matching. Interesting results concerning sublinear expected running time were proposed by Baeza-Yates and Régnier [BYR93]. Their algorithm finds an $(m \times m)$ pattern in an $(n \times n)$ text in expected time $\mathcal{O}(n^2/m)$. Kärkkäinen and Ukkonen [KU94] found an algorithm that finds exact occurrences in expected time $\mathcal{O}\left(\frac{n^2}{m^2} \log_{|A|} m^2\right)$, which is optimal. ($A$ denotes the alphabet.)

3.4.1 Content based image retrieval

Besides methods belonging to the area of combinatorial pattern matching, described in this chapter, there are numerous methods used in Computer Graphics and Computer Vision. We will not describe them here, the purpose of this short section is just to point to their existence.

Some of the methods may be used as a preprocessing of the picture for further processing using methods of combinatorial pattern matching. For a survey of recursive picture decomposition and a content based image retrieval see [RMJ06] and the numerous references in there.

In the following, let us turn our attention fully to the area of combinatorial pattern matching methods.

3.4.2 Two-dimensional languages and cellular automata

A theory of two-dimensional languages was originated by Giammarresi and Restivo [GR97].

At about the same time with the publication of [Bir77, Bak78], Inoue and Nakamura presented new applications of *tessellation acceptors* (cellular computing system working in multiple dimensions) in the 2D pattern matching [IN77].

A review of topics related to these systems and their applications is collected in Polcar's work [Pol04]. His work also mentions some 2D-specific distances [KS87, BY98] not used here (e.g. KS, R, C, L, RC).

A study of some aspects of picture languages and related models with references to related literature can be found in [Prů04]. The topics discussed there include picture languages, two-dimensional finite-state, on-line tessellation, and forgetting automata, and grammars with productions in context-free form.

3.4.3 2D exact pattern matching algorithms

In the following sections on-line exact two-dimensional pattern matching algorithms are presented, those cited most often in the literature were selected.

3.4.3.1 *2D exact pattern matching – algorithm of Baker and Bird*

The solution of the two-dimensional exact pattern matching done by Baker and Bird is cited in almost every paper about this topic.

A pattern array may be viewed as a sequence of strings, for example its columns. To locate columns of the pattern array within columns of a text array searching for several strings is required (Fig. 4.1). The Bird and Baker's solution uses well-known *Aho-Corasick* algorithm [AC75] of pattern matching for a set of patterns. The use of the AC pattern matching machine has significant advantage: it is deterministic. This allows construction of associated output actions in every state of the AC pattern matching machine and therefore direct conversion of text array TA to TA' by writing active state identifiers into the original text array TA.

Let us show an example of the idea presented above. First, let PA, TA be pattern and text array, respectively.

$$PA = \begin{array}{|c|c|c|}\hline a & c & a \\ \hline b & b & a \\ \hline c & a & b \\ \hline \end{array}\ ,\ TA = \begin{array}{|c|c|c|c|c|c|c|}\hline b & b & a & b & b & a & b \\ \hline a & a & c & a & c & b & a \\ \hline b & b & b & a & c & a & c \\ \hline a & c & a & b & b & a & b \\ \hline c & a & a & c & a & b & a \\ \hline b & b & b & b & a & c & c \\ \hline a & c & c & a & b & a & b \\ \hline \end{array}\ ,\ |PA| = (3 \times 3),\ |TA| = (7 \times 7).$$

An example of the AC pattern matching machine for searching three strings of a pattern array of size (3×3) is figured in Fig. 3.2. The first column containing string "*abc*" is related

to final state 5; the second column containing string "*cba*" is related to final state 8; the last column of "*aab*" to final state 3.

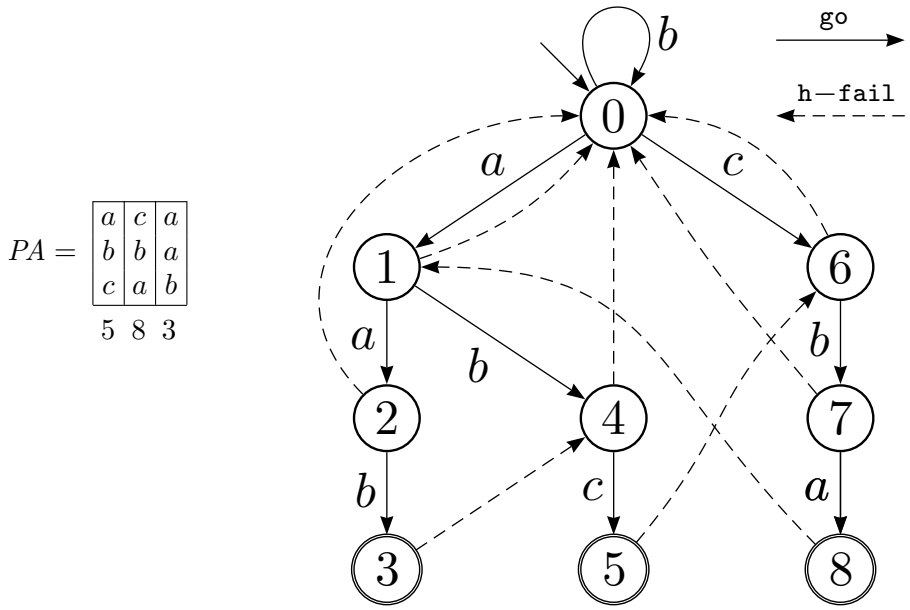


Figure 3.2. The transition diagram of an AC pattern matching machine for the example pattern array

Once array TA' is generated, the bottom row of PA have to be located (i.e. the last string, when we read PA horizontally) within array TA'. Original text array TA and resulting array TA' after processing by AC pattern matching machine is in Fig. 3.3.

$$TA = \begin{array}{|c|c|c|c|c|c|c|} \hline b & b & a & b & b & a & b \\ \hline a & a & c & a & c & b & a \\ \hline b & b & b & a & c & a & c \\ \hline a & c & a & b & b & a & b \\ \hline c & a & a & c & a & b & a \\ \hline b & b & b & b & a & c & c \\ \hline a & c & c & a & b & a & b \\ \hline \end{array} \qquad TA' = \begin{array}{|c|c|c|c|c|c|c|} \hline 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ \hline 1 & 1 & 6 & 1 & 6 & 4 & 1 \\ \hline 4 & 4 & 7 & 2 & 6 & 1 & 6 \\ \hline 1 & 5 & 8 & 3 & 7 & 2 & 7 \\ \hline 6 & 1 & 1 & 5 & 8 & 3 & 8 \\ \hline 7 & 4 & 4 & 7 & 2 & 5 & 6 \\ \hline 8 & 5 & 5 & 8 & 3 & 1 & 7 \\ \hline \end{array}$$

Figure 3.3. Text array TA before and after preprocessing based on the run of the AC pattern matching machine in columns

To search for a string of identifiers of final states associated with pattern array PA a string matching algorithm is needed, in this case the authors used the *Knuth-Morris-Pratt* algorithm [KMP77].

Set of final states F of AC automaton from example in Fig. 3.2 is $F = \{3, 5, 8\}$. Recall that columns have final states labeled 5, 8, 3, in that order. Occurrences of string "583" in the preprocessed array of states TA' gives 2D occurrences of the pattern array in the original text array TA. Result of matching for the pattern array is in Fig. 3.4.

	a	c	a			
	b	b	**a**	**c**	**a**	
	c	a	**b**	**b**	**a**	
		a	**c**	**a**	**b**	
		b	*b*	*a*		
		c	*a*	*b*		

0	0	1	0	0	1	0
1	1	6	1	6	4	1
4	4	7	2	6	1	6
1	5	8	3	7	2	7
6	1	1	**5**	**8**	**3**	8
7	4	4	7	2	5	6
8	5	*5*	*8*	*3*	1	7

Figure 3.4. 2D occurrences of pattern array *PA* in *TA* (left) and the relation to the result of *KMP* pattern matching in array TA' of states of the AC pattern matching machine (right)

Let A be a finite alphabet, *TA* be a text array of size $(n \times n')$ and *PA* be a pattern array of size $(m \times m')$. Then the presented method of the two-dimensional exact pattern matching has $\mathcal{O}(mm' + nn')$ asymptotic time complexity [Bir77, Bak78].

3.4.3.2 *2D exact pattern matching – algorithm of Karp and Rabin*

Let us briefly return to discussion of the Karp and Rabin algorithm in Sec. 3.1.4.1. As already mentioned, in [KR87] the authors proposed also a method for multidimensional pattern matching. It is based on the linearisation of particular block of multidimensional text and computation of a fingerprint value of this block, now treated as a string. Then comparison of the resulting value with the fingerprint value of a pattern linearised in the same manner answers the question whether at this location there is a possible match or not. As with the one-dimensional case, to be sure a 2D occurrence is really there, symbol by symbol check is to be used. Using elaborate linearisation, they are able to match even non-rectangular shapes.

3.4.3.3 *2D exact pattern matching – algorithm of Zhu and Takaoka*

In this algorithm, Zhu and Takaoka [ZT89] reused the Karp and Rabin algorithm for one-dimensional pattern matching, described in Sec. 3.1.4.1. They avoided linearisation of the multidimensional structures, mentioned above in Sec. 3.4.3.2. They proceed in a manner that is common for many 2D pattern matching algorithms. They use some 1D pattern matching algorithm along one dimension of the 2D text and then do another matching in the other dimension of this preprocessed 2D text.

The idea of algorithm is to use the same hash function as in Karp and Rabin algorithm along one dimension of text and pattern array. The resulting two-dimensional arrays of numbers are then searched for the numbers of the preprocessed pattern using either KMP [KMP77] or Boyer-Moore algorithm [BM77].

Algorithm 3.9: Algorithm of Zhu and Takaoka
Input: Pattern array $PA \in A^{(m \times m')}$, text array $TA \in A^{(n \times n')}$, $1 \leq m \leq n$, $1 \leq m' \leq n'$, d be the maximal value of all symbols of the alphabet, q be a large prime, procedures and data structures defined in Alg. 3.10.
Output: Reports two-dimensional occurrences of PA in TA.
Description: Hashing is done in columns, then KMP or BM (procedure 1Dsearch) finds occurrences of string P' in string T' and found potential 2D occurrences are then verified in TA individually.
Method:

```
b = 1
for i = 1 to m − 1 do
   b = (d · b) (mod q)
end
patternhash()
texthash()
for row = m′ to n' do
   if 1Dsearch(T′, P′, columns) = true then
      if columns is not empty then
         foreach column in columns do
            if directCompare(column − m + 1, row − m′ + 1)=true then
               print "PA found at [column − m + 1, row − m′ + 1]"
            end
         end
      end
   end
   if row < n′ then change(row)
end
```

The principle is similar to the Bird and Baker algorithm, strings made of columns or rows of a pattern are searched in columns or rows of the text, respectively, using Karp and Rabin algorithm to compute new representation of the text, a string consisting of hashes. KMP or BM algorithm serve for matching in the preprocessed string of hashes to search for another string of hashes representing the pattern.

Algorithm 3.9 presents the version with optimised space complexity.

The time complexity of Alg. 3.9 is $\mathcal{O}(N_{PA} + N_{TA})$, where N_{PA} is the size of the 2D pattern and N_{TA} is the size of the 2D text. $\mathcal{O}(N_{PA} + N_{TA})$ takes the hash function calculation, the largest number of comparisons made by procedure directCompare, implemented as the KMP search, is $\mathcal{O}((n' - m')n)$. The space complexity of Alg. 3.9 is $\mathcal{O}(N_{PA} + N_{TA})$.

Algorithm 3.10: Algorithm of Zhu and Takaoka – procedures
Input: Pattern array $PA \in A^{(m \times m')}$, text array $TA \in A^{(n \times n')}$, $1 \leq m \leq n$, $1 \leq m' \leq n'$, d be the maximal value of all symbols of the alphabet, q be a large prime.
Output: Provides necessary procedures for the main algorithm.
Description: These are the procedures used in the main algorithm 3.9. Hashing is done in columns, P', T' are the preprocessed (hashed) strings. 2D occurrences are verified using the original arrays.
Method:

```
procedure patternhash()
    for j = 1 to m do
        P'[j] = 0
        for i = 1 to m' do
            P'[j] = (P'[j] · d + ord(PA[i, j])) (mod q)
        end
    end
end
procedure texthash()
    for j = 1 to n do
        T'[j] = 0
        for i = 1 to m' do
            T'[j] = (T'[j] · d + ord(TA[i, j])) (mod q)
        end
    end
end
procedure change(row)
    for j = 1 to n do
        T'[j] = (T'[j] + d · q − b · ord(TA[j, row − m' + 1])) · d + ord(TA[j, row + 1]) (mod q)
    end
end
function directCompare(column, row): boolean
    j = 1; found = true
    while (j ≤ m') ∧ (found = true ) do
        i = 1
        while (i ≤ m) ∧ (found = true ) do
            if PA[i, j] ≠ TA[column + i − 1, row + j − 1] then found = false
            i = i + 1
        end
        j = j + 1
    end
end
```

3.4.3.4 *2D exact pattern matching – algorithm of Baeza-Yates and Régnier*

The algorithm of Baeza-Yates and Régnier [BYR93] uses dictionary searching to locate a two-dimensional pattern in a two-dimensional text. The idea is to treat the pattern as a dictionary of ordinary strings and perform a search in only n'/m' rows of the text to find all candidate positions. Obviously, no pattern occurrence will be missed. Suppose there is a match of one of the strings from the dictionary, that is one complete row, say l^{th}, of the two-dimensional pattern has been found at position x, y, where $m \leq x \leq n$, $m' \leq y \leq n'$. Then the $m' - l$ rows above and $l - 1$ rows below the current row in the two-dimensional text are to be searched for a 2D occurrence, starting from the column $x - m + 1$. In the algorithm, the search is performed using Aho-Corasick algorithm, similarly to Bird and Baker algorithm.

Algorithm 3.11: Algorithm of Baeza-Yates and Régnier
Input: Pattern array $PA \in A^{(m \times m')}$, text array $TA \in A^{(n \times n')}$, $1 \leq m \leq n$, $1 \leq m' \leq n'$.
Output: Report of all occurrences of PA in TA.
Method:

```
for row = 1 to m' do
   addString(dictionary, PA[1..m, row])
end
createAC(dictionary)
nextline = false
for row = m' − 1 to row ≤ n' step m' do
   for i = 1 to i ≤ m do
      if ACsearch(row, i) then
         for j = 1 to j < i do
            if not ACsearch(row − i + j, j) then nextline = true ; break
         end
         for j = 1 to j ≤ m − i + 1 do
            if not ACsearch(row + j, j) then nextline = true ; break
         end
         if not nextline then printMatch(i, j)
                         else nextline = false
      else
         break
      end
   end
end
```

3.4.4 2D approximate pattern matching

A 2D approximate pattern matching usually considers only substitutions for rectangular patterns, which is much simpler than the general case with insertions and deletions. Moreover, it is very analogous to the approximate matching with k mismatches known from the 1D case (using the Hamming distance, Def. 2.23).

A 2D pattern matching with k mismatches (or equivalently using the 2D Hamming distance k, $k \in \mathrm{N}$) means to find all occurrences of PA in TA with equal or less than k mismatching symbols. There exist several algorithms solving this problem: in 1987 Krithivasan and Sitalakshmi [KS87] achieved $\mathcal{O}\left(km\left(m' \log m' + mm'/k + nn'\right)\right)$ time complexity; Amir and Landau [AL91] $\mathcal{O}\left(\left(k + \log |A|\right) n^2\right)$ time and $\mathcal{O}\left(n^2\right)$ space. A similar solution is given by Crochemore and Rytter in [CR94]. Another solution proposed by Ranka and Heywood [RH91] has lower space requirements $\mathcal{O}(kn)$ but works in time $\mathcal{O}\left(\left(k + a\right)\left(b \log b + n^2\right)\right)$, where $a = \min(m, m')$ and $b = \max(m, m')$, this result is asymptotically optimal for $k \approx a$.

The best expected time for this type of the 2D pattern matching has algorithm of Kärkkäinen and Ukkonen [KU94]: $\mathcal{O}\left(\frac{n^2}{m^2}(k+1)\log_{|A|} m^2\right)$, provided that $k \leq \frac{m^2}{4\left\lceil \log_{|A|} m^2 \right\rceil}$, and a 2D text has size $(n \times n)$, a 2D pattern $(m \times m)$. Both exact and approximate algorithms of [KU94] need preprocessing of P which takes time and space $\mathcal{O}\left(m^2 \log_{|A|} m^2\right)$ or $\mathcal{O}\left(m^2\right)$. Another solution based on filtering algorithms was proposed by Baeza-Yates and Navarro [BYN00]. They presented some new 2D edit distances (C, L, R, RC) and their filters are constructed to use them. As a simplification of these filters the authors presented one for the 2D matching with mismatches. It works in average time $\mathcal{O}(\frac{kn^2}{m})$, supposed that $k \leq \frac{m}{2\log_{|A|} m}(1 + o(1))$.

3.5 Representation of multidimensional matrices for indexing

There are several methods of multidimensional matrices representation for indexing in the literature.

Methods described are based on:

1. Vertex splitting, where elements of an array may be used multiple times in the representing data structure.
2. Tree representation (prefix/suffix linearisation), where suffix trees and suffix arrays are constructed from carefully chosen submatrices of the subject matrix.
3. Compact representation, a labeled, oriented matrix graph.

3.5.1 Two-dimensional structures

As noted e.g. in Giancarlo and Grossi [GG97b], the first indexing data structure for two dimensional arrays has been proposed by Gonnet [Gon88]. He first introduced a notion of suffix trees for a picture (a PAT-tree) using a decomposition into a spiral string. Giancarlo [Gia95] presented the *L-suffix tree* as a generalization of the suffix tree to square matrices and obtained a two-dimensional suffix array from L-suffix trees. L-string equivalent notion has been described independently by Amir and Farach [AF92]. The time complexity of the construction is $\mathcal{O}(n^2 \log n)$, using $\mathcal{O}(n^2)$ space for an array of $(n \times n)$. The same time complexity is reached also in [KKP03]. Na, Giancarlo and Park [NGP07] presented on-line version with the same construction time complexity, $\mathcal{O}(n^2 \log n)$, optimal for unbounded alphabets.

Kim, Kim and Park [KKP98, KKP03] presented an indexing structure for two-dimensional arrays (*I-suffix*), extensible to higher dimensions. They used a *Z-suffix* structure for indexing in three dimensional space.

The only limitation is the arrays need to be square and cubic, respectively, i.e. $(n \times n)$ and $(n \times n \times n)$, respectively. The presented $\mathcal{O}(n^2 \log n)$ time construction algorithm for direct construction of I-suffix arrays and $\mathcal{O}(n^3 \log n)$ time construction algorithm for Z-suffix arrays. This is the first algorithm for three-dimensional index data structure.

Giancarlo and Grossi [GG96, GG97b], besides they gave an expected linear-time construction algorithm for square pictures, presented the general framework of two-dimensional suffix tree families. E.g. [GG97b, Def. 10.18] defines an index for a rectangular array as a rooted tree whose arcs are labeled by *block characters*. A block character is in general some sub-picture of the given picture, its shape need not be rectangular (Note 2.87). Similarly to tries used in string processing, these three conditions hold for such an index:

1. No two arcs leaving the same node are labeled by equal block characters.
2. For each sub-picture P there exists at least one node on the path from the root and concatenated labels on this path form this sub-picture P.
3. For any two nodes u and v, v being the ancestor of u, and $P(u), P(v)$ being the sub-pictures made of concatenated labels from the root to u, v, respectively, $P(v) \sqsubset P(u)$.

A suffix tree for pictures is then made of this index so that maximal paths of one-child nodes are compressed into single arcs.

Moreover, they proved the lower bound on the number of nodes of the suffix tree for 2D pictures: $\Omega(n^2 n')$, where $(n \times n')$ is the size of a picture, and $n \leq n'$. These suffix trees can be built in $\mathcal{O}(n^2 n' \log n')$ time and stored in optimal $\mathcal{O}(n^2 n')$ space [GG97b, Theorem 10.22]. It means the construction of suffix tree for a square picture is easier than for an arbitrary sized picture. These results, used in [GG97b], originated in Giancarlo [Gia93].

The generalized index for structures of d dimensions is due to the same authors [GG97a], for two previous works in this area see the references in the paper. They present *raw* and *heterogeneous* index that differ in work optimality of algorithms searching in them. Additionally, their method allows *dimensional wildcards*, that is, the pattern array may have less dimensions than is the number of dimensions of the indexed text array. The Arbitrary CRCW PRAM

algorithm builds the index in $\mathcal{O}(\log N)$ time using $\frac{N^2}{n_{max}}$ processors, where N is the size of the text array, n_{max} is the longest side of the text array. The decision whether a pattern array is a sub-array of the text or not can be made in $\mathcal{O}(\log M)$ time using $\frac{M}{\log M}$ processors, where M denotes the size of the pattern array.

Two-dimensional Pattern Matching

4 Two-dimensional Pattern Matching Using Finite Automata

4.1 Overview of our approach

In this chapter the idea of dimensional (linear) reduction is used to provide a generic algorithm of the 2D pattern matching using finite automata. The dimensional reduction is known for a very long time and widely used (in the area of 2D matching cf. [Bir77, Bak78, Ami92, Ami04] or a survey in [CR02]). In our approach, we construct algorithms systematically using finite automata to do the dimensional reduction and find any two-dimensional occurrences for the two-dimensional exact and approximate pattern matching.

In principle, during the dimensional reduction we obtain a mapping between final states of a finite automaton and one-dimensional shapes (strings) of the d dimensional pattern, and a preprocessed text array that is a new $d-1$ dimensional shape over a secondary alphabet of final states' labels. Then the linear reduction is used again and after $d-1$ steps we finally obtain the one-dimensional problem, solvable by one-dimensional pattern matching automaton.

Based on the generic algorithm, a couple of automata based models and algorithms for the 2D exact and 2D approximate pattern matching using the 2D Hamming distance are presented. For that purpose some of the wide scale of classical finite automata solving the one-dimensional exact and approximate pattern matching problems [CH97, Mel95, MH97] are reused. Proposed methods generalize one-dimensional pattern matching approach based on finite automata. The results presented in this chapter were published in [ŽM04, ŽM06].

4.2 Generic algorithm

Reusing of finite automata from the one-dimensional pattern matching for the two-dimensional pattern matching has several advantages:

- the same formal method of automata construction for pattern matching algorithms in both cases,
- description of all problems using a unified view,
- finite automata are relatively well known,
- they process a text in time proportional to the text's length, (for a fixed alphabet in linear time),
- there are known simulation methods of finite automata [Sel80, WM92, Hol00], and
- for some types of problems there are known algorithms constructing appropriate deterministic finite automata directly, some of them even in linear time [CH97].

In Sec. 3.1.1.1 it has been mentioned that 1D pattern matching problems are sequential problems and that it is possible to solve them using finite automata. However, two or more dimensional problems are inherently not sequential, therefore appropriate transformation of non-sequential problems to sequential ones is required.

The idea of solving multidimensional pattern matching problems using finite automata is simple: multiple automata are to be used, passing their results among them and reducing dimension of the problem by one in each step. In the last step classical pattern matching can be used.

In two dimensions it means:

1. A pattern matching automaton processes text array TA along one dimension, e.g. in 2D set of columns (or rows), which produces preprocessed text array TA' of the same shape as TA.
2. A pattern matching automaton searches for a special "representing" string in rows (or columns, it depends on the dimension selected in the previous step) of TA', each occurrence of this string determines position of a 2D occurrence in TA.

The strategy of searching for PA in the text array TA is outlined in Alg. 4.1.

Algorithm 4.1: A generic algorithm of the two-dimensional pattern matching using pattern matching automata
Input: A certain pattern matching problem that specifies the edit distance (if any) and a value k, $1 \leq k < mm'$, be the maximum number of allowed errors or $k = 0$ for the exact matching. Let pattern array PA be $PA \in A^{(m \times m')}$, text array TA be $TA \in A^{(n \times n')}$. Let Π be the set of all (distinct) columns of PA, treated as individual strings. Without loss of generality let us suppose the preprocessing starts vertically, i.e. in columns of TA.
Output: Reports all occurrences or first occurrence only, etc. It depends on further specification of searching.
Method:

1: Construct dictionary matching automaton $M(\Pi)$ (of type $SFF?CO$, see Tab. 3.1), $M(\Pi)= =(Q, A, \delta, I, F)$, for pattern matching of multiple strings. Each final state of $M(\Pi)$ corresponds to some pattern in Π and eventually to some number of errors found in a particular pattern. Types of errors (if any) are given by a kind of distance considered for the 2D matching ("?" in $SFF?CO$). Columns of the pattern array are identified by final states of the automaton $M(\Pi)$. (Note there can be less than m final states owing to possible identities between some columns.)

2: Automaton $M(\Pi)$ is applied to each column of TA and new array TA' is generated. Its elements are determined by the run of $M(\Pi)$. In case of the exact matching it suffices to store whether some string from Π ends at a given element (Lemma 4.3 and formula (4.4) show that in this case only one of all string from Π can be found in each step). In case of the approximate matching a number of errors for all strings from Π found in particular step must be stored into TA'.

3: For further matching a one-dimensional representation of the pattern array PA has to be computed, it is *string* R over the set of final states representing the exact match: the i^{th} symbol of R is the final state identified with the i^{th} column of PA without considering errors.

4: Build string matching automaton M' ($SFO??O$) searching for R with at most k errors. Counting these errors means to limit the sum of errors found in step 2 in each string (column) of PA in a particular 2D occurrence.

5: Automaton M' locates representing string R inside the rows of TA', reporting eventual (1D) occurrences of R in TA' and therefore also 2D occurrences of PA in the original TA.

Note 4.1: In step 4 of Alg. 4.1 the $SFO??O$ classification with two degrees of freedom is used, although in the following only one of them is required. It is the first "?" on the fourth position

in the classification string. This is because we have some ideas for developing further algorithms based on this approach that may require to use both.

4.3 2D exact pattern matching

Let pattern array *PA* be viewed as a sequence of strings. Without loss of generality let these strings be its columns. To locate columns of the pattern array within columns of the text array requires searching for several strings, see the idea in Fig. 4.1.

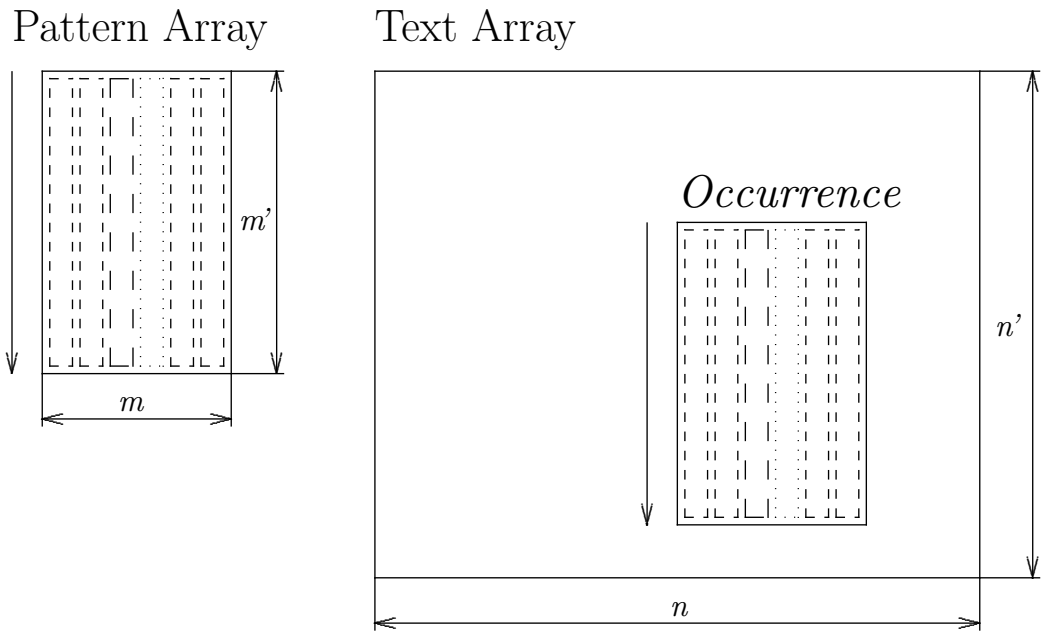


Figure 4.1. Pattern matching for columns of pattern array *PA* in text array *TA*

4.3.1 Finite automata model of the 2D exact pattern matching

Let m be the number of columns and m' be the number of rows of pattern array *PA*, m_d be the number of distinct columns of *PA*, $m_d \leq m$. In the following text all steps of Alg. 4.1 will be implemented describing the idea of the 2D exact pattern matching using finite automata.

According to step 1 of Alg. 4.1, the $M(\Pi)$ automaton of type *SFF?CO* has to be used. For purposes of the 2D exact pattern matching it is the *SFFECO* automaton, because a location of individual patterns' exact occurrences is required (Alg. 4.1, step 2).

SFFECO automaton $M(\Pi)$, $M(\Pi) = (Q, A, \delta, \{q_0\}, F)$, accepts language $L(M)$, $L(M) = A^*E(\Pi)$, where $E(\Pi) = \{P \in A^{m'};\ P \in \Pi\}$ (cf. Def. 2.35). Construction of automaton $M(\Pi)$ for searching for set Π, $\Pi = \{P_1, P_2, \ldots, P_{m_d}\}$, consists of a trie construction of m_d patterns and adding a selfloop at the initial state. $M(\Pi)$ construction is done by Alg. 4.2 and its result is in Fig. 4.2. Note that *SFFECO* here has its maximum size, i.e. patterns in Π have no common prefix.

Lemma 4.2
The number of states of a trie for set Π of m_d patterns, each of length m', constructed by Alg. 4.2, has at most $m_d m' + 1$ states. The maximum number of states of such a trie is $mm' + 1$ and $mm' + 1 \geq m_d m' + 1$.

Algorithm 4.2: Construction of FA for the exact matching of set of strings Π
Input: Set Π of (distinct) strings of pattern array PA (its columns or rows). Cardinality of set Π is $|\Pi| = m_d$.
Output: FA $M(\Pi) = (Q, A, \delta, \{q_0\}, F)$ accepting language $L(M)$, $L(M) = A^*E(\Pi) = \{wP_i;\ w \in A^*, P_i \in \Pi, 1 \leq i \leq m_d\}$.
Method:
$m_d = |\Pi|$
$m' = |P_1|$ $\{\, |P_1| = |P_2| = \cdots = |P_{m_d}| \,\}$
Let $Q = \{q_0\}$, q_0 is the initial state representing the empty prefixes of all strings from Π.
Every state q, $q \in Q$, of the *SFFECO* corresponds to prefix $p_{i,1}p_{i,2}\cdots p_{i,j}$ of some pattern $P_i \in \Pi$, $1 \leq i \leq m_d$, $1 \leq j \leq m'$.
Let us define $\delta(q, p_{i,j+1}) = q'$, where q' corresponds to prefix $p_{i,1}p_{i,2}\cdots p_{i,j}p_{i,j+1}$ of pattern P_i.
Every state which corresponds to any whole pattern $|P_i|$ is a final state and may have associated some output action reporting that pattern P_i has been found.
$\delta(q_0, a) = \delta(q_0, a) \cup \{q_0\},\ \forall a \in A$ $\{$ the selfloop at the initial state $\}$

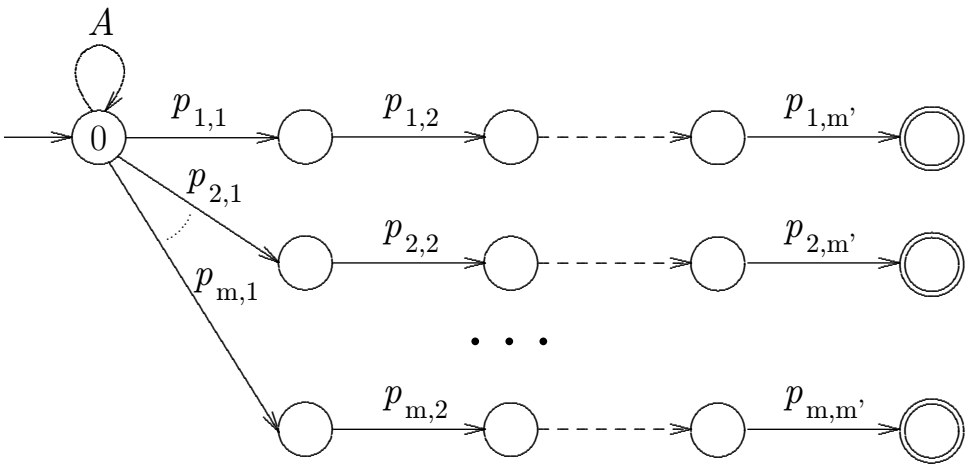


Figure 4.2. *SFFECO* automaton searching for set $\Pi = \{P_1, P_2, \ldots, P_m\}$

Proof
In Alg. 4.2 we have built a *SFFECO* automaton that is in fact a trie with a selfloop on its initial state. The highest number of states is reached in situation where there is no common prefix of any two strings from Π. Then *SFFECO* automaton is in fact composed of m_d *SFOECO* automata (for the full exact matching of one string) who share their initial state. Therefore the number of states of constructed trie is $m_d m' + 1$. Moreover, the number of distinct strings in the set Π cannot be greater than the number of columns of the pattern array PA, $m \geq m_d$, thus inequality $mm' + 1 \geq m_d m' + 1$ holds. □

Lemma 4.3
Let $M(\Pi)$, $M(\Pi) = (Q, A, \delta, \{q_0\}, F)$, be a finite automaton for the exact matching of set of patterns constructed by Alg. 4.2. FA $M(\Pi)$ has at most one active final state after reading each input symbol and executing all subsequent transitions.

Proof
Since Π is the set of strings, it cannot contain any pairwise equal strings. From presumptions and construction of FA $M(\Pi)$ we know that lengths of all strings in the set are equal, so in every step of its run at most one exact occurrence of a string from the set Π can be found. □

In step 2 the automaton $M(\Pi)$ preprocesses TA and produces TA' according to formula (4.3) below. Let $M(\Pi) = (Q, A, \delta, q_0, F)$ be DFA (how to obtain the appropriate DFA see

Sec. 4.3.2), $q \in Q$, q is the active state after reading a symbol from the element $TA[i,j]$, then

$$TA'[i,j] = \begin{cases} q & ; \ q \in F \\ 0 & ; \ q \notin F \end{cases} \quad \forall i,j \ \ 1 \leq i \leq n, \ 1 \leq j \leq n' . \tag{4.3}$$

Let the $M(\Pi)$, constructed by Alg. 4.2, has output actions associated with its final states only. When a final state becomes active, its associated output action puts its identification into array TA' on the active element. Lemma 4.3 guarantees that at most one such output action can be active in each step. Otherwise formula (4.3) guarantees that the active element of TA' is initialised by a label not in F. After processing of TA by $M(\Pi)$ all elements of TA' are properly set.

4.3.1.1 *The representing string*

In step 3 of Alg. 4.1 construction of *representing string* R of pattern array PA is required. String R represents the original 2D pattern array in such a way that every column becomes its representing symbol in R. This representing string is a result of the linear reduction, it is a one-dimensional representation of a 2D entity in symbols of a new alphabet, here those symbols are labels of final states of $M(\Pi)$.

Algorithm 4.3: Construction of string R representing pattern array PA
Input: Connection between strings from set Π and the final states of automaton $M(\Pi)$, $M(\Pi) = (Q, A, \delta, \{q_0\}, F)$.
Output: String R of length m: $R = r_1 r_2 \cdots r_m$, $R \in F^m$, $m \geq 1$.
Let $m_d = |\Pi|$. Then string R is created as follows:
Method:
for $i = 1$ **to** m **do**
$r_i = s_j$, $s_j \in F$ such that column $PA[i]$ is represented by a label of the final state s_j of $M(\Pi)$, $1 \leq j \leq m_d$.
end

Remark 4.4
The representing string and its construction Alg. 4.3 is universal for all kinds of the 2D pattern matching using finite automata, because it does not consider any error distances.

Example 4.5
Let $PA = \begin{array}{|c|c|c|}\hline g & c & g \\ \hline t & g & t \\ \hline t & a & t \\ \hline \end{array}$ and let columns $PA[1] = PA[3] = gtt$ and $PA[2] = cga$ be accepted by states f_1 and f_2, respectively, $f_1, f_2 \in F$ be final states of automaton $M(\Pi)$. Then representing string R will be 121.

4.3.1.2 *Searching for 2D exact occurrences using the exact pattern matching automaton*

After application of $M(\Pi)$ on every column of TA the text array TA' is prepared for application of some one-dimensional pattern matching method, e.g. finite automaton for the exact pattern matching.

According to step 4 of Alg. 4.1, automaton M' of type $SFO??O$ has to be used. For purposes of the 2D exact pattern matching it is the simplest one, the $SFOECO$ automaton.

$SFOECO$ automaton M' is constructed by Alg. 3.1 over the alphabet $F \cup \{0\}$, that is over the set of final state identifiers of automaton $M(\Pi)$, united with at least one symbol that represents the rest of states of $M(\Pi)$ (recall the idea of the reduced alphabet, Def. 2.10). In detail, $M' = (Q', F \cup \{0\}, \delta', \{q_0'\}, F')$ and accepts language $L(M) = (F \cup \{0\})^* E(R)$, where

$E(R) = \{R;\ R \in F^m\}$, $F = \{s_1, s_2, \ldots, s_{m_d}\}$, $|F| = m_d = |\Pi|$, m is the length of rows of PA. Automaton M' is depicted in Fig. 4.3.

FA M' searches for pattern R of length m in each row of TA' individually (Alg. 4.1, step 5). M' searches for occurrences of R and therefore it can report also 2D occurrences of PA, because at the moment R is found it has been verified that all columns of PA are found in the appropriate place and order.

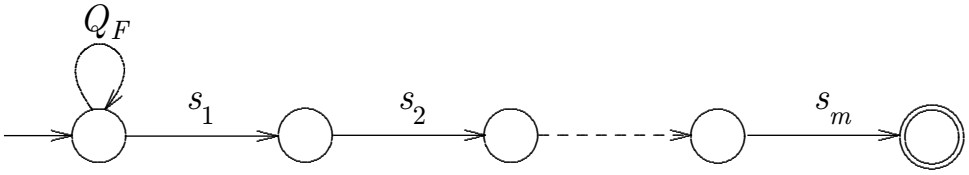


Figure 4.3. *SFOECO* automaton for the exact pattern matching of string R, $|R| = m$, over finite alphabet $Q_F = F \cup \{0\}$, F are final states of $M(\Pi)$ automaton

4.3.2 Deterministic finite automata for 2D exact pattern matching

The Bird and Baker algorithm (Sec. 3.4) uses for 2D exact pattern matching well known algorithms of Aho-Corasick and Knuth-Morris-Pratt, and their method works in linear time. This fact is a strong motivation for us to achieve the linear time with our method too.

Our model of the 2D exact pattern matching requires two finite automata: $M(\Pi)$ automaton (Alg. 4.1, step 1) is the *SFFECO* automaton (Fig. 4.2); M' automaton (step 4) is the *SFOECO* automaton (Fig. 4.3).

These automata have neat structure, so they are useful as a model, but they are also nondeterministic. There is no problem with this fact, as these FA's can be either simulated or determinised before use. However, in order to achieve the linear time complexity for the 2D exact pattern matching using our method, direct constructions of equivalent deterministic finite automata have to be used (Def. 2.42, Def. 2.51). Fortunately, it is possible in this case (cf. [CH97]): DFA $M(\Pi)$ can be constructed as a *linear dictionary-matching automaton* with time complexity $\mathcal{O}(\log|A||PA|) = \mathcal{O}(mm')$, supposing alphabet A is fixed. Its space complexity is $\mathcal{O}(\log|A||PA|) = \mathcal{O}(mm')$, DFA M' can be constructed as a *linear string-matching automaton* with time and space complexity $\mathcal{O}(|R|) = \mathcal{O}(m)$.

Lemma 4.2 holds in this case without changes, because it describes Π trie's maximum number of states. Lemma 4.3 holds for deterministic version too. $M(\Pi)$ is a deterministic finite automaton now, hence we can simply state why there cannot be found more than one exact occurrence of a string from Π in each step, because

$$\forall u, v \in \Pi, u \neq v,\ u, v \in A^*, |u| = |v| = m'\,. \tag{4.4}$$

Let A be a finite alphabet, TA be the text array of size $(n \times n') = N_{TA}$, $TA \in A^{(n \times n')}$, PA be the pattern array of size $(m \times m') = N_{PA}$, $PA \in A^{(m \times m')}$ and the pattern array is not larger than the text array: $m \leq n\ \wedge\ m' \leq n'$. Let m, m' be the number of PA columns and rows, respectively, and Π be the set of columns of PA. Presented approach leads to the following result:

Theorem 4.6
The 2D exact pattern matching using the direct construction of DFA's has asymptotic time complexity $\mathcal{O}(N_{TA})$. This time complexity is the same as in the Bird and Baker solution, it means it is linear with the size of a given text array.

Proof
Referring to Alg. 4.1 again:

1. the construction of dictionary matching automaton $M(\Pi)$ takes $\mathcal{O}(N_{PA})$ time,
2. pattern matching in TA is done by $M(\Pi)$ in linear time, $\mathcal{O}(N_{TA})$,
3. construction of representing string R requires $\mathcal{O}(m)$ time, it depends only on the length of the appropriate side of PA (e.g. on the length of rows), see Alg. 4.3,
4. construction of linear string matching automaton M' takes $\mathcal{O}(|R|) = \mathcal{O}(m)$ time,
5. pattern matching in TA' is done by M' also in $\mathcal{O}(N_{TA})$ time.

The asymptotic time complexity of the two-dimensional exact pattern matching is then a sum of the complexities above, that is $\mathcal{O}(2N_{TA} + N_{PA} + 2m) = \mathcal{O}(N_{TA})$. □

The space complexity depends on sizes of pattern matching automata and a space for a temporary data:

Proposition 4.7
The basic version of the 2D exact pattern matching using finite automata presented above has asymptotic space complexity $\mathcal{O}(N_{TA})$.

Proof
1. Automaton $M(\Pi)$ ($LDMA$) needs $\mathcal{O}(N_{PA})$ space (A is the finite alphabet),
2. preprocessed array TA' requires $\mathcal{O}(N_{TA})$ space,
3. representing string R needs $\mathcal{O}(m)$ space,
4. automaton M' ($LSMA$) needs $\mathcal{O}(|R|) = \mathcal{O}(m)$ space, and
5. in the last step only constant extra space is needed.

The two-dimensional exact pattern matching can be done with $\mathcal{O}(N_{TA})$ asymptotic time complexity. □

Note 4.8: In this version of the 2D exact matching the space occupied by $M(\Pi)$ can be reused right after R is computed.

4.3.3 Space-efficient 2D exact pattern matching

This section shows how to use some natural properties of finite automata to reduce the space complexity of the 2D exact pattern matching.

From Proposition 4.7 we see immediately that the largest structure is preprocessed text array TA'. New text array TA' has the same size as TA, and because we are dealing with pictures it is clear that its size can be very large. So its usage is not optimal and we have to avoid it.

It is possible to utilize the fact that all data in a computer are stored on some permanent storage device and they are loaded into memory before any processing. Our automaton $M(\Pi)$ can work over the original array TA, replacing every symbol read, because after any active element is read it will never be read again. This eliminates the need for any extra space, except for the automaton itself. On the other hand, the original text array is destroyed and if needed, it has to be loaded into memory again.

Although replacing of text array is possible, it is not very universal. However, we still did not use all powerful properties that finite automata, despite their simplicity, have. To be able to restart a run of a deterministic finite automaton only its active state and the current position in the input text are required. Automaton $M(\Pi)$ matches in all columns of TA individually, so $\mathcal{O}(n)$ extra space is needed to store all active states of $M(\Pi)$ in each row of TA. Once one row of TA' is computed, it is possible to do matching in it using automaton M' to find possible 2D occurrences. Then the row's space can be reused, eliminating efficiently the need to store whole array TA'.

Looking closely at this refinement, we see a reduced alphabet can be used in construction of automaton M' to avoid unnecessary transitions. The alphabet can be reduced to the final states'

labels and one representative of the rest of the states. This is because new TA' is constructed over all state labels Q of automaton $M(\Pi)$, $TA' \in Q^{(n \times n')}$, to be able to restart its run.

A new formula defining content of elements of TA' follows: let $M(\Pi) = (Q, A, \delta, q_0, F)$ be DFA, $q \in Q$, q is the active state after reading a symbol from the element $TA[i, j]$, then

$$TA'[i, j] = q\,;\ q \in Q,\ \forall i, j\ \ 1 \leq i \leq n,\ 1 \leq j \leq n'\,. \tag{4.5}$$

At this moment, we have DFA $M(\Pi)$ with output actions associated with all its states, preprocessing text array TA over A to TA' over Q according to formula (4.5), and DFA M' accepting pattern R over $F \cup \{x\}$, where x represents all non-final states' labels $Q \setminus F$.

Theorem 4.9
The 2D exact pattern matching using pattern matching automata has asymptotic space complexity $\mathcal{O}(N_{PA} + n)$.

Proof
The space complexity of automata $M(\Pi)$ and M', $\mathcal{O}(N_{PA})$ and $\mathcal{O}(m)$, respectively, remains unchanged (proof of Proposition 4.7, step 1 and 4, respectively).

Since $M(\Pi)$ treats each column and M' each row individually, to be able to restart the preprocessing phase it is required to store the active states of $M(\Pi)$ only. The extra space required is $\mathcal{O}(n)$, the number of columns of TA.

Steps 2 and 5 of Alg. 4.1 are now "interleaved", therefore both automata are stored in memory at the same time. The asymptotic space complexity of the 2D exact pattern matching is then $\mathcal{O}(N_{PA}) + \mathcal{O}(m) + \mathcal{O}(n) = \mathcal{O}(N_{PA} + n)$. □

Note 4.10: To save some processing time, it is sufficient to start matching for 2D occurrences (Alg. 4.1, step 5) in row $TA'[x, m']$, $1 \leq x \leq n$, where the topmost occurrences may be located.

4.4 2D approximate pattern matching

In the previous text, we have seen one method of the two-dimensional exact pattern matching. Its preprocessing phase is based on the processing of one-dimensional information from columns of a text array. It uses a dictionary matching automaton to convert prefixes of all strings from the set of columns of the pattern into symbols of a new alphabet, the alphabet of state labels of the matching automaton. However, prefixes, represented by non-final states, are not interesting for us, only whole string (column) match indicators are important. Hence, for the final matching phase a reduced alphabet is used. It is reduced to labels of automaton's final states plus one representing all other states.

4.4.1 Finite automata model of the 2D approximate pattern matching using the 2D Hamming distance

In this section a new, automata based, method of the 2D approximate pattern matching using the 2D Hamming distance (Def. 2.113) is introduced. To keep an analogy with the 1D case we name it also 2D matching with mismatches. In the 2D exact matching a dimensional reduction was sufficient to do the task. In case of the 2D approximate pattern matching more information about each prefix is needed: we need to know some prefix ends at the actual element, moreover, we need to know that there is (no more than) a certain number of (1D) mismatches in the prefix.

Let us refer again to the generic algorithm (Alg. 4.1). In step 1 approximate dictionary matching automaton $M(\Pi)$ for the approximate matching of set of strings has to be built. In

this case it is *SFFRCO* automaton for the approximate matching of set of strings using the Hamming distance (Def. 2.23). $M(\Pi)$ accepts a language $L(M) = A^* H_k(\Pi)$, where

$$H_k(\Pi) = \{X;\ X \in A^*, D_H(X,P) \leq \min(k, m'-1)\ \wedge\ P \in \Pi\}, \tag{4.6}$$

k is a given number of allowed 2D mismatches in the occurrence of pattern array PA and m' is the length of columns of PA (suppose we start the processing vertically). The construction of $M(\Pi)$ is outlined in Alg. 4.4.

Since $M(\Pi)$ searches for a set of strings, it consists of *SFORCO* sub-automata for the approximate pattern matching using the Hamming distance. (*SFORCO* automata and their usage are discussed for example in [Mel95].)

The reason why the distance $\min(k, m'-1)$ in formula (4.6) has to be used is that *SFORCO* sub-automaton for pattern of length m' can find its occurrences with at most $m'-1$ mismatches in every column of *TA*. *SFORCO* sub-automaton matching for a pattern of length m' with at most $k' = m' - 1$ mismatches is in Fig. 4.4.

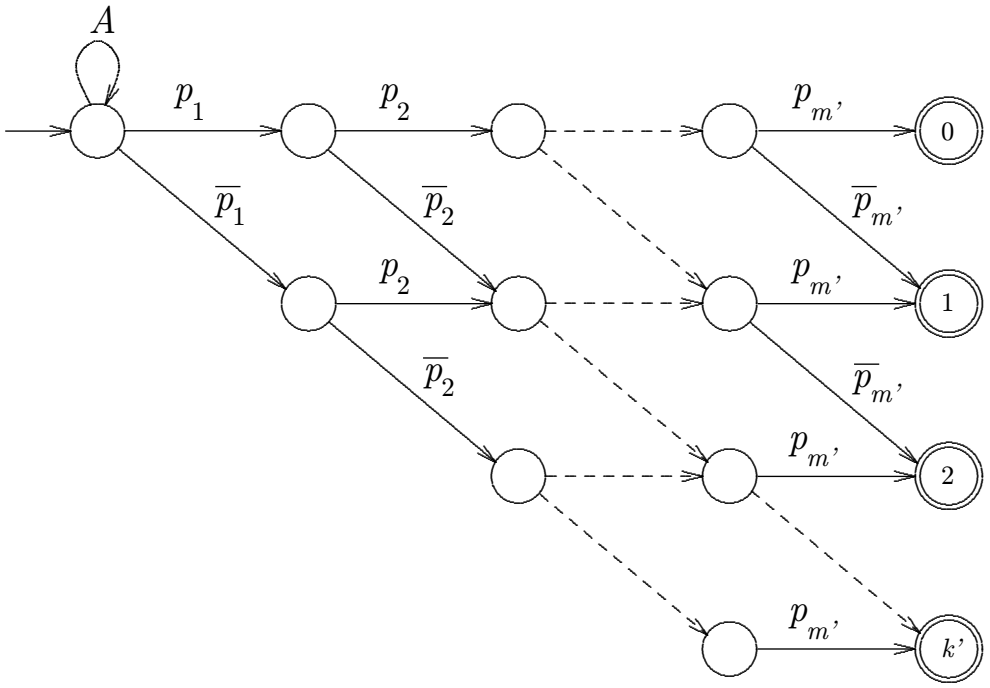


Figure 4.4. FA for the approximate pattern matching of some pattern P, $P \in \Pi$, using the Hamming distance, $m' = |P|$, the number of errors allowed is $k' = m' - 1$

The final states of $M(\Pi)$ indicate which one of (1D) patterns was found and the amount of mismatches found in a particular occurrence within columns of *TA*. Note that only one exact match can be found in each step, see formula (4.4), however multiple patterns can be found at the same moment with at least one error. In Alg. 4.4 the final states of $M(\Pi)$ are labeled by integers that are used in the definition of mapping δ. In addition to these numbers, a special label is assigned to each final state. These labels are required by further constructions, as well as they make referring to particular states easier. For our purposes, they are labeled by an ordered couple (s, x) (e.g. Fig. 4.5), where s is a string identification (number of its matching *SFORCO*), to which this final state belongs, and x is the number of mismatches found in it. An example (the *SFFRCO* automaton) is shown in Fig. 4.5, $\bar{x}$ represents a complement of symbol $x \in A$ (Def. 2.2).

For the following two statements (4.11 and 4.12) let $M(\Pi)$, $M(\Pi) = (Q, A, \delta, I, F)$, be *SFFRCO* automaton for the approximate matching of a set of patterns using the Hamming

Algorithm 4.4: Construction of FA for the approximate matching of set of strings Π using the Hamming distance
Input: Set Π of (distinct) strings of pattern array PA (its columns or rows), the cardinality of set Π is $m_d = |\Pi|$, and the maximum number of 2D errors allowed k, $k \in \mathbf{N}$, $k < mm'$.
Output: FA $M(\Pi) = (Q, A, \delta, I, F)$ accepting language $L(M)$, $L(M) = A^* H_k(\Pi) = \{uv;\ u, v \in A^*, D_H(v, P) \leq \min(k, |P| - 1)\ \wedge\ P \in \Pi\}$.
Method:
$I = \emptyset$; $F = \emptyset$
$k' = \min(k, |P| - 1)$, where $P \in \Pi$
$|Q_{SFORCO}| = (k' + 1)(|P| + 1 - \frac{k'}{2})$, where $P \in \Pi$ { Lemma 3.2 }
for $i = 1$ **to** m_d **do** { for each string $P_i \in \Pi$ }
  $first = (i - 1)|Q_{SFORCO}|$
  Use Alg. 3.2 to construct *SFORCO* automaton M_i, $M_i = (Q_i, A, \delta_i, \{q_{first}\}, F_i)$, for $P_i \in \Pi$ with at most $\min(k, |P_i| - 1)$ mismatches, such that its states' numbers begin with the value *first*. { This makes all states unique in $M(\Pi)$. }
  $I = I \cup \{q_{first}\}$
  foreach $q_j \in F_i$ **do**
    Assign an alias of state q_j to $q_{i,x}$, where q_j represents match of P_i with exactly x mismatches.
    $F = F \cup \{q_j\}$
  end
  $Q = Q \cup Q_i$
  foreach $q_i \in Q_i$ **do**
    $\delta(q_i, a) = \delta_i(q_i, a)$, $\forall a \in A$
  end
end

distance, constructed by Alg. 4.4, the number of strings in set Π be $m_d = |\Pi|$, k be the maximum number of 2D mismatches (errors) allowed and $k' = \min(k, |P| - 1)$, where $P \in \Pi$.

Lemma 4.11
A nonminimized (nondeterministic) finite automaton $M(\Pi)$ has
1. $m_d(k' + 1)(m' + 1 - \frac{k'}{2})$ states, and
2. this includes $m_d(k' + 1)$ final states.

Proof
1. The number of states of each sub-automaton is $(k' + 1)(m' + 1 - \frac{k'}{2})$, see Lemma 3.2. The number of sub-automata in automaton $M(\Pi)$ is equal to the number of distinct columns in pattern array PA, i.e. m_d.
2. Moreover, every sub-automaton has $k' + 1$ final states to distinguish k' errors, and thus both claims hold. □

Proposition 4.12
SFFRCO automaton $M(\Pi)$ may have after each reading of input symbol and executing all subsequent transitions
1. at most m_d final states active,
2. among these active final states at most one final state indicating the exact occurrence of pattern P_i of Π in the text array,
3. at most m_d active final states, indicating occurrences of m_d different patterns with l mismatches, $0 < l \leq k'$.

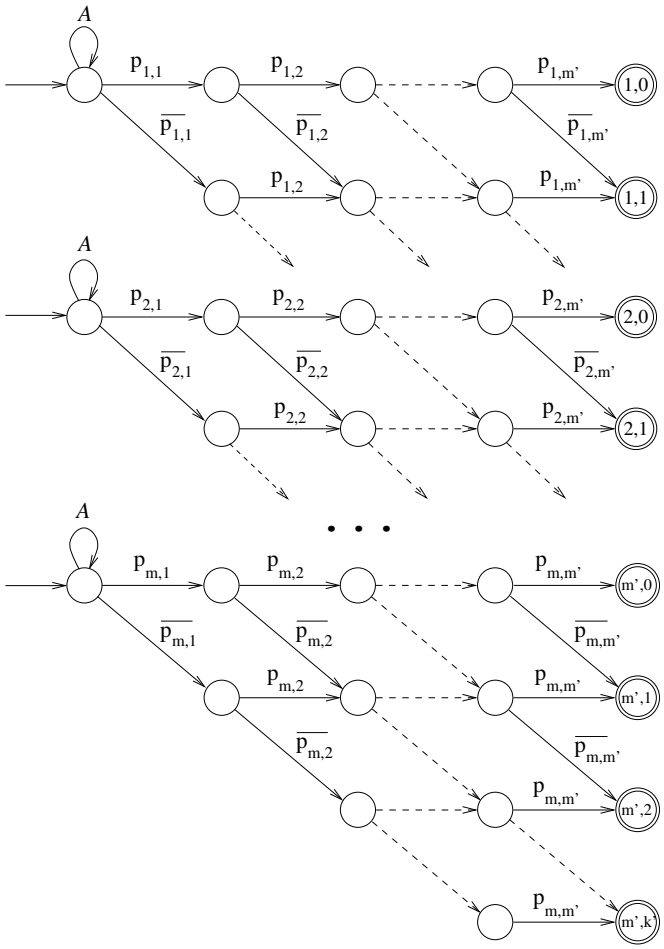


Figure 4.5. *SFFRCO* automaton $M(\Pi)$ with multiple initial states searching for set of strings $\Pi = \{P_1, P_2, \ldots, P_m\}$ with at most $k' = m' - 1$ mismatches allowed

Proof

1. Our first claim holds if and only if every sub-automaton constructed by Alg. 3.2 can have at most one final state active in each step. This property has been proved in [Hol00] using the induction over the length of the oriented path from the initial state. Since this property holds, there can be at most m_d such states in whole automaton $M(\Pi)$ (one in every sub-automaton) and after reading r symbols, $r \geq m'$, there can be at most m_d active final states.
2. The second property and its proof is similar to Lemma 4.3. As Π is the set of strings of equal length, it cannot contain any pairwise equal strings. As a consequence, in every step of run of $M(\Pi)$ at most one exact occurrence of a string from the set can be found. Moreover, if there is an exact occurrence of string P_i, $1 \leq i \leq m_d$, there can be at most $m_d - 1$ approximate occurrences of strings P_j, $j \neq i$, in particular step, because other $m_d - 1$ strings differ from P_i at least in one symbol.
3. The third statement is a direct consequence of the previous two: if there is no active final state indicating the exact occurrence then there could be up to m_d active final states indicating occurrences with l mismatches, $0 < l \leq k$. □

Remark 4.13

In [Hol00] there were introduced two versions of the *SFORCO* automaton, the so-called $\bar{p}$-version (the one we use here) and A-version that has transitions representing the *replace* operation defined for all symbols of alphabet A. In the latter case multiple final states may

be active in each step. The A-version of an approximate pattern matching automaton may be useful in applications since its simulation is more straightforward. Let us point out that the statement of Proposition 4.12 holds for this version too, with the following modification in run of $M(\Pi)$.

Let some sub-automaton number i, $1 \leq i \leq |\Pi|$, with final states F_i, has multiple active final states in a particular step. Let X be a set of active final states, $X \subseteq F_i$, and let operation "min" selects the minimum number of mismatches the states in set X indicate. Then $\exists q$ such that $q = \min(X)$. Now we can "deactivate" those active final states $r \in X$, where $r \neq q$, because they indicate more than the minimum number of errors in this step and thus they are not interesting. In other words, sequences of transitions ending in the configurations (r, ε), $r \in X$, are not considered.

Now let us continue to develop the algorithm: in step 2 of Alg. 4.1 array TA' is generated. *SFFRCO* $M(\Pi)$ must save enough information for the next matching phase, so array TA' will need asymptotically more memory than TA.

As a consequence of Proposition 4.12, the space required to store every element of TA' may be enlarged by a coefficient equal to the number of patterns in set Π. (In case of the 2D exact pattern matching space requirements of TA and TA' were asymptotically the same.)

Let $M(\Pi) = (Q, A, \delta, I, F)$, $q \in Q$ be the active state after reading symbol from element $TA[i, j]$, an ordered couple (s, x) be the label of a final state of $M(\Pi)$, $\oslash \notin Q$ be a special symbol not among the labels of states.

Then it holds for $\forall i, j, s$ $1 \leq i \leq n, 1 \leq j \leq n', 1 \leq s \leq |\Pi|$:

$$TA'[i, j, s] = \begin{cases} x & ; \ q \text{ active}, q \in F, q = (s, x), \\ \oslash & ; \ s^{th} \text{ sub-automaton has no active final state.} \end{cases} \tag{4.7}$$

Let l_s be the number of mismatches for each pattern number s and $k' = \min(k, |P_s| - 1)$, $P_s \in \Pi$. Every element of TA' provides following information about potential occurrence of a column of PA ending at a particular element in the original array TA:

1. the exact match, $l_s = 0$,
2. the approximate match with l_s mismatches, $0 < l_s \leq k'$,
3. no match, $TA'[i, j, s] = \oslash$ ($l_s > k'$ mismatches found).

4.4.1.1 *Construction of the representing string from $M(\Pi)$ for the approximate matching*

In step 3 of Alg. 4.1 string R, representing original pattern array PA in preprocessed text array TA', needs to be constructed. Originally string R is constructed by Alg. 4.3 over the set of final states of dictionary matching automaton $M(\Pi)$, but here $M(\Pi)$ has multiple final states for each string $P \in \Pi$. The above mentioned Alg. 4.3 can be used for construction of R in this case, however with a slight adjustment. It takes the final states' labels of $M(\Pi)$ of the form (s, x) and

- from each sub-automaton M_i of $M(\Pi)$ only one (arbitrary) final state is taken,
- only the number of a particular string is considered ("s" part of (s, x) couple).

4.4.1.2 *Searching for 2D approximate occurrences using the 2D Hamming distance*

In step 4 of Alg. 4.1 *SFO??O* automaton M' is required for the error counting approximate pattern matching of representing string R. In this step, the *SFOLCO* automaton has to be used, that is a finite automaton able to match pattern R using the Γ distance (Def. 2.27).

Let F in this section denote the set of final states of finite automaton $M(\Pi)$ and secondary alphabet A' be set F extended with a special symbol "$\oslash$": $A' = F \cup \{\oslash\}$. Symbols of alphabet A' denote the number of errors found at given element in the particular pattern from Π. Symbol $\oslash$ represents the situation the number of errors in particular string is at given element greater than the number of mismatches allowed in it.

Let M' be $M' = (Q', A', \delta', \{q'_{0,0}\}, F')$, δ' is a mapping $Q' \times (s, A') \mapsto \mathcal{P}(Q')$, where Q' are states of M', s is the number of string from Π. Functioning of M' over text array TA' is slightly different from a run of finite automaton over linear text. M' is able to select (nondeterministically) the appropriate symbol from the set of symbols stored in each element of TA'. On the other hand it does not extend the idea of nondeterminism in the sense of existence of the sequence of transitions leading to accepting configuration (Def. 2.39, 2.40).

The active final state of M' indicates that an approximate 2D occurrence of PA in TA has been found. Furthermore, it shows the total number of 2D mismatches found in a particular 2D occurrence, up to the given maximum k.

Its construction algorithm is presented in Alg. 4.5. It is relatively straightforward but special attention has to be taken to "borderline cases". Another interesting case is when $k \geq m'$ is given. It means that the number of 2D errors is greater than the number of mismatches $M(\Pi)$ is able to find in one dimension. To be able to count 2D errors by one-dimensional means, $\oslash$ transitions are introduced in automaton M', representing $m' = |R|$ errors, $R \in \Pi$. Hence each state may have at most $m' + 1$ outgoing transitions, only state $q'_{0,0}$ has more, because of the selfloop.

In order to compute the Γ distance, we have to specify the ordering of alphabet A'. As one would expect, symbols of A' are ordered using second part x of (s, x) couple, i.e. the number of mismatches found in string $P_s \in \Pi$.

Let the distances between two symbols, i.e. between the labels of final states of $M(\Pi)$, be defined as follows:

$$|(s,x) - (t,y)| = \begin{cases} |x - y| & ; \ s = t, \\ m' & ; \ ((s,x) \vee (t,y)) = \oslash, \\ m' & ; \ s \neq t\,. \end{cases} \tag{4.8}$$

Example 4.14
To illustrate formula (4.8) let us compare symbols: $|(1,0) - (1,3)| = 3$, $|(2,3) - (5,0)| = m'$, $|(1,0) - \oslash| = m'$, $|\oslash - \oslash\,| = m'$.

These comparisons are used in the error counting in TA'.

Let err be the current value of mismatches found in a particular occurrence of the representing string. If symbol (i, j) is found in position where $(i, 0)$ is expected, then

$$l = \begin{cases} j & ; \ j \neq \oslash, \\ m' & ; \ j = \oslash. \end{cases} \tag{4.9}$$

It means the number of mismatches l is added to the current value of err, if $err + l \leq k$. If $j = \oslash$, $k \geq m'$ errors allowed and $err + m' \leq k$, the number of mismatches m' is added to the current value, because column of PA of length m' was not found on its expected position.

Note 4.15: The idea how M' works resembles matching with the Γ distance, so M' is classified as the *SFOLCO* automaton. One can object to this classification pointing out that it is also the *SFOHCO* problem, where Γ and Δ distances are combined (Sec. 3.1.1.1). To this objection, we note that the problem solved here is to sum the errors found (i.e. Γ), not to limit the distances between symbols (Δ). In fact every problem where the Γ distance is used is also Γ, Δ problem, where the values of both criteria are the same. Having formula (4.8), we have the alphabet properly ordered and need not to use the Δ distance explicitly.

Lemma 4.16
Finite automaton for the approximate pattern matching of a string representing pattern array from Alg. 4.5 has exactly $1 + \sum_{i=0}^{k} \left(m - \left\lfloor \frac{|i-1|}{m'} \right\rfloor\right)$ states.

Algorithm 4.5: Construction of FA for the approximate pattern matching of string R over the set of identifiers of final states of FA $M(\Pi)$ from Alg. 4.4
Input: Pattern R over the set of final states of $M(\Pi)$, $|R| = m$, and m' be the length of patterns in set Π. The maximum number of 2D errors allowed k, $k \leq mm'$.
Output: FA M', $M' = (Q', A', \delta', \{q'_{0,0}\}, F')$, accepting language $L(M') = A'^{*}\Gamma(R) = \{wx;\ w, x \in A'^{*}, D_{\Gamma}(x, R) \leq k\}$.
Sets Q', F' and mapping δ' are constructed in the following way:
Method:
$Q' = \{q'_{0,0}\}$ { the initial state }
for $i = 0$ **to** k **do**
  $first = \left\lfloor \frac{|i-1|}{m'} \right\rfloor + 1$ { a "depth" of the first non-initial state in the i^{th} row }
  for $j = first$ **to** m **do**
    $Q' = Q' \cup \{q'_{j,i}\}$ { add a new state }
    if $j = m$ **then**
      $F' = F' \cup \{q'_{j,i}\}$ { add a new final state }
      Assign an alias $q'_{j,i} = i$. { the number of errors found }
    end
    if $j = 1$ **then**
      if $i < m'$ **then** $\delta'\left(q'_{j-1,0}, (R[j], i)\right) = q'_{j,i}$
        else $\delta'\left(q'_{j-1,0}, (R[j], \oslash)\right) = q'_{j,i}$ { if $i = m'$ }
    else
      $x = \max(0, i - m')$
      if $j = first$ **then** $last = i \pmod{m'}$
        else $last = \min(i, m')$
      for $l = 0$ **to** $last$ **do**
        if $(i \geq m') \wedge (l = 0)$ **then** $\delta'\left(q'_{j-1,x}, (R[j], \oslash)\right) = q'_{j,i}$
          else $\delta'\left(q'_{j-1,x+l}, (R[j], i - x - l)\right) = q'_{j,i}$
      end
    end
  end
end
$\delta'(q'_{0,0}, a) = \delta'(q'_{0,0}, a) \cup \{q'_{0,0}\},\ \forall a \in A'$ { the selfloop of the initial state }

Proof
Considering Alg. 4.5, we see there is one initial state and then at most m states for each number of errors i. More precisely, the length of "rows" of all states indicating the same value of 2D errors is decreasing in steps $\left\lfloor \frac{|i-1|}{m'} \right\rfloor$. □

Note 4.17: Looking closely on Alg. 4.5 one can notice that except the selfloop of initial state $q'_{0,0}$ the rest of mapping δ' is deterministic.

Let k be the number of mismatches allowed, $k < |P_i|$, $P_i \in \Pi$. In such case the number of 2D mismatches in any occurrence is lower than the length of columns of PA. Then *SFOLCO* automaton has simpler form than in the general case, as it allows at most $|P_i| - 1 = k'$ mismatches while searching for the representing string (Fig. 4.6).

The largest possible form of finite automaton M' is depicted in Fig. 4.7. Note that only its small part could have been depicted, just to show the reader one possible way how M' can reasonably be figured. The maximum number of mismatches k can be as high as $k = mm' - 1$. It means that to find a 2D occurrence only one symbol has to be found at its correct place. This is in perfect accordance with our experiences from the 1D pattern matching.

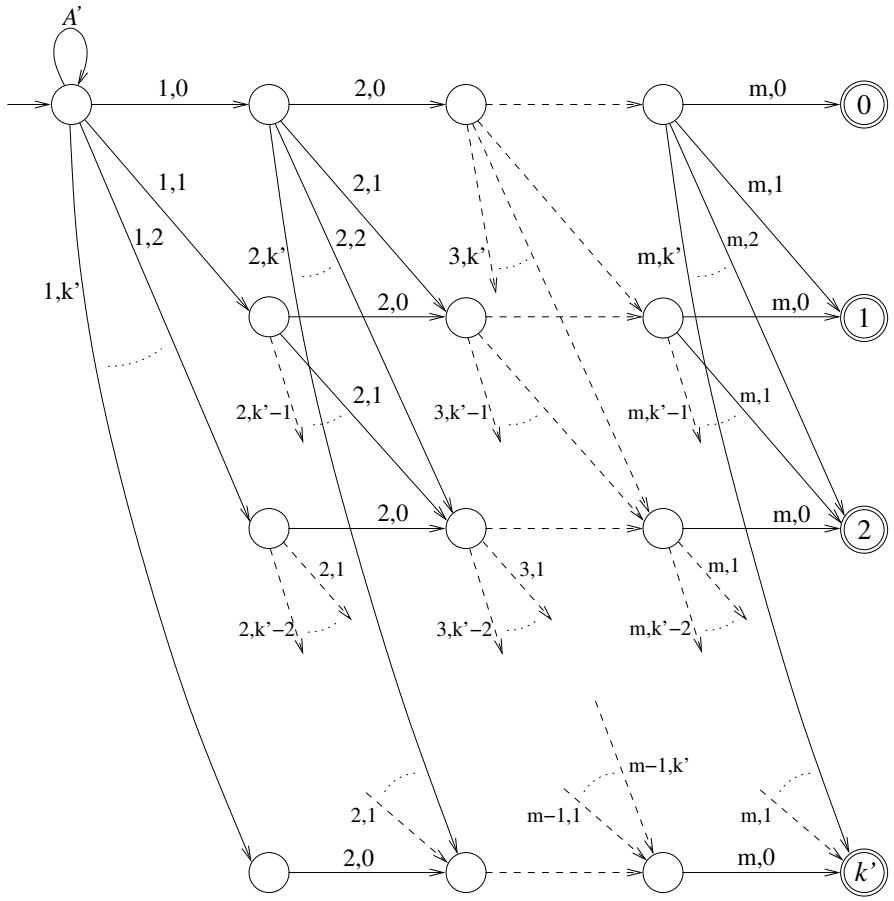


Figure 4.6. Finite automaton for counting errors in occurrences of the representing string $R = 1 \cdot 2 \cdots m$, $|R| = m$, the number of mismatches allowed is $|P_i| - 1 = k'$, $P_i \in \Pi$, Q_F the denotes set of final states' labels of $M(\Pi)$ automaton

4.4.2 Practical 2D pattern matching using the 2D Hamming distance

Our model of this type of the 2D approximate pattern matching requires two finite automata: $M(\Pi)$ automaton (Alg. 4.1, step 1) is the *SFFRCO* automaton (Fig. 4.5) and M' automaton (step 4) is the automaton *SFOLCO* automaton for counting errors while matching for representing string R (Fig. 4.7). These are nondeterministic and there are no known direct construction methods of their deterministic versions (in contrast to the 2D exact matching, Sec. 4.3.2).

In such situation we can either transform FA to DFA using the standard subset construction [HMU01] or simulate a run of the FA. The former method is not suitable for our purposes, because the time complexity of determinisation (transformation of FA to DFA) depends on the number of states of the resulting DFA and it may be quite high, up to $2^{|Q_{\mathrm{FA}}|}$. In practice this is not usual situation, nevertheless common substrings in patterns of Π will likely cause this number of states to be high. In the rest of the chapter we use the latter method, a simulation of run of $M(\Pi)$ and M'. (The problem of simulation of pattern matching automata of various kinds was in detail described by Holub in [Hol00].)

For the reader's convenience, let us remind again our basic presumptions: let A be a finite alphabet, TA be the text array of size $(n \times n') = N_{TA}$, $TA \in A^{(n \times n')}$, PA be the pattern array of size $(m \times m') = N_{PA}$, $PA \in A^{(m \times m')}$. Let m, m' be the numbers of PA columns and rows, respectively, and Π be the set of PA columns.

For the simulation of run of $M(\Pi)$ we use dynamic programming, a general technique used in various branches of computer science, in this case dynamic programming for pattern

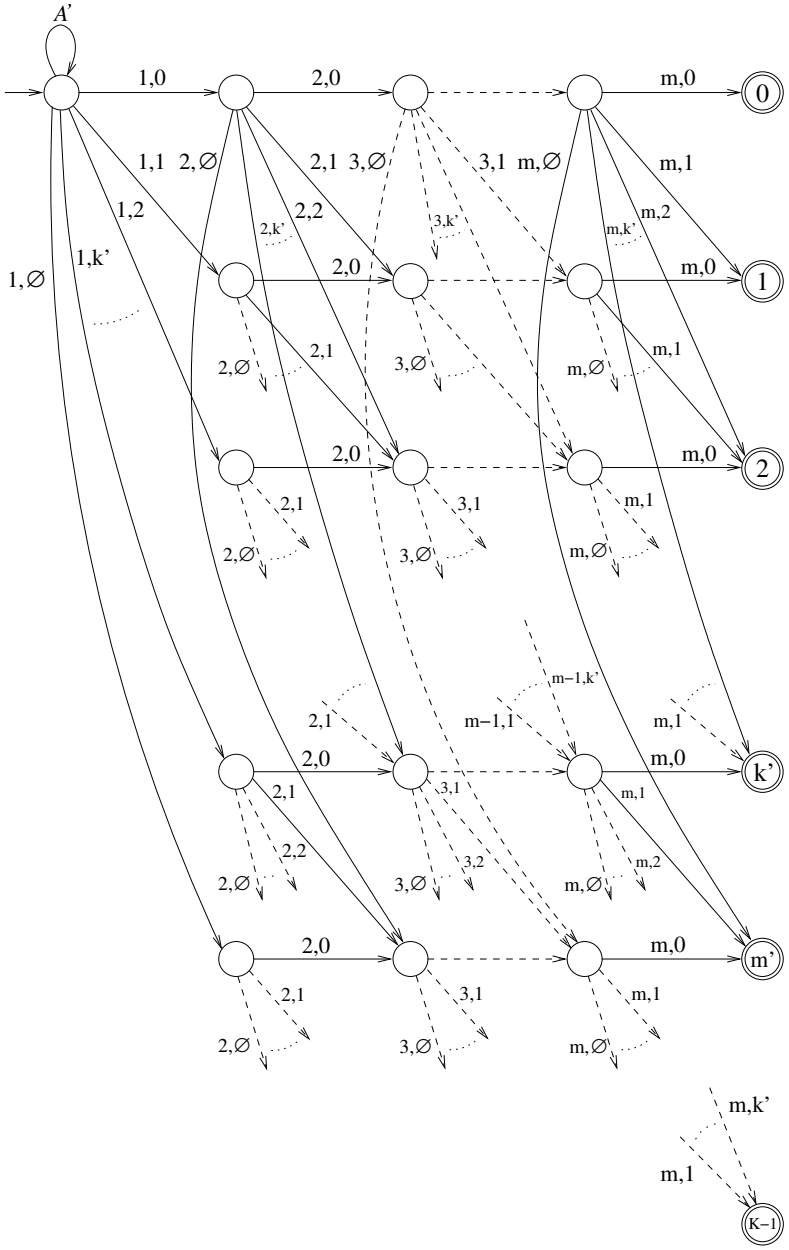


Figure 4.7. Finite automaton for counting errors in occurrences of the representing string $R = 1{\cdot}2 \cdots m$, $|R| = m$, where the maximum number of mismatches is allowed, i.e. $k = mm' - 1$, $K = mm'$, Q_F the denotes set of final states' labels of $M(\Pi)$ automaton

matching using the Hamming distance [Sel80, Ukk85b, Mel95]. This method for string matching computes matrix D of size $(m + 1) \times (n + 1)$ for a pattern of length m and a text of length n. Each element of D usually contains the edit distance between the string ending at current position in text T and the prefix of pattern P. The advantages of this method are that it is very simple and can be memory-efficient, for a pattern of length m it may require only $\mathcal{O}(m)$ space. It works in time $\mathcal{O}(mn)$ for the text of length n and this complexity is independent on the number of errors. Ukkonen [Ukk85b] improved the expected time of the standard dynamic programming for the approximate string matching to $\mathcal{O}(kn)$, by computing only the zone of the dynamic programming matrix consisting of the prefix of each column ending with the last k in the column, where k is the maximum number of mismatches in the occurrence. This improvement does not help much in our case, because for the 2D approximate matching we

usually need more than m errors to be allowed and then we have $k = m - 1$.

From the construction of $M(\Pi)$ we see there is at most m_d sub-automata for the approximate pattern matching of $m_d \leq m$ strings of PA. These can be simulated by the above mentioned dynamic programming in time $\mathcal{O}(mm'nn')$ that is independent on the number of 2D errors.

Automaton M' has a special structure (almost deterministic, see Alg. 4.5) and can be easily simulated using Alg. 4.6 in time $\mathcal{O}(mnn' - mm'n)$.

Algorithm 4.6: Simulation of FA M' (Alg. 4.1, step 5)
Input: Let $R[1..m]$ be the representing string of pattern array PA, $TA'[1..n, 1..n', 1..m']$ be the preprocessed text array, $k \geq 0$ be the allowed number of 2D mismatches, $1 \leq m \leq n$, $1 \leq m' \leq n'$.
Output: R is either accepted in the particular step with $\leq k$ mismatches and a 2D occurrence of PA reported or not.
Method:

1: In each step automaton M' has at most $m+1$ active states (see Fig. 4.7), then (potentially) large structure of M' can be substituted by a linear array $E[0..m-1]$ indicating the active states and for each of them the minimal number of errors found.
2: **for** $y = m'$ **to** n' **do** { no occurrence above row m' is possible for this type of matching }
3: Make all $E[q]$ inactive { each row treated individually }
4: **for** $x = 1$ **to** n **do**
5: Make $E[0]$ active, $E[0] = 0$ { $q'_{0,0}$ is always active }
6: **foreach** active state q in E **do**
7: $err = TA'[x, y, R[q+1]]$
8: **if** $err = \oslash$ **then** $err = m'$ **end**
9: $err = err + E[q]$
10: **if** $err \leq k$ **then**
11: **if** $q + 1 = m$ **then**
12: **print** "PA found with err mismatches, its origin is at $(x-m+1, y-m'+1)$"
13: **else**
14: $E[q+1]$ will be active, $E[q+1] = err$
15: **end**
16: **else**
17: $E[q+1]$ will be inactive in the next step
18: **end**
19: **end**
20: **end**
21: **end**

Proposition 4.18
A simulation of automaton M' using Alg. 4.6 for the pattern matching of the representing string has $\mathcal{O}(mN_{TA} - nN_{PA})$ asymptotic time complexity.

Proof
Algorithm 4.6 has clear structure: if we observe its *for*-cycles, we can establish the time complexity as follows: $\mathcal{O}((n'-m')n(m+1)) = \mathcal{O}((n'-m')(mn+n)) = \mathcal{O}(mnn'+nn'-mm'n-m'n) = \mathcal{O}(mnn' - mm'n) = \mathcal{O}(mN_{TA} - nN_{PA})$. □

Note 4.19: Without optimisation of the *for*-cycle in step 2, Alg. 4.6, the asymptotic time complexity would be $\mathcal{O}(mN_{TA})$.

Theorem 4.20
The described implementation of our method of the 2D approximate pattern matching with finite automata using the 2D Hamming distance has asymptotic time complexity $\mathcal{O}(N_{TA}N_{PA})$.

Proof
Let us follow the steps of Alg. 4.1:
1-2: FA $M(\Pi)$ is composed of at most m sub-automata of type *SFORCO* and their independent simulation over TA requires $m \cdot \mathcal{O}(m'nn')$.
3: Pattern R is constructed in time $\mathcal{O}(m)$ from the relation between the final states of simulated $M(\Pi)$ and columns of PA.
4-5: Using Alg. 4.6 for simulation of M' we obtain $\mathcal{O}(mnn' - mm'n)$ time complexity (Proposition 4.18).
Overall asymptotic time complexity is then $\mathcal{O}(mm' \cdot nn' + m + mnn' - mm'n) = \mathcal{O}(N_{PA}N_{TA})$. □

Proposition 4.21
The space complexity of the basic version of the 2D approximate pattern matching using the 2D Hamming distance presented above is $\mathcal{O}(mN_{TA})$.

Proof
Each of dynamic programming matrices for m patterns of set Π needs $\mathcal{O}(m')$ space, because in the preprocessing phase we search for strings of length m' in columns of TA. The largest structure is the preprocessed array TA' that requires $\mathcal{O}(mnn')$ space. In the next phase TA' is still needed, but simulation of M' requires only $\mathcal{O}(m)$ space.

The preprocessing phase is more space-consuming, so the presented version of 2D approximate pattern matching simulation has $\mathcal{O}(mm' + mnn') = \mathcal{O}(mN_{TA})$ asymptotic space complexity. □

4.4.3 Space-efficient 2D approximate pattern matching using the 2D Hamming distance

First optimisation of time complexity we already did in Alg. 4.6, step 2. In the following, let us closely look at improvements of the space complexity.

The space complexity of the 2D approximate pattern matching using the 2D Hamming distance has similar properties that have been discussed in Sec. 4.3.3, so let us present the idea very quickly. In the following we will use the same principle to obtain much more favourable space complexity $\mathcal{O}(mn + N_{PA}n)$.

From Proposition 4.21 we see immediately that the largest structure is preprocessed text array TA'. It is three dimensional and it will likely be excessively large.

According to the idea from Sec. 4.3.3, we have to be able to restart a run of finite automaton $M(\Pi)$ in columns of TA. We use the dynamic programming to simulate sub-automata of $M(\Pi)$, so $\mathcal{O}(mm')$ space is needed to store current state of the dynamic programming (state of $M(\Pi)$) in each row of TA. Once one row of TA' is computed, it is possible to do matching in it using the simulation algorithm of automaton M' to find possible 2D occurrences. Then the row's space can be reused, eliminating effectively the need to store whole array TA'.

Theorem 4.22
The space complexity of the optimised version of the 2D approximate pattern matching using the 2D Hamming distance is $\mathcal{O}(nN_{PA})$.

Proof

1-2: We have n columns of TA, in each of them run m_d, $m_d \leq m$, dynamic programming simulations of $M(\Pi)$ sub-automata in $\mathcal{O}(mm')$ space each, that is $\mathcal{O}(mm'n)$. To store the results of simulation in each column $\mathcal{O}(m)$ space is needed, that is $\mathcal{O}(mn)$.
3: We also have to store an assignment of PA columns to the sub-automata of $M(\Pi)$ in the representing string with the space complexity $\mathcal{O}(m)$.
4-5: Simulation algorithm of M' has asymptotic space complexity $\mathcal{O}(m)$.
Since steps 2 and 5 of Alg. 4.1 are interleaved, all space requirements have to be summed up and the resulting asymptotic space complexity is $\mathcal{O}\left(mm'n + mn + 2m\right) = \mathcal{O}\left(nN_{PA}\right)$. □

4.5 Concluding remarks

In this chapter, the two-dimensional exact and approximate pattern matching using finite automata has been presented.

The simulation-based implementation of our two-dimensional approximate pattern matching works in time independent of the number of allowed mismatches, as expected. The Ukkonen's optimisation of dynamic programming significantly reduces the running time if the number of mismatches $k < m$. By the nature of this optimisation, the reduction is larger for smaller number of mismatches k and for large alphabets. However, the implementation that uses dynamic programming needs to check every element of the text array with every element of each of unique columns of the pattern array (at least if $k \geq m$). This slows the processing down and it is thus significantly slower in comparison to asymptotically optimal (for certain k) Ranka and Heywood algorithm [RH91].

5 Dagology/Operations Over Linear Representation of Acyclic Graphs

STRINGOLOGY is commonly known as a popular nickname for a discipline dealing with string or text algorithms [CR02, Preface]. In the following pages, a new branch of study is presented, so called *Dagology*. Its name clearly suggests the main tool for representing objects we are interested in: Directed Acyclic Graphs (*DAG*, Def. 2.76) and trees.

As the research in this area is at its beginning, along the topics presented we will encounter many open questions and problems. However, our results and observations described in this chapter will serve as a base for further research and development.

This chapter begins with a short motivation, then less formal, example based, approach is taken to explain the methods and finally a formal algorithms and necessary propositions are given.

5.1 Motivation

In the summary of previous works, Sec. 3.5.1, there were presented methods of multidimensional pictures representation and, particularly, structures for their indexing. In case of two-dimensions, square pictures can be indexed using generalized PATRICIA-tree [Gon88], *L-suffix* or *I-suffix* trees [GG97b, KKP98]. Kim, Kim, and Park in [KKP98] described method how *I-suffix* trees can be extended to index multidimensional structures. As an example, they gave *Z-suffix* tree for cubic (3D) pictures.

The latter construction algorithms are fast, since they construct the indices with only logarithmic multiplicative penalty, e.g. for $(n \times n)$ matrix the asymptotical time complexity is $\mathcal{O}(n^2 \log n)$.

For arbitrary sized matrices, Giancarlo and Grossi [GG96, GG97b] presented a two-dimensional suffix tree made of two-dimensional characters. They proved the lower bound on the number of nodes of the suffix tree for 2D pictures: $\Omega(n^2 n')$, where $(n \times n')$ is the size of a picture, and $n \leq n'$. These suffix trees can be built in $\mathcal{O}(n^2 n' \log n')$ time and stored in optimal $\mathcal{O}(n^2 n')$ space.

Our aim is to have a prefix, a suffix, and a factor automaton for pictures. Their construction will be presented in this chapter. These indexing tools are based on a tree-like representation of a picture and therefore they require a pushdown store. Proposed methods are completely new and there are many open questions remaining.

5.2 Overview of our approach

Dagology in general takes inspiration from two sources: the first one are the advances of arbology, which were partially illustrated in Sec. 3.2. The second one is the practical problem of phylogenetic tree description. This topic is covered in the same chapter, in Sec. 3.3. The existence of gene transfer among species located in different subtrees of the phylogenetic tree

makes a DAG from such a tree and therefore, it can no longer be treated as a tree by, for example, tree pattern matching algorithms. However, in practice, the edges are never inserted arbitrarily, the resulting graph remains to be acyclic, and this is the property we need. Hence another nickname for the field of study we are treating now: *acyclistics*.

The methods described in this work have taken one more particular inspiration, namely from one-dimensional text prefix, factor and suffix (DAWG) automata. These automata are used to index the text and then they allow to search for any prefix, factor, or suffix, respectively, in time linear to the length of the pattern, and not depending on the length of the text being indexed.

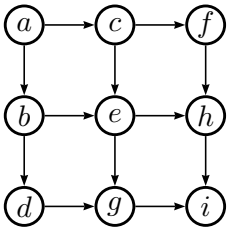
Now let us describe basic principles which our method is built upon. Our method uses vertex splitting (noted in Sec. 3.5) to build a tree representing the picture. *Vertex splitting* is our informal name for a technique, where (some) nodes of an array or a tree may appear multiple times in resulting data structure. Such a technique is not new, however. As an example, let us recall the construction of [KKP98], recursively dividing picture into sub-pictures, starting at the origin and taking bottom and right sub-picture with the respect to the current vertex. We use the same principle.

Let us demonstrate the idea in the following example.

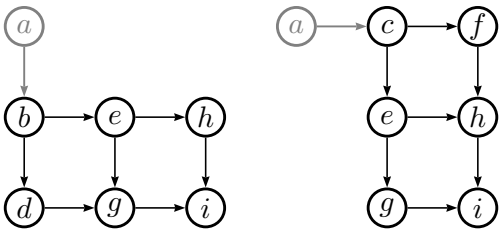
Example 5.1

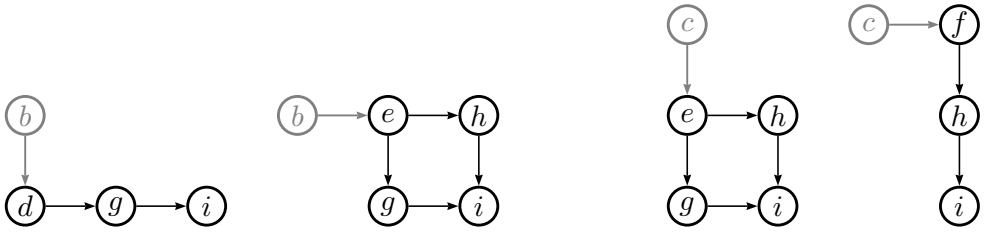
We are given picture P over alphabet A of size (3×3). $P = \begin{array}{|c|c|c|}\hline a & c & f \\ \hline b & e & h \\ \hline d & g & i \\ \hline \end{array}$.

Elements of P can be interpreted as vertices of a labeled graph. Let the vertices be connected by arcs, starting from the picture origin (Def. 2.97) and connecting two closest neighbouring vertices of each vertex to the right and to the bottom of the graph, if any.



Now, the vertex splitting will be performed. There are two sub-pictures for vertex a, as indicated in the figure. Then the same is done recursively for the resulting bottom and right sub-pictures with respect to their origins, b and c, respectively.

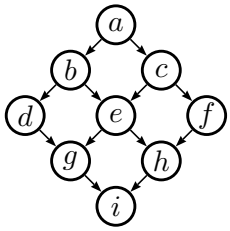




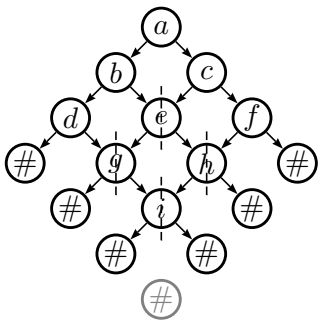
The algorithm continues until sub-pictures (1×1) are obtained, vertices labeled by i in this case. Let us show all remaining steps for the leftmost subtree, note that only the right context is available now. The rightmost subtree is analogous, but only bottom context is available.

The square sub-picture is the remaining part. It is easy to divide it now, we show it just for the sake of completeness:

Let us show another explanation, where vertex splitting principle will be demonstrated. If TA is rotated by $-\frac{\pi}{4}$ with the origin as the center of rotation, we obtain:



Now, let us take the origin a as the root of the tree and, starting from the root, transform recursively all topmost vertices shared by two subtrees by duplicating them and adding edges as appropriate. For practical reasons, let us add some extra nodes labeled by special symbol $\#$, $\# \notin A$. These will serve as leaves of the tree we are going to build. The algorithm of this extension is presented in Alg. 5.1. Alphabet of the extended picture is $A_{\#} = A \cup \{\#\}$. $A_{\#}$ is a ranked alphabet (Def. 2.14), where arity of all symbols a, $a \in A$, is defined as $\text{arity}(a) = 2$, and $\text{arity}(\#) = 0$.



Extended picture TA', $TA' \in A_{\#}^{(4\times4)}$ is $TA' =$

a	c	f	#
b	e	h	#
d	g	i	#
#	#	#	

Note that the dimmed, unconnected, #-state is useless and hence it will not be considered a part of the extended picture.

The resulting tree of extended picture TA' will be denoted by tree(TA') and it is depicted in Fig. 5.1.

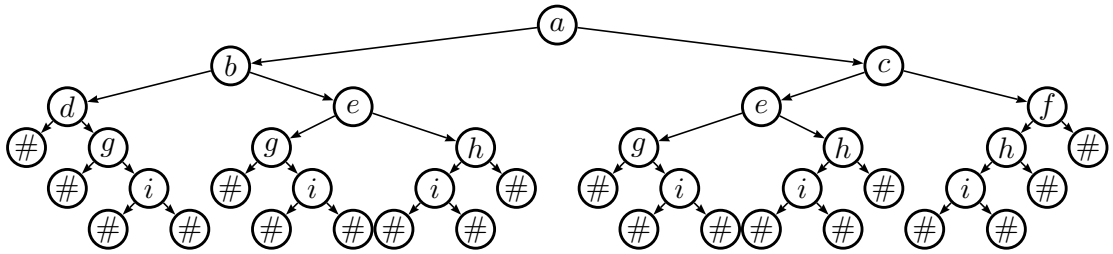


Figure 5.1. tree(TA')

In the last step of this example, let us construct a prefix notation of the tree in Fig. 5.1, that is pref(tree(TA')). Using Alg. 3.6, we obtain

$$\text{pref}(\text{tree}(TA')) = abd\#g\#i\#\#eg\#i\#\#hi\#\#\#ceg\#i\#\#hi\#\#\#fhi\#\#\#\#\,.$$

5.2.1 Picture-to-tree transformation

Let us have a two-dimensional picture over alphabet A. First of all, we extend this two-dimensional picture with a special border symbol #, $\# \notin A$. This is commonly used in several other methods, namely as an input for two-dimensional on-line tessellation acceptors [IN77, GR97], mentioned in Sec. 3.4.2. We use this special symbol to detect the end of a column and end of a row of a picture.

In Alg. 5.1, a picture is extended only with minimum required additional symbols #. Since our method has no use for the top or left border made of symbol #, we avoid to augment the picture with unneeded borders.

Algorithm 5.1: Construction of an extended 2D picture
Input: Let P be a picture, R be its minimal bounding array, $R \in A^{(n\times n')}$. Let $A_{\#} = A \cup \{\#\}$ be the ranked alphabet, where arity(#) = 0, arity(a) = 2 for all symbols a, $a \in A_{\#}, a \neq \#$.
Output: Picture P' extended on its right and bottom border with symbol #, R' be its minimal bounding array, $R' \in A_{\#}^{((n+1)\times(n'+1))}$. (Element $[n+1, n'+1]$ is left undefined, it is never used.)
Method:
$P'[n+1, y] = \#$ for all y, where $1 \leq y \leq n'$,
$P'[x, n'+1] = \#$ for all x, where $1 \leq x \leq n$.

Next, algorithm 5.2 for transformation of a picture into a tree is presented, its application to picture P will be denoted by tree(P).

Algorithm 5.2 is correct: it starts at the picture origin and recursively traverses the bottom and the right sub-pictures. Since the bottom row and the rightmost column are filled with symbols #, the recursion ends right at the leaves of the tree being constructed.

Lemma 5.2
Let t be a tree constructed by Alg. 5.2. The maximal depth of any node in t, depth(t), is depth(t) = $n + n' - 1$.

Proof
First, let us note some facts: The longest path in t starts at the root of t and follows the bottom context at each splitting position. At most $n' - 1$ of such splittings lead to node labeled by the

Algorithm 5.2: Construction of a tree representation of a two-dimensional picture
Input: Picture P extended with symbol # (Alg. 5.1), $P \in A_{\#}^{**}$, $P \neq \lambda$.
Output: Binary ordered labeled tree t, each node of t be labeled by a symbol s, $s \in A_{\#}$.
Description: Label, left and right children of node v are assigned as a triplet: (label, left child, right child). The algorithm starts at the origin of P. The application of the algorithm with P being its input will be denoted by tree(P).
Method:

$$\text{createnode}(x,y) = \begin{cases} (P[x,y], \text{createnode}(x,y+1), \text{createnode}(x+1,y)) & \text{, if } P[x,y] \neq' \#', \\ (P[x,y], nil, nil) & \text{, otherwise,} \end{cases}$$

$x = 1, y = 1,$
$t = (P[x,y], \text{createnode}(x,y+1), \text{createnode}(x+1,y))\,.$

following element of the first column of the picture, the n'-th leads to a leaf #. At $n'-1$-st splitting we continue to the right context, after n steps a leaf is reached. $(*)$

Now we will continue by induction on the size of the original un-extended picture: $(n \times n')$.

1. Assume $n = n' = 1$. The binary ordered labeled tree constructed by Alg. 5.1 and Alg. 5.2 from the picture (1×1) has one node with the origin of the picture and two leaves labeled by #. $\text{depth}(t) = n + n' - 1 = 1$. The statement holds.
2. Assume the statement holds for all n, $n \leq c$, $c \in \mathrm{N}$, n' fixed. We have to prove the statement also holds for $n+1$.
   Induction step, by dimensions:
   As shown, adding one column will prolong the sequence of right contexts followed in $(*)$, thus the length of it is $n' - 1 + n + 1 = n + n'$. The statement gives $n' + (n+1) - 1 = n + n'$.
   Assume the statement holds for all n', $n' \leq c$, $c \in \mathrm{N}$, n fixed. We have to prove the statement also holds for $n' + 1$.
   Adding one row will prolong the sequence of bottom contexts followed in $(*)$, thus the length of it is $n' + 1 - 1 + n = n + n'$. The statement gives $(n' + 1) + n - 1 = n + n'$.

The lemma holds. □

Lemma 5.3
The asymptotical time complexity of Alg. 5.2 for construction of the tree representation of a picture is $\mathcal{O}(2^{\text{depth}(t)})$.

Proof
First, let us remind that the number of nodes of a complete binary tree is $2^{\text{depth}(t)-1} + 2^{\text{depth}(t)}$. Since all non-leaf nodes of the tree being constructed have exactly two children, it is clear that the maximum number of nodes is limited by the number of nodes of a complete binary tree. □

In Sec. 3.2, there were mentioned basic properties of trees in the prefix notation. Let us extend these results to our representation of pictures in the prefix notation.

For the following three lemmas:
Let *TA* be a picture and *TA*′ be *TA* extended by Alg. 5.1. Let t be a tree constructed by Alg. 5.2, such that $t = \text{tree}(TA')$. Let *PA* be a sub-picture of *TA*, s be a tree made over extended *PA*.

Lemma 5.4
For every $PA \sqsubseteq TA$, such that *PA* is a 2D prefix of *TA*, it holds that pref(s) is a subsequence of pref(t) in general and its first symbol is the first symbol of pref(t). (Treetop matching may be used.)

Proof
Let the number of rows of *PA* be smaller than that of *TA* and $\text{pref}(s) = a\,\text{pref}(s_1)\,\text{pref}(s_2)$, $\text{pref}(t) = a\,\text{pref}(t_1)\,\text{pref}(t_2)$, $a \in A$ be the origin of the picture. Since subtrees s_1 and t_1 are of different height, $\text{pref}(s_1) \neq \text{pref}(t_1)$ and there will be the prefix notation of other subtrees interleaved. pref(s) is therefore a subsequence of pref(t).

Since every 2D prefix must contain the origin, the first symbols of a 2D prefix and of the picture match. □

Lemma 5.5
For every $PA \sqsubseteq TA$, such that PA is a 2D suffix of TA, it holds that $\mathrm{pref}(s)$ is a substring of $\mathrm{pref}(t)$. (Subtree matching may be used.)

Proof
From Def. 2.105 we know the last element of the last row of a picture must be included in any 2D suffix of the picture.

Let us show the statement by induction on the height of the subtree.

Assume $\mathrm{height}(s') = 1$. Subtree s' represents a picture of one element, say a, and $\mathrm{pref}(s') = a\#\#$. Since all symbols except # of ranked alphabet $A_{\#}$ have arity 2, and # has arity 0, s' is a tree and the statement holds for it.

Let b be an element of a picture, its position be different from element labeled by a. Assume the statement holds for two subtrees s_1, s_2, $\mathrm{height}(s_i) = k - 1$, $k > 2$, $i \in \{1, 2\}$, such that s_1 represents the bottom context of element b and s_2 the right one. Now we have to verify whether the statement holds also for a tree s_3, $\mathrm{height}(s_3) = k$, that is created using b as a new root, s_1, s_2 be its left and right subtree, respectively.

Since $\mathrm{pref}(s_3) = b\,\mathrm{pref}(s_1)\,\mathrm{pref}(s_2)$, prefix notation of s_3 is a substring of t and the lemma holds. □

Lemma 5.6
For every $PA \sqsubseteq TA$, such that PA is a 2D factor of TA, it holds that $\mathrm{pref}(s)$ is a subsequence of $\mathrm{pref}(t)$. (Tree template matching may be used.)

Proof
The argumentation in this case would be similar to 2D prefix case. If the 2D factor is smaller than the 2D text, its prefix notation indeed appears as a subsequence of $\mathrm{pref}(t)$. □

5.3 Two-dimensional pattern matching in pictures in the tree representation

In this short section we will use the representation of a picture, based on the tree notation, presented in Sec. 5.2.1. There are numerous methods for various kinds of matching in trees, they are described in [CDG+07, Cle08], or it is possible to use references in the papers listed in Sec. 3.2.1, that is [JM09, Jan09, FJM09, JM10].

Another possibility is to use the prefix notation of the tree transformed pictures. This variant allows to use some of the methods developed in [JM10]. The subtree automata are described also in [Jan09]. This approach is more convenient to the global aim in this work, that is matching using automata. In this case the automata are pushdown automata. Specifically for trees in prefix notation, the tree pattern pushdown automata (from [JM10]) are analogous to factor automata used in string matching, they make an index from the tree (in our case a 2D text is transformed into a tree) in the prefix notation and then the pattern in the prefix notation is searched in it. The tree pattern pushdown automaton finds the rightmost leaves of all occurrences of a tree pattern in the subject tree. The most significant advantage over other tree algorithms is that for given input tree pattern of size m, the tree pattern PDA performs its searching in time linear in m, independently of the size n of the subject tree. This is faster than the time that could be theoretically achieved by any standard tree automaton, because

the standard deterministic tree automaton runs on the subject tree. As a consequence, the searching performed by the tree automaton can be linear in n at best.

The tree pattern PDA is by construction an input-driven PDA (Def. 2.61). Any input-driven PDA can be determinised [WW01].

However, there exists a method constructing deterministic tree pattern pushdown automata directly from tree patterns [FJM09].

Using some tree algorithm or tree matching pushdown automata, in either case the size of the tree representation of a picture will be prohibitive. In the former case, it will influence the time of matching, in the latter, the pushdown automaton becomes large. The redundancy in the PDA is caused by repeated subtrees in the tree representation.

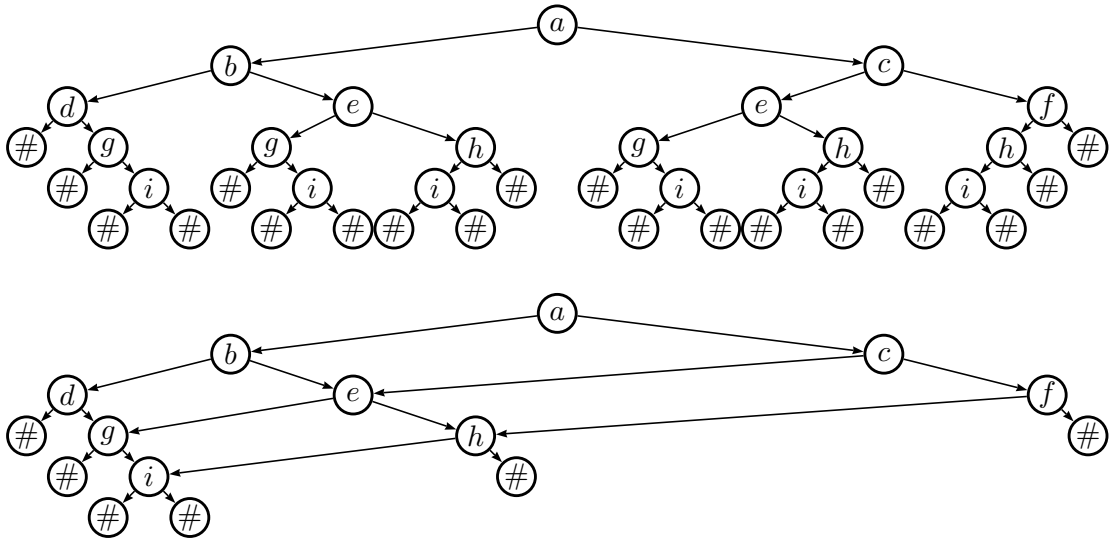
In the following, we will show a pushdown automata based model for picture indexing, addressing some of the problems.

5.4 Indexing of two-dimensional picture in the prefix notation

In this section, a pushdown automata based picture index will be presented. It reuses its own parts in a manner resembling the lateral gene transfer links of Newick notation. This means some part of a genome of some species is reused in the genome of some other species, non-related in terms of the standard line of ancestry.

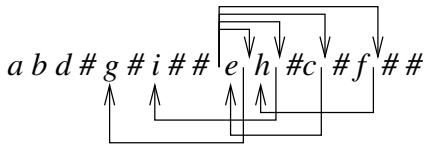
Example 5.7

Let us illustrate the idea reusing Fig. 5.1 below. The high number of redundant subtrees, induced by the picture-to-tree transformation, can be eliminated by redundant subtree removal and by addition of edges leading to roots of appropriate subtrees. These additional transitions make a DAG from the tree, however.



Clearly, all paths available in the tree are available in the directed acyclic graph as well. All nodes have the same arity as in the tree. The number of nodes of the DAG is exactly the same as the number of elements of the text array being represented. Our pushdown automata construction is based on these properties.

One can imagine the redundancy can be eliminated even in the prefix notation, using set of links in the text:



Another interesting property is that the pushdown automaton construction does not need the tree representation of picture constructed and available. The pushdown automaton can be constructed from the picture directly. Since the tree representation of a 2D pattern can also be generated on the fly, the only significant extra data structure in memory will be the indexing pushdown automaton. On the other hand, we pay for the reusability of pushdown automata components by increase in the number of pushdown store symbols. This converts the category of the PDA, since it is no longer input-driven. It is not known yet if it is possible to transform these pushdown automata to e.g. visibly pushdown automata (Def. 2.66) to be able to make them deterministic.

5.4.1 Two-dimensional pattern matching using pushdown automata

At first, let us recapitulate the steps of an algorithm for 2D picture indexing. Some of the steps are rather formal, as noted below.

1. Extend 2D text and 2D pattern to provide easy identification of their borders (Alg. 5.1). (This step is formal.)
2. Construct an index of a 2D text in the form of a pushdown automaton (Alg. 5.6).
3. Construct a tree from the extended 2D pattern (Alg. 5.2). (This step is formal.)
4. Construct a prefix notation from the tree obtained in the previous step (Alg. 3.6). (This step is formal, the prefix notation including the # symbols can be done on the fly by the algorithm mentioned.)
5. A pushdown automaton then searches for linearised 2D pattern in the indexed 2D text, determining possible 2D occurrences of *PA* in *TA*.

In this section, a construction of picture indexing automata will be discussed. These automata allow to accept all 2D prefixes, 2D factors and 2D suffixes of a given 2D text in the prefix notation. The deterministic indexing pushdown automaton accepts the prefix notation of a tree made of 2D prefix, 2D factor, or 2D suffix, in time linear to the number of nodes of the tree. The type of 2D pattern accepted is given by construction of the pushdown automaton.

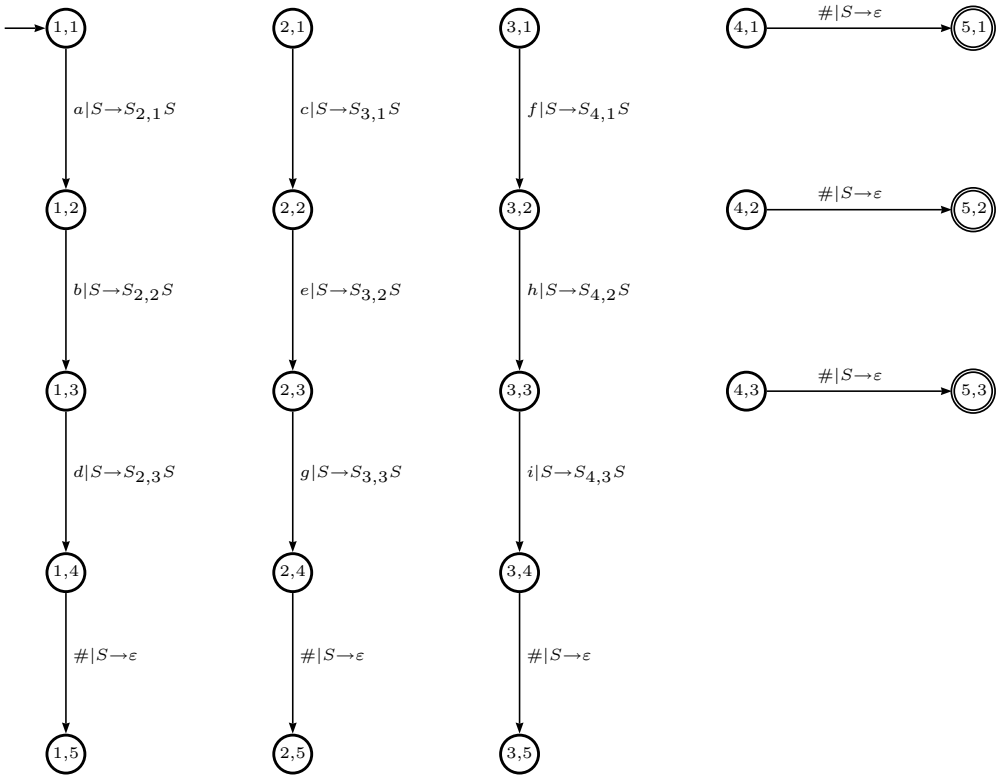
5.4.1.1 *Construction elements*

The construction of a pushdown automaton for 2D text indexing is based on linearisation of a 2D array with adjacency relation among array elements preserved.

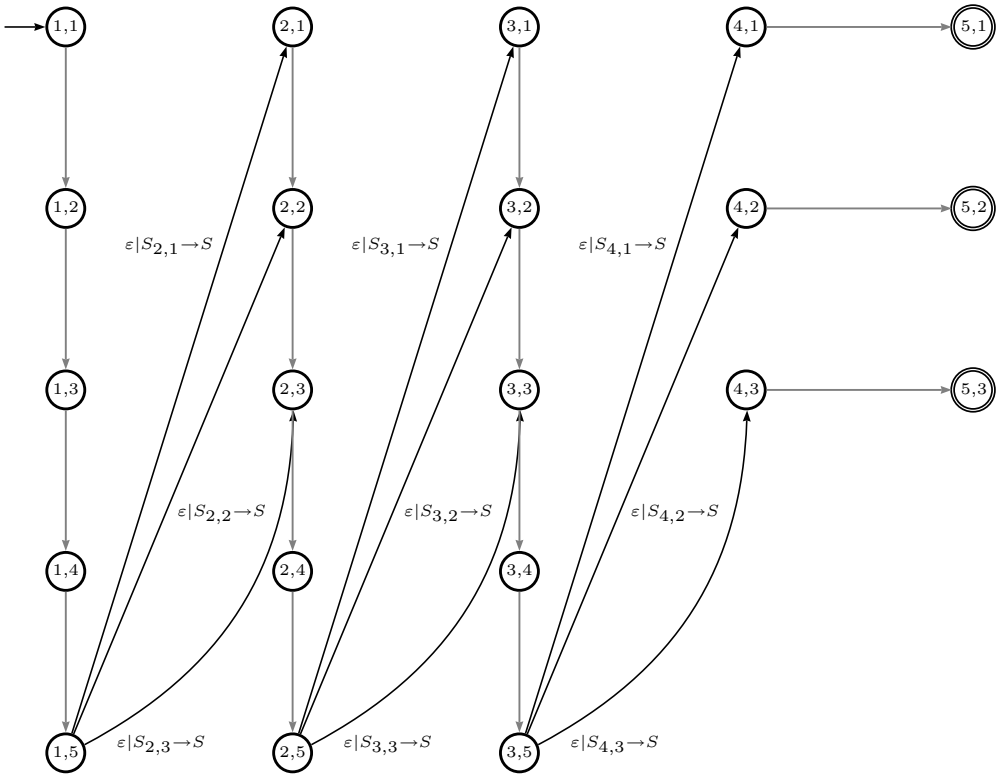
First algorithm in this section constructs a base for three variants of pushdown automata presented later.

Algorithm 5.3: Pushdown automaton accepting the picture in prefix notation
Input: Text array TA extended with symbol #, $TA \in A_{\#}^{(n+1\times n'+1)}$.
Output: Automaton M, $M = (Q, A_{\#}, G, \delta, q_{1,1}, S, \emptyset)$, accepting language $L_\varepsilon(M)$, $L_\varepsilon(M) = \{\text{pref}(\text{tree}(TA))\}$.
Method:
$\delta = \emptyset$, $Q = \emptyset$,$D = \{S\}$,
$D = D \cup \{S_{x+1,y}\}$ for all x, y, where $1 \leq x < n$, $1 \leq y < n'$,
{ `create_nodes` }
$Q = Q \cup \{q_{x,y}\}$ for all x, y, where $1 \leq x \leq n+1$, $1 \leq y \leq n'+1$ and $y < n' \vee x < n$,
{ `create_basic_transitions` }
$\delta(q_{x,y}, TA[x,y], S) = \delta(q_{x,y}, TA[x,y], S) \cup \{(q_{x,y+1}, S_{x+1,y}S)\}$ for all x, y, where $1 \leq x < n$, $1 \leq y < n'$,
{ `create_hash_transitions` }
$\delta(q_{n,y}, \#, S) = \delta(q_{n,y}, \#, S) \cup \{(q_{n+1,y}, \varepsilon)\}$ for all y, where $1 \leq y < n'$,
$\delta(q_{x,n'}, \#, S) = \delta(q_{x,n'}, \#, S) \cup \{(q_{x,n'+1}, \varepsilon)\}$ for all x, where $1 \leq x \leq n$,
{ `create_bottom_return_transitions` }
$\delta(q_{x,n'+1}, \varepsilon, S_{x+1,y}) = \delta(q_{x,n'+1}, \varepsilon, S_{x+1,y}) \cup \{(q_{x+1,y}, S)\}$ for all x, y, where $1 \leq x < n$, $1 \leq y < n'$,
{ `create_right_return_transitions` }
$\delta(q_{n+1,y}, \varepsilon, S_{x,y-1}) = \delta(q_{n+1,y}, \varepsilon, S_{x,y-1}) \cup \{(q_{x,y-1}, S)\}$ for all x, y, where $2 \leq x \leq n$, $2 \leq y < n'$.

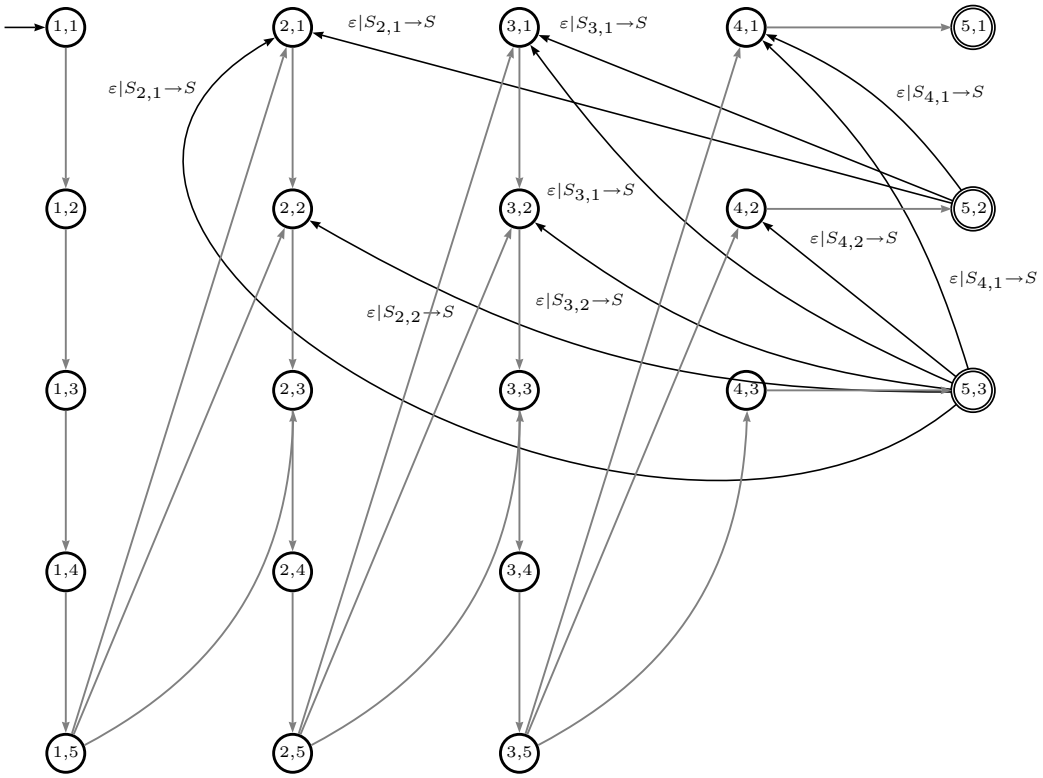
Example 5.8 (`create_nodes` + `create_basic_transitions` + `create_hash_transitions`)
To illustrate the work of Alg. 5.3, let us take again the picture P from example 5.1 in this and the following examples. For the sake of clarity, the construction steps are depicted in several figures, because it might be more difficult to pair the labels with appropriate transitions in the complete figure, presented in Example 5.11. The operation of nodes creation with basic set of transitions added results in the following transition diagram:



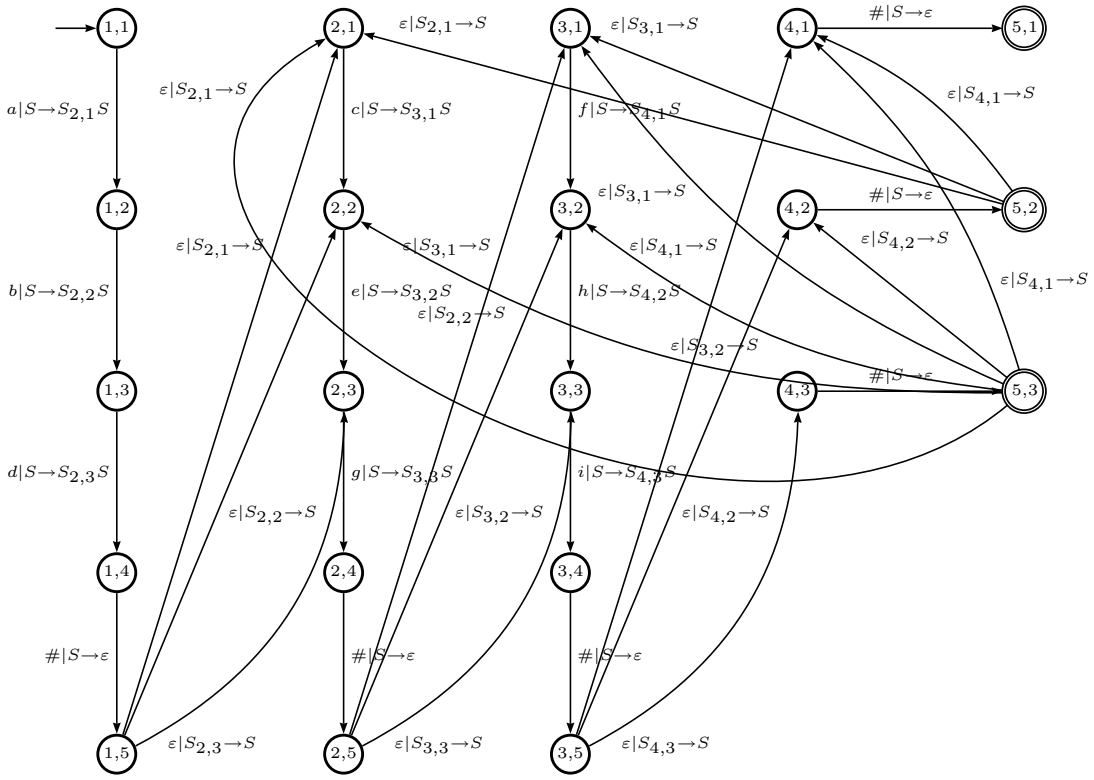
Example 5.9 (`create_nodes` + `create_bottom_return_transitions`)
The transitions representing end of each column are depicted in this example, all transitions from the previous example are dimmed and their labels removed.



Example 5.10 (`create_nodes` + `create_right_return_transitions`)
In this example, the transitions representing end of each row are depicted, all transitions from the previous examples are dimmed and their labels removed.



Example 5.11
Pushdown automaton M, $M = (Q, A, G, \delta, q_{1,1}, S, \emptyset)$, constructed by Alg. 5.3.



The following algorithms add transitions allowing the automaton M to accept the prefix notation of trees representing 2D suffixes and 2D prefixes.

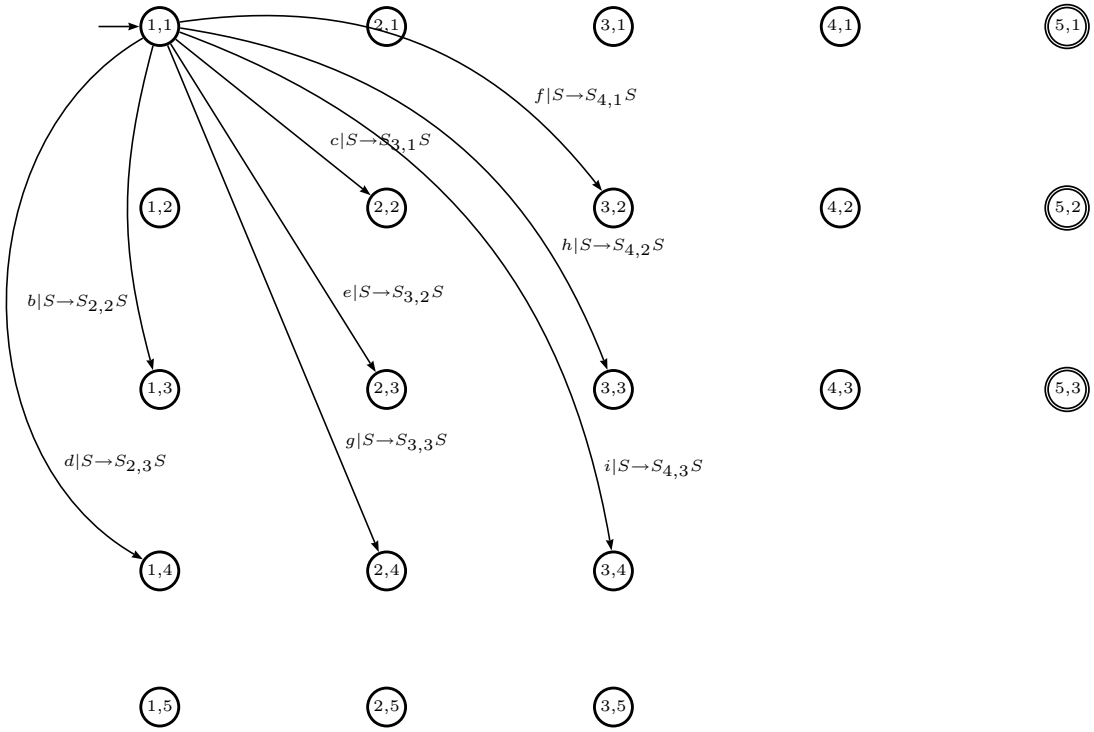
Algorithm 5.4: Suffix transitions

Input: Text array TA extended with symbol #, $TA \in A_{\#}^{(n+1\times n'+1)}$. Pushdown automaton M, $M = (Q, A, G, \delta, q_{1,1}, S, \emptyset)$, constructed by Alg. 5.3.
Output: Modified pushdown automaton M.
Method:
$\delta(q_{1,1}, TA[x, y], S) = \delta(q_{1,1}, TA[x, y], S) \cup \{(q_{x,y+1}, S_{x+1,y}S)\}$ for all x, y, where $1 \leq x < n$, $1 \leq y < n'$ and $x > 1 \vee y > 1$.

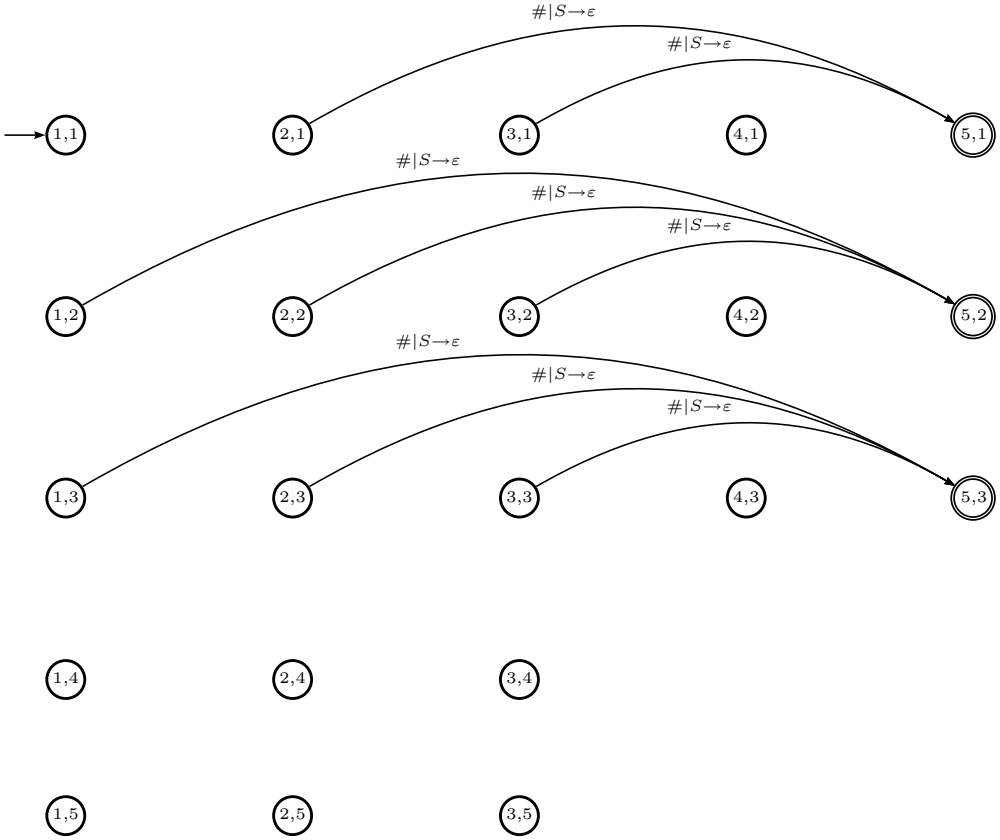
Algorithm 5.5: Prefix end transitions

Input: Text array TA extended with symbol #, $TA \in A_{\#}^{(n+1\times n'+1)}$. Pushdown automaton M, $M = (Q, A, G, \delta, q_{1,1}, S, \emptyset)$, constructed by Alg. 5.3.
Output: Modified pushdown automaton M.
Method:
$\delta(q_{x,y}, \#, S) = \delta(q_{x,y}, \#, S) \cup \{(q_{n+1,y}, \varepsilon)\}$ for all x, y, where $1 \leq x < n$, $1 \leq y < n'$ and $x \neq 1 \vee y \neq 1$.

Example 5.12 (`create_nodes` + `create_suffix_transitions`)
Using Alg. 5.4 and nodes obtained by `create_nodes` part of Alg. 5.3, we obtain the following states and transitions. These transitions allow to find the prefix notation of the tree representation of 2D suffixes of a text array being indexed.



Example 5.13 (`create_nodes` + `create_prefix_end_transitions`)
Using Alg. 5.5 and nodes obtained by `create_nodes` part of Alg. 5.3, we obtain the following states and transitions. These transitions allow to find the prefix notation of the tree representation of 2D prefixes of a text array being indexed.



5.4.1.2 *Construction of pushdown automata for 2D text indexing*

Algorithm 5.6: Construction of pushdown automaton accepting all 2D factors
Input: Text array TA extended with symbol #, $TA \in A_{\#}^{(n+1\times n'+1)}$.
Output: Automaton M, $M = (Q, A_{\#}, G, \delta, q_{1,1}, S, \emptyset)$, accepting language $L_\varepsilon(M)$, $L_\varepsilon(M) = \{w; w = \text{pref}(\text{tree}(\text{2D factor of } TA))\}$.
Method:
Create pushdown automaton M using Alg. 5.3.
Add suffix transitions to its δ using Alg. 5.4.
Add prefix transitions to its δ using Alg. 5.5.

Algorithm 5.7: Construction of pushdown automaton accepting all 2D prefixes
Input: Text array TA extended with symbol #, $TA \in A_{\#}^{(n+1\times n'+1)}$.
Output: Automaton M, $M = (Q, A_{\#}, G, \delta, q_{1,1}, S, \emptyset)$, accepting language $L_\varepsilon(M)$, $L_\varepsilon(M) = \{w; w = \text{pref}(\text{tree}(\text{2D prefix of } TA))\}$.
Method:
Create pushdown automaton M using Alg. 5.3.
Add prefix transitions to its δ using Alg. 5.5.

Algorithm 5.8: Construction of pushdown automaton accepting all 2D suffixes
Input: Text array TA extended with symbol #, $TA \in A_{\#}^{(n+1\times n'+1)}$.
Output: Automaton M, $M = (Q, A_{\#}, G, \delta, q_{1,1}, S, \emptyset)$, accepting language $L_\varepsilon(M)$, $L_\varepsilon(M) = \{w; w = \text{pref}(\text{tree}(\text{2D suffix of } TA))\}$.
Method:
Create pushdown automaton M using Alg. 5.3.
Add suffix transitions to its δ using Alg. 5.4.

The correctness of the 2D factor pushdown automaton, constructed by Alg. 5.6 and accepting 2D factors of a picture in prefix notation, is established by the following lemma.

Lemma 5.14
Let TA be a picture extended with symbol #, $TA \in A_{\#}^{(n+1\times n'+1)}$, pushdown automaton M, $M = (Q, A, G, \delta, q_{1,1}, S, \emptyset)$, be constructed by Alg. 5.6 over TA. Pushdown automaton M accepts prefix notation of any 2D factor of TA by the empty pushdown store. That is, M accepts the prefix notation of any $TA[i..k, j..l]$, $1 \le i < k \le n$, $1 \le j < l \le n'$.

Proof
We will proceed by induction on the height of the tree t constructed by Alg. 5.2 from the given 2D factor.

1. If the 2D factor has only one element, $TA[i, j]$, its arity is two (Note 2.15). This picture is to be extended by two nullary symbols #, Alg. 5.1. Extended picture is $\begin{array}{|c|c|}\hline TA[i,j] & \# \\ \hline \# & \\ \hline \end{array}$ and the corresponding tree t, constructed by Alg. 5.2, has height$(t) = 1$, pref$(t) = TA[i, j]\#\#$. There exists a transition (suffix link) $(q_{i,j+1}, S_{i+1,j}S) \in \delta(q_{1,1}, TA[i, j], S)$.
   Therefore, $(q_{1,1}, TA[i, j]\#\#, S) \vdash_M^+ (q_{n+1,j}, \varepsilon, \varepsilon)$ and pushdown automaton M accepts the smallest 2D factor by the empty pushdown store.
2. Assume the statement of the lemma holds also for two 2D factors f_b and f_r, $f_b = TA[i..k, j+1..l]$, $t_r = TA[i+1..k, j..l]$. Let $t_b = \text{tree}(f_b)$, $t_r = \text{tree}(f_r)$ and height$(t_b) \le m$, height$(t_r) \le m$, $m \ge 1$.
   We have to prove the statement holds also for the prefix notation constructed from tree t, $t = TA[i, j]\,\text{pref}(t_b)\,\text{pref}(t_r)$, height$(t) \ge m + 1$.

Since there exists a transition (suffix link) $(q_{i,j+1}, S_{i+1,j}S) \in \delta(q_{1,1}, TA[i,j], S)$, pushdown automaton M goes through the following configurations:
$(q_{1,1}, TA[i,j]\,\mathrm{pref}(t_b)\,\mathrm{pref}(t_r), S) \vdash_M$
$(q_{i,j+1}, \mathrm{pref}(t_b)\,\mathrm{pref}(t_r), S_{i+1,j}S) \vdash_M^+$
$(q_{n+1,j+1}, \mathrm{pref}(t_r), S_{i+1,j}) \vdash_M$
$(q_{i+1,j}, \mathrm{pref}(t_r), S) \vdash_M^+$
$(q_{n+1,j}, \varepsilon, \varepsilon)$.
Pushdown automaton M accepts the 2D factor by the empty pushdown store and the proof is complete. □

The correctness of the 2D prefix pushdown automaton, constructed by Alg. 5.7 and accepting 2D prefixes of a picture in prefix notation, is established by the following lemma.

Lemma 5.15
Let TA be a picture extended with symbol #, $TA \in A_{\#}^{(n+1\times n'+1)}$, pushdown automaton M, $M = (Q, A, G, \delta, q_{1,1}, S, \emptyset)$, be constructed by Alg. 5.7 over TA. Pushdown automaton M accepts prefix notation of any 2D factor of TA by the empty pushdown store. That is, M accepts the prefix notation of any $TA[1..k, 1..l]$, $1 \le k \le n$, $1 \le l \le n'$.

Proof
According to Def. 2.104, any 2D prefix contains the element $TA[1,1]$.

We will proceed by induction on the height of tree t constructed by Alg. 5.2 from the given 2D prefix.

1. If the 2D prefix has only one element, $TA[1,1]$, its arity is two (Note 2.15). This picture is to be extended by two nullary symbols #, Alg. 5.1. Extended picture is $\begin{array}{|c|c|}\hline TA[1,1] & \# \\ \hline \# & \\ \hline \end{array}$ and the corresponding tree t, constructed by Alg. 5.2, has $\mathrm{height}(t) = 1$, $\mathrm{pref}(t) = TA[1,1]\#\#$. There exists a (basic) transition $(q_{1,2}, S_{2,1}S) \in \delta(q_{1,1}, TA[1,1], S)$.
Therefore, $(q_{1,1}, TA[1,1]\#\#, S) \vdash_M^+ (q_{n+1,1}, \varepsilon, \varepsilon)$ and pushdown automaton M accepts the smallest 2D prefix by the empty pushdown store.
2. Assume the statement of the lemma holds also for a prefix notation of a tree over the 2D prefix composed from element $TA[1,1]$ and two 2D factors f_{b_1} and f_{r_1}, $f_{b_1} = TA[1..k, 2..l]$, $t_{r_1} = TA[2..k, 1..l]$. Let $t_{b_1} = \mathrm{tree}(f_{b_1})$, $t_{r_1} = \mathrm{tree}(f_{r_1})$ and $\mathrm{height}(t_{b_1}) \le m$, $\mathrm{height}(t_{r_1}) \le m$, $m \ge 1$.
Let two 2D factors f_{b_2} and f_{r_2} be $f_{b_2} = TA[1..k+1, 2..l+1]$, $t_{r_2} = TA[2..k+1, 1..l+1]$. Let $t_{b_2} = \mathrm{tree}(f_{b_2})$, $t_{r_2} = \mathrm{tree}(f_{r_2})$ and $\mathrm{height}(t_{b_2}) \le m+1$, $\mathrm{height}(t_{r_2}) \le m+1$, $m \ge 1$.
We have to prove the statement holds also for prefix notation constructed from tree t, $t = TA[1,1]\,\mathrm{pref}(t_{b_2})\,\mathrm{pref}(t_{r_2})$, $\mathrm{height}(t) \ge m+1$.
Since there exists a (basic) transition $(q_{1,2}, S_{2,1}S) \in \delta(q_{1,1}, TA[1,1], S)$, pushdown automaton M goes through the following configurations:
$(q_{1,1}, TA[1,1]\,\mathrm{pref}(t_{b_2})\,\mathrm{pref}(t_{r_2}), S) \vdash_M$
$(q_{1,2}, \mathrm{pref}(t_{b_2})\,\mathrm{pref}(t_{r_2}), S_{2,1}S) \vdash_M^+$
$(q_{n+1,l+1}, \mathrm{pref}(t_{r_2}), S_{2,1}) \vdash_M$
$(q_{2,1}, \mathrm{pref}(t_r), S) \vdash_M^+$
$(q_{n+1,1}, \varepsilon, \varepsilon)$.
Pushdown automaton M accepts the 2D prefix by the empty pushdown store and the lemma holds. □

The correctness of the 2D suffix pushdown automaton, constructed by Alg. 5.8 and accepting 2D suffixes of a picture in the prefix notation, is established by the following lemma.

Lemma 5.16
Let TA be a picture extended with symbol #, $TA \in A_{\#}^{(n+1\times n'+1)}$, pushdown automaton M, $M = (Q, A, G, \delta, q_{1,1}, S, \emptyset)$, be constructed by Alg. 5.8 over TA. Pushdown automaton M accepts the prefix notation of any 2D suffix of TA by the empty pushdown store. That is, M accepts the prefix notation of any $TA[i..n, j..n']$, $1 \leq i \leq n$, $1 \leq j \leq n'$.

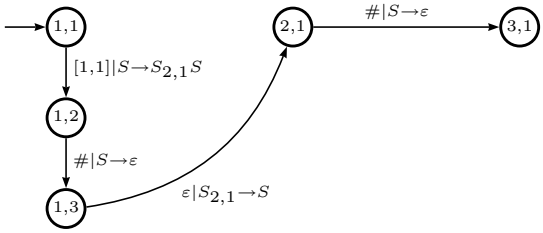
Proof
The proof of Lemma 5.14 can be applied in this context. The only change is that according to Def. 2.105 any 2D suffix contains the element $TA[n, n']$.

As pushdown automaton M accepts a 2D suffix in the prefix notation by the empty pushdown store, the proof is complete. □

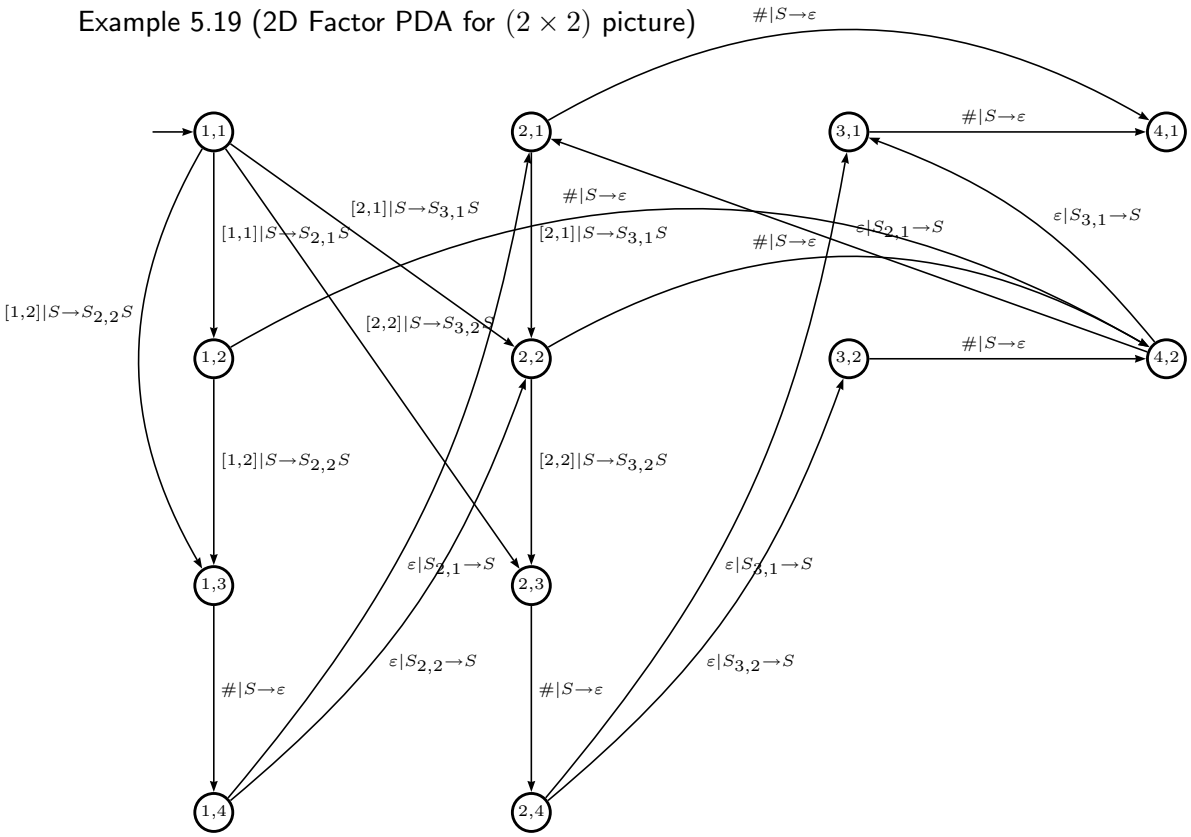
Lemma 5.17
Let TA be a picture extended with symbol #, $TA \in A_{\#}^{(n+1\times n'+1)}$, pushdown automaton M, $M = (Q, A, G, \delta, q_{1,1}, S, \emptyset)$, be constructed by Alg. 5.7 over TA. Pushdown automaton M is the deterministic pushdown automaton.

Proof
The criteria for a deterministic pushdown automaton, specified in Def. 2.60, are met by the construction. Pushdown automaton M is the deterministic pushdown automaton. □

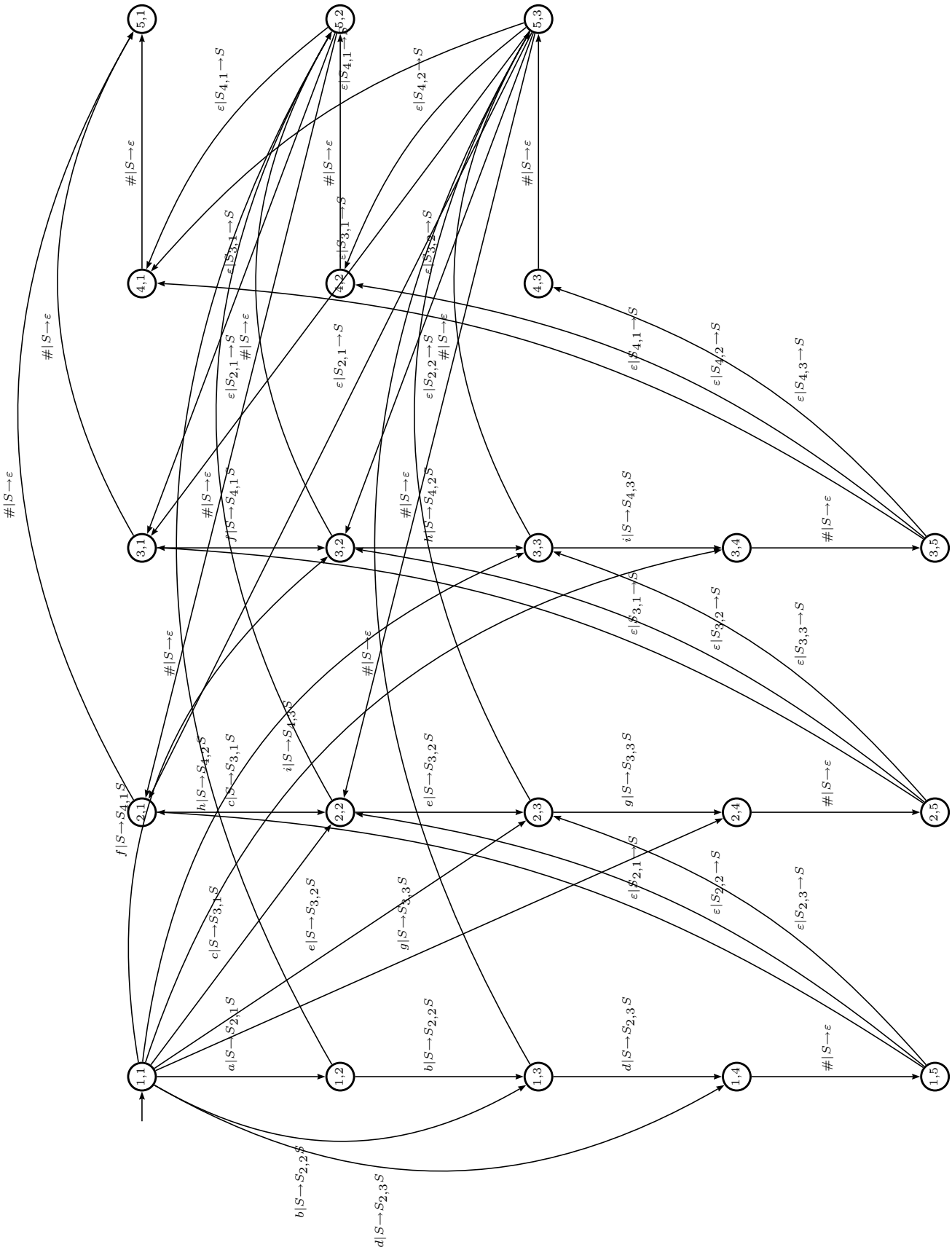
Example 5.18 (2D Factor PDA for one pixel picture)



Example 5.19 (2D Factor PDA for (2×2) picture)



Example 5.20 (2D Factor PDA for (3×3) picture)



5.5 Generic algorithm for indexing of multidimensional picture in the prefix notation

The idea of dimensional reduction is used to provide a generic algorithm of the n-dimensional pattern matching using pushdown automata. We use it to obtain the generic algorithm of the two-dimensional pattern matching using finite automata in Sec. 4.

In this chapter by the dimensional reduction of the multidimensional picture, we obtain a directed acyclic graph (Def. 2.76) or a tree. The DAG is constructed in such a way that it or, more precisely, its linear textual representation may serve as an input for a run of a pushdown automaton or a tree automaton.

This differ from the approach described in Sec. 4 where the dimensional reduction processes the n-dimensional structure (text) repetitively, until a one-dimensional text is obtained. In this chapter the preprocessing of the n-dimensional structure is performed only once.

Advantages of such an approach to n-dimensional pattern matching:

- standard pushdown automata can be used,
- a text is processed in time proportional to the length of the linear representation of the text.

Open question is whether the pushdown automata can be transformed so that they can belong to the class of visibly pushdown automata and thus could be transformed into deterministic pushdown automata.

Algorithm 5.9: A generic algorithm of the multidimensional pattern matching using pushdown automata
Input: Let pattern array PA and text array TA be n-dimensional pictures extended by the # symbol. Let $|PA| \leq |TA|$.
Output: Reports all occurrences or first occurrence only, etc. It depends on further specification of searching.
Method:

1: Construct a suitable representation from a n-D text and n-D pattern in the form of a tree, if needed.
2: Create a pushdown automaton as an index of text TA over the tree representation.
3: Construct a linearised form of the pattern from the tree. (E.g. Alg. 3.6, Alg. 3.7, Alg. 3.8).
4: Run the pushdown automaton over linearised pattern to determine possible (multidimensional) occurrences of PA in TA.

5.6 Concluding remarks

In this chapter, we presented a processing of pictures using pushdown automata.

We noted the 2D prefix indexing pushdown automaton is deterministic. It is interesting the same property holds also in 1D case, where the prefix automaton is very efficient. The same property holds for tree-top matching.

One of the open questions is whether there exists a deterministic version of the two-dimensional suffix pushdown automaton. The problem can be avoided using the definition of 2D prefix and 2D suffix. It allows us to use the 2D prefix pushdown automaton to the transponed picture and search for 2D suffixes this way.

6 Conclusion

MAIN contributions of this work are based on the application of finite automata to two-dimensional exact and approximate pattern matching and on the application of pushdown automata to two-dimensional exact pattern matching and to two-dimensional text indexing.

Finite automata
First, a general pattern matching automata based approach of two-dimensional pattern matching problems is presented. Finite automata are used in our approach, because they are well known and at present time the pattern matching automata classification serves as a model of all one-dimensional pattern matching problems. Their research brought up several favourable properties, including elaborate simulation methods of run of finite automaton, allowing direct usage of finite automata constructed for the particular problem. Our aim was to reuse these results for the two-dimensional pattern matching.

Based on the generic algorithm, two particular methods have been presented, one for the 2D exact and one for the 2D approximate pattern matching using the 2D Hamming distance. Beside automata based models we dealt with their implementation issues. In the former case, there exists very similar approach of Baker and Bird, using pattern matching machines (the forward transition function is combined with the *fail*-function), working in linear time. In general it is impossible to use a simulation of nondeterministic models and obtain the linear time complexity. In such case our new method would have been inferior to the method of Baker and Bird and not suitable for applications. Fortunately, there exist direct construction methods of the equivalent deterministic pattern matching automata and using them our method is able to work in the linear time, too. We presented a new way how to significantly reduce the space needed to the size of only one row of the text array. More precisely, the optimised method of the two-dimensional exact pattern matching using deterministic finite automata works with $\mathcal{O}(N_{TA})$ asymptotic time and $\mathcal{O}(N_{PA} + n)$ asymptotic space complexity, $|PA| = N_{PA}$, $|TA| = N_{TA}$, n is the length of one side of TA (e.g. the number of its columns).

Then we presented the model of the two-dimensional approximate pattern matching using the 2D Hamming distance. The matching can be done in $\mathcal{O}(N_{TA})$, if the required determinisation of the preprocessing automaton is not considered. Compared to the previous case, there is no known direct construction of the deterministic automata for approximate pattern matching needed in our algorithm. It is possible to simulate the pattern matching automata and obtain asymptotic time complexity $\mathcal{O}(N_{TA}N_{PA})$ for simulation with dynamic programming. This is not optimal, however, it does not depend on the number of errors. Also for this method, we presented a way how to make its otherwise large space requirements modest. Our optimised method of the two-dimensional approximate pattern matching has $\mathcal{O}(nN_{PA})$ asymptotic space complexity.

Pushdown automata
Second, a method of two-dimensional exact pattern matching using pushdown automata has been presented. We have shown how a picture can be decomposed into a tree, where the context of each element of the picture is preserved. The tree can be linearised into its prefix notation and a pattern matching can be performed over this representation.

For two-dimensional pattern matching over the trees in prefix notation, different methods can be used. From different tree pattern matching algorithms, we have mentioned the applica-

tion of pushdown automata for tree pattern matching. It is because the pushdown automata correspond directly with the area of interest of this thesis. Moreover, in principle, they can do the matching faster than other tree pattern matching methods known.

Based on the same representation, we have proposed new method of picture indexing. Our pushdown automata constructed for this purpose allow to match for a tree representation of a two-dimensional pattern in the prefix notation. Namely, we can search for a two-dimensional prefix, suffix, or factor of the two-dimensional text.

All these pushdown automata are indexing the two-dimensional text and allow to search for the two-dimensional pattern in time proportional to the size of the pattern itself, independently of the size of the text. Two-dimensional factor and suffix automata are nondeterministic. However, two-dimensional prefix automaton is deterministic and optimal in size, since its size is proportional to the size of two-dimensional text. The lower bound for storing nodes of the suffix tree for two-dimensional pictures is $\Omega(nN_{TA})$, the best construction algorithm runs in $\mathcal{O}(N_{TA} \log n)$ for square pictures. Our algorithm for construction of two-dimensional prefix automaton runs in $\mathcal{O}(N_{TA})$ time and the automaton has $\mathcal{O}(N_{TA})$ states.

Open problems

There are many open problems, mainly in the area of two-dimensional text indexing. Let us recapitulate at least some of them:

- Is it possible to find finite automata model for two-dimensional distances not preserving the shape of the picture?
- Does there exist a deterministic version of two-dimensional factor pushdown automaton?
- Is it possible to transform two-dimensional factor and suffix pushdown automata to e.g. visibly pushdown automata in order to make them deterministic?
- If so, does the determinisation of the two-dimensional factor pushdown automaton of $|Q|$ states always result in $\mathcal{O}\left(2^{|Q|^2}\right)$ states of its deterministic version?

7 Bibliography

[ABF92] Amihood Amir, Gary Benson, and Martin Farach: Alphabet independent two dimensional matching. In *Proceedings of the 24th ACM Symposium on the Theory of Computing*, pp. 59–68, Victoria, Canada, 1992. ACM Press.

[AC75] Alfred Vaino Aho and Margaret John Corasick: Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[AF92] Amihood Amir and Martin Farach: Two-dimensional dictionary matching. *Inf. Process. Lett.*, 44(5):233–239, 1992.

[AL91] Amihood Amir and Gad M. Landau: Fast parallel and serial multidimensional approximate array matching. *Theor. Comput. Sci.*, 81(1):97–115, 1991.

[AM04] Rajeev Alur and Parthasarathy Madhusudan: Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM Symposium on the Theory of Computing*, pp. 202–211, New York, NY, 2004. ACM Press.

[Ami92] Amihood Amir: Two-dimensional periodicity and its applications. Technical Report GIT-CC-92/29, College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280, July 1992.

[Ami04] Amihood Amir: Theoretical issues of searching aerial photographs: A bird's eye view. In Miroslav Balík, Jan Holub, and Milan Šimánek, editors, *Proceedings of the Prague Stringology Conference 2004*, pp. 1–23, Czech Technical University in Prague, Czech Republic, September 2004.

[AU71] Alfred Vaino Aho and Jeffrey D. Ullman: *The theory of parsing, translation and compiling*, Vol. I Parsing. Prentice Hall, Englewood Cliffs, New York, 1971.

[Bak78] Theodore P. Baker: A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comput.*, 7(4):533–541, November 1978.

[Bal03] Miroslav Balík: DAWG versus suffix array. In Jean-Marc Champarnaud and Denis Maurel, editors, *Proceedings of the 7th Conference on Implementation and Application of Automata*, No. 2608 in Lecture Notes in Computer Science, pp. 229–244. Springer-Verlag, Berlin/Heidelberg, 2003. Revised papers.

[BB01] Jerome Barthelemy and Alain Bonardi: Similarity in computational music: a musicologist's approach. In *Proceedings of the 1st International Conference on WEB Delivering of Music*, pp. 107–115, Washington, DC, USA, 2001. IEEE Computer Society Press.

[BBE+83] Anselm Blumer, Janet Blumer, Andrzej Ehrenfeucht, David Haussler, and Ross M. McConnel: Linear size finite automata for the set of all subwords of a word: an outline of results. *Bull. Eur. Assoc. Theor. Comput. Sci.*, 21:12–20, 1983.

[BBE+85] Anselm Blumer, Janet Blumer, Andrzej Ehrenfeucht, David Haussler, M. T. Chen, and Joel Seiferas: The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40(1):31–55, 1985.

[Bir77] Richard S. Bird: Two-dimensional pattern matching. *Inf. Process. Lett.*, 6(5):168–170, October 1977.

[BM77] Robert Stephen Boyer and J. Strother Moore: A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[BY98] Ricardo A. Baeza-Yates: Similarity in two-dimensional strings. In Wen-Lian Hsu and Ming-Yang Kao, editors, *Proceedings of the 4th Annual International Conference on Computing and Combinatorics*, No. 1449 in Lecture Notes in Computer Science, pp. 319–328. Springer-Verlag, Berlin, August 1998.

[BYN99] Ricardo A. Baeza-Yates and Gonzalo Navarro: Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[BYN00] Ricardo A. Baeza-Yates and Gonzalo Navarro: New models and algorithms for multidimensional approximate pattern matching. *J. Discret. Algorithms*, 1(1):21–49, 2000.

[BYR93] Ricardo A. Baeza-Yates and Mireille Régnier: Fast two-dimensional pattern matching. *Inf. Process. Lett.*, 45(1):51–57, 1993.

[CCI+99] Emilios Cambouropoulos, Maxime Crochemore, Costas S. Iliopoulos, Laurent Mouchard, and Yoan J. Pinzon: Algorithms for computing approximate repetitions in musical sequences. In R. Raman and J. Simpson, editors, *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms*, pp. 129–144, Perth, WA, Australia, 1999.

[CDG+07] Hubert Comon, Max Dauchet, Rémy Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi: Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. Release October, 12th 2007.

[CH97] Maxime Crochemore and Christophe Hancart: Automata for matching patterns. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, pp. 399–462. Springer-Verlag, Berlin, 1997.

[Cle08] Loek Cleophas: *Tree Algorithms. Two Taxonomies and a Toolkit.* PhD thesis, Technische Universiteit Eindhoven, Eindhoven, 2008.

[CLRV08] Gabriel Cardona, Mercè Llabrés, Francesc Rosselló, and Gabriel Valiente: A distance metric for a class of tree-sibling phylogenetic networks. *Bioinformatics*, 24(13):1481–1488, 2008.

[CR94] Maxime Crochemore and Wojciech Rytter: *Text algorithms.* Oxford University Press, 1994.

[CR02] Maxime Crochemore and Wojciech Rytter: *Jewels of Stringology.* World Scientific Publishing, Hong-Kong, 2002. 310 pages.

[Cro85] Maxime Crochemore: Optimal factor transducers. In Alberto Apostolico and Zvi Galil, editors, *Combinatorial Algorithms on Words*, Vol. 12 of *NATO Advanced Science Institutes, Series F*, pp. 31–44. Springer-Verlag, Berlin, 1985.

[Cro86] Maxime Crochemore: Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.

[CRV08] Gabriel Cardona, Francesc Rosselló, and Gabriel Valiente: Extended Newick: it is time for a standard representation of phylogenetic networks. *BMC Bioinformatics*, 9(532), 2008.

[CRV09] Gabriel Cardona, Francesc Rosselló, and Gabriel Valiente: Comparison of tree-child phylogenetic networks. *IEEE/ACM Trans. on Computational Biology and Bioinformatics*, 6(4):552–569, October 2009.

[Dam64] Fred J. Damerau: A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, 1964.

[Doo99] W. Ford Doolittle: Phylogenetic classification and the universal tree. *Science*, 284(5423):2124–2128, 1999.

[Far97] Martin Farach: Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th IEEE Annual Symposium on Foundations of Computer Science*, pp. 137–143, Miami Beach, FL, 1997.

[Fel04] Joseph Felsenstein: *Inferring Phylogenies*. Sinauer Associates, Sunderland, Massachusetts, 2004. 580 pages.

[FJM09] Tomáš Flouri, Jan Janoušek, and Bořivoj Melichar: Tree pattern matching by deterministic pushdown automata. In Maria Ganzha and Marcin Paprzycki, editors, *Proceedings of the International Multiconference on Computer Science and Information Technology, Workshop on Advances in Programming Languages*, Vol. 4, pp. 659–666. IEEE Computer Society Press, 2009.

[FM00] Paolo Ferragina and Giovanni Manzini: Opportunistic data structures with applications. In *Proceedings of the 41st IEEE Annual Symposium on Foundations of Computer Science*, p. 390, Washington, DC, USA, 2000. IEEE Computer Society Press.

[FP74] Michael J. Fischer and Mike S. Paterson: String matching and other products. In Richard M. Karp, editor, *Complexity of Computation*, Vol. 7, pp. 113–125. SIAM-AMS Proceedings, 1974.

[Gal85] Zvi Galil: Open problems in stringology. In Alberto Apostolico and Zvi Galil, editors, *Combinatorial Algorithms on Words*, Vol. 12 of *NATO ASI Series F*, pp. 1–8. Springer-Verlag, Berlin, 1985.

[GBYS92] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider: New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures And Algorithms*, chapter 5, pp. 66–82. Prentice Hall, 1992.

[GG96] Raffaele Giancarlo and Roberto Grossi: On the construction of classes of suffix trees for square matrices: Algorithms and applications. *Inf. Comput.*, 130(2):151–182, 1996.

[GG97a] Raffaele Giancarlo and Roberto Grossi: Multi-dimensional pattern matching with dimensional wildcards: Data structures and optimal on-line search algorithms. *J. Algorithms*, 24(2):223–265, 1997.

[GG97b] Raffaele Giancarlo and Roberto Grossi: Suffix tree data structures for matrices. In Alberto Apostolico and Zvi Galil, editors, *Pattern matching algorithms*, chapter 10, pp. 293–340. Oxford University Press, 1997.

[Gia93] Raffaele Giancarlo: An index data structure for matrices, with applications to fast two-dimensional pattern matching. In F. Dehne, J.-R. Sack, N. Santoro, and S. Whitesides, editors, *Proceedings of the 3rd Workshop on Algorithms and Data Structures*, No. 709 in Lecture Notes in Computer Science, pp. 337–348, Montréal, Canada, 1993. Springer-Verlag, Berlin.

[Gia95] Raffaele Giancarlo: A generalization of the suffix tree to square matrices, with applications. *SIAM J. Comput.*, 24(3):520–562, 1995.

[Gon88] Gaston H. Gonnet: Efficient searching of text and pictures. Report OED-88-02, University of Waterloo, 1988.

[GP92] Zvi Galil and Kunsoo Park: Truly alphabet-independent two-dimensional pattern matching. In *Proceedings of the 33th IEEE Annual Symposium on Foundations of Computer Science*, pp. 247–256, Pittsburgh, PA, 1992. IEEE Computer Society Press.

[GR97] Dora Giammarresi and Antonio Restivo: Two-dimensional languages. In *Handbook of Formal Languages*, Vol. III (Beyond Words), pp. 216–267. Springer-Verlag, Berlin/Heidelberg, 1997.

[Gus97] Dan Gusfield: *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, Cambridge, 1997.

[GV00] Roberto Grossi and Jeffrey Scott Vitter: Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the 32nd ACM Symposium on the Theory of Computing*, pp. 397–406, New York, NY, 2000. ACM Press.

[GV05] Roberto Grossi and Jeffrey Scott Vitter: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[Ham50] Richard Wesley Hamming: Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950.

[Har70] Malcolm C. Harrison: Set comparison using hashing techniques, 1970. 23 pages, available at: `http://www.archive.org/details/setcomparisonusi00harr`.

[Har71] Malcolm C. Harrison: Implementation of the substring test by hashing. *Commun. ACM*, 14(12):777–779, December 1971.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman: *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 2001. Second edition, 535 pages.

[Hol00] Jan Holub: *Simulation of nondeterministic finite automata in pattern matching.* Dissertation thesis, Czech Technical University in Prague, Czech Republic, 2000.

[IN77] Katsushi Inoue and Akira Nakamura: Some properties of two-dimensional on-line tessellation acceptors. *Inf. Sci.*, 13(2):95–121, 1977.

[Jan09] Jan Janoušek: String suffix automata and subtree pushdown automata. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2009*, pp. 160–172, Czech Technical University in Prague, Czech Republic, 2009.

[JM09] Jan Janoušek and Bořivoj Melichar: On regular tree languages and deterministic pushdown automata. *Acta Inform.*, 46(7):533–547, November 2009.

[JM10] Jan Janoušek and Bořivoj Melichar: Subtree pushdown automata and tree pattern pushdown automata for trees in prefix notation. *(Submitted)*, 2010. Submitted to Acta Inform.

[KKP98] Dong Kyue Kim, Yoo Ah Kim, and Kunsoo Park: Constructing suffix arrays for multi-dimensional matrices. In Martin Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, No. 1448 in Lecture Notes in Computer Science, pp. 126–139, Piscataway, NJ, 1998. Springer-Verlag, Berlin/Heidelberg.

[KKP03] Dong Kyue Kim, Yoo Ah Kim, and Kunsoo Park: Generalizations of suffix arrays to multi-dimensional matrices. *Theor. Comput. Sci.*, 302(1–3):401–416, 2003.

[KMP77] Donald E. Knuth, James H. Morris, Jr, and Vaughan R. Pratt: Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[KR87] Richard M. Karp and Michael O. Rabin: Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

[KS87] Kamala Krithivasan and R. Sitalakshmi: Efficient two-dimensional pattern matching in the presence of errors. *Inf. Sci.*, 43(3):169–184, 1987.

[KU94] Juha Kärkkäinen and Esko Ukkonen: Two and higher dimensional pattern matching in optimal expected time. In *Proceedings of the 5th ACM-SIAM Annual Symposium on Discrete Algorithms*, pp. 715–723, Arlington, VA, 1994.

[Lev66] Vladimir I. Levenshtein: Binary codes capable of correcting deletions, insertions and reversals. *Sov. Phys. Dokl.*, 10(8):707–710, 1966.

[LSSY02] Tak-Wah Lam, Kunihiko Sadakane, Wing-Kin Sung, and Siu-Ming Yiu: A space and time efficient algorithm for constructing compressed suffix arrays. In *Proceedings of the 8th Annual International Conference on Computing and Combinatorics*, Vol. 2387 of *Lecture Notes in Computer Science*, pp. 401–410. Springer-Verlag, Berlin, 2002.

[McC76] Edward M. McCreight: A space-economical suffix tree construction algorithm. *J. Algorithms*, 23(2):262–272, 1976.

[Mel95] Bořivoj Melichar: Approximate string matching by finite automata. In Václav Hlaváč and Radim Šára, editors, *Computer Analysis of Images and Patterns*, No. 970 in Lecture Notes in Computer Science, pp. 342–349. Springer-Verlag, Berlin, 1995.

[MH97] Bořivoj Melichar and Jan Holub: 6D classification of pattern matching problems. In Jan Holub, editor, *Proceedings of the Prague Stringology Club Workshop '97*, pp. 24–32, Czech Technical University in Prague, Czech Republic, 1997. Collaborative Report DC–97–03.

[MHP05] Bořivoj Melichar, Jan Holub, and Tomáš Polcar: Text searching algorithms. Available on: `http://www.stringology.org/athens/`, November 2005.

[MM93] Udi Manber and Gene Myers: Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[Mor68] Donald R. Morrison: PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *J. Assoc. Comput. Mach.*, 15(4):514–534, October 1968.

[Mor96] Virginia Morell: TreeBASE: The roots of phylogeny. *Science*, 273(5275):569, 1996. `http://www.treebase.org/`.

[Mye86] Eugene W. Myers: An $\mathcal{O}(ND)$ difference algorithm and its variations. *Algorithmica*, 1(1):251–266, March 1986.

[NGP07] Joong Chae Na, Raffaele Giancarlo, and Kunsoo Park: On-line construction of two-dimensional suffix tree in $\mathcal{O}(n^2 \log n)$ time. *Algorithmica*, 48:173–186, 2007.

[Ols90] Gary Olsen: "Newick's 8:45" tree format standard. `http://evolution.genetics.washington.edu/phylip/newick_doc.html`, August 1990.

[Pag99] Mark Pagel: Inferring the historical patterns of biological evolution. *Nature*, 401:877–884, October 1999.

[Pol04] Tomáš Polcar: Two-dimensional pattern matching. Postgraduate study report DC-PSR-04-05, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic, 2004.

[Prů04] Daniel Průša: *Two-dimensional Languages*. PhD thesis, Faculty of Mathematics and Physics, Charles University, Prague, 2004.

[RH91] Sanjay Ranka and Todd Heywood: Two-dimensional pattern matching with k mismatches. *Pattern Recognit.*, 24(1):31–40, 1991.

[RMJ06] Marta Rukoz, Maude Manouvrier, and Geneviève Jomier: δ-distance: A family of dissimilarity metrics between images represented by multi-level feature vectors. *Information Retrieval*, 9(6):633–655, December 2006.

[Sad00] Kunihiko Sadakane: Compressed text databases with efficient query algorithms based on the compressed suffix array. In Der-Tsai Lee and Shang-Hua Teng, editors, *Proceedings of the 11th International Symposium on Algorithms and Computation*, No. 1969 in Lecture Notes in Computer Science, pp. 410–421. Springer-Verlag, Berlin/Heidelberg, 2000.

[SBB+02] Jason E. Stajich, David Block, Kris Boulez, Steven E. Brenner, Stephen A. Chervitz, Chris Dagdigian, Georg Fuellen, James G. R. Gilbert, Ian Korf, Hilmar Lapp, Heikki Lehväslaiho, Chad Matsalla, Chris J. Mungall, Brian I. Osborne, Matthew R. Pocock, Peter Schattner, Martin Senger, Lincoln D. Stein, Elia Stupka, Mark D. Wilkinson, and Ewan Birney: The Bioperl toolkit: Perl modules for the life sciences. *Genome Res.*, 12(10):1611–1618, October 2002.

[Sel80] Peter H. Sellers: The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, 1(4):359–373, 1980.

[Smy03] William F. Smyth: *Computing Patterns in Strings*. Addison Wesley Pearson, 2003.

[Ukk85a] Esko Ukkonen: Algorithms for approximate string matching. *Inf. Control*, 64(1–3):100–118, 1985.

[Ukk85b] Esko Ukkonen: Finding approximate patterns in strings. *J. Algorithms*, 6(1–3):132–137, 1985.

[Ukk95] Esko Ukkonen: On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[Val02] Gabriel Valiente: *Algorithms on Trees and Graphs*. Springer-Verlag, Berlin, 2002. 504 pages.

[Wei73] Peter Weiner: Linear pattern matching algorithms. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pp. 1–11, Washington, DC, USA, 1973. IEEE Computer Society Press.

[WM92] Sun Wu and Udi Manber: Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.

[WW01] Klaus Wagner and Gerd Wechsung: *Computational Complexity.* Springer-Verlag, Berlin, 2001. 556 pages.

[ZT89] Rui Feng Zhu and Tadao Takaoka: A technique for two-dimensional pattern matching. *Commun. ACM*, 32(9):1110–1120, 1989.

8 Relevant Refereed Publications of the Author

[ŽM04] Jan Žďárek and Bořivoj Melichar: Finite automata and two-dimensional pattern matching. In Marjan Heričko, Ivan Rozman, Matjaž B. Jurič, Vladislav Rajkovič, Tanja Urbančič, Mocja Bernik, Maja Bučar, and Andrej Brodnik, editors, *Proceedings of the 7th International Multiconference Information Society IS'2004, Theoretical Computer Science section*, Vol. D, pp. 185–188, Ljubljana, Slovenia, 2004. Institut "Jožef Stefan".

[ŽM06] Jan Žďárek and Bořivoj Melichar: On two-dimensional pattern matching by finite automata. In Jacques Farré, Igor Litovsky, and Sylvain Schmitz, editors, *Proceedings of the 10th Conference on Implementation and Application of Automata*, No. 3845 in Lecture Notes in Computer Science, pp. 329–340. Springer-Verlag, Berlin/Heidelberg, 2006. Revised selected papers.

Index