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

TEXT SEARCHING ALGORITHMS

VOLUME I: FORWARD STRING MATCHING

Bořivoj Melichar, Jan Holub, Tomáš Polcar

November 2005

Preface

Text is the simplest and most natural representation of information in a range of areas. Text is a linear sequence of symbols from some alphabet. The text is manipulated in many application areas: processing of text in natural and formal languages, study of sequences in molecular biology, music analysis, etc.

The design of algorithms that process texts goes back at least thirty years. In particular, the 1990_s produced many new results. This progress is due in part to genome research, where text algorithms are often used.

The basic problem of text processing concerns string matching. It is used to access information, and this operation is used very frequently.

We have recognized while working in this area that finite automata are very useful tools for understanding and solving many text processing problems. We have found in some cases that well known algorithms are in fact simulators of non-deterministic finite automata serving as models of these algorithms. Far this reason the material used in this course is based mainly on results from the theory of finite automata.

Because the string is a central notion in this area, Stringology has become the nickname of this subfield of algorithmic research.

We suppose that you, the reader of this tutorial, have basic knowledge in the following areas:

Finite and infinite sets, operations with sets.
Relations, operations with relations.
Basic notions from the theory of oriented graphs.
Regular languages, regular expressions, finite automata, operations with finite automata.

The material included in this tutorial corresponds to our point of view on the respective aspects of Stringology. Some parts of the tutorial are the results of our research and some of the principles described here have not been published before.

Prague, November 2004 Authors

Contents

1 Text retrieval systems **7**

1.1 Basic notions and notations 8
1.2 Classification of pattern matching problems 12
1.3 Two ways of pattern matching 14
1.4 Finite automata 15
1.5 Regular expressions 20
1.5.1 Definition of regular expressions 20
1.5.2 The relation between regular expressions and finite automata 21

2 Forward pattern matching **23**

2.1 Elementary algorithm 24
2.2 Pattern matching automata 25
2.2.1 Exact string and sequence matching 25
2.2.2 Substring and subsequence matching 27
2.2.3 Approximate string matching - general alphabet . . . 30
2.2.3.1 Hamming distance 30
2.2.3.2 Levenshtein distance 32
2.2.3.3 Generalized Levenshtein distance 33
2.2.4 Approximate string matching - ordered alphabet . . . 34
2.2.4.1 Δ–distance 35
2.2.4.2 Γ-distance 36
2.2.4.3 (Δ, Γ)-distance 38
2.2.5 Approximate sequence matching 39
2.2.6 Matching of finite and infinite sets of patterns 41
2.2.7 Pattern matching with "don't care" symbols 45
2.2.8 Matching a sequence of patterns 48
2.3 Some deterministic pattern matching automata 48
2.3.1 String matching 49
2.3.2 Matching of a finite set of patterns 50
2.3.3 Regular expression matching 52
2.3.4 Approximate string matching – Hamming distance . . 56
2.3.5 Approximate string matching – Levenshtein distance . 57
2.4 The state complexity of the deterministic pattern matching automata 57
2.4.1 Construction of a dictionary matching automaton . . 60
2.4.2 Approximate string matching 64
2.4.2.1 Hamming distance 64
2.4.2.2 Levenshtein distance 66
2.4.2.3 Generalized Levenshtein distance 68
2.4.2.4 Γ distance 70
2.5 (Δ, Γ) distance 73

2.6 Δ distance 75

3 Finite automata accepting parts of a string 77
3.1 Prefix automaton 77
3.2 Suffix automaton 80
3.3 Factor automaton 87
3.4 Parts of suffix and factor automata 92
3.4.1 Backbone of suffix and factor automata 93
3.4.2 Front end of suffix or factor automata 93
3.4.3 Multiple front end of suffix and factor automata . . . 96
3.5 Subsequence automata 97
3.6 Factor oracle automata 99
3.7 The complexity of automata for parts of strings 108
3.8 Automata for parts of more than one string 109
3.9 Automata accepting approximate parts of a string 115

4 Borders, repetitions and periods 120
4.1 Basic notions 120
4.2 Borders and periods 121
4.2.1 Computation of borders 123
4.2.2 Computation of periods 124
4.3 Border arrays 125
4.4 Repetitions 129
4.4.1 Classification of repetitions 129
4.4.2 Exact repetitions in one string 131
4.4.3 Complexity of computation of exact repetitions 141
4.4.4 Exact repetitions in a finite set of strings 146
4.4.5 Computation of approximate repetitions 151
4.4.6 Approximate repetitions – Hamming distance 151
4.4.7 Approximate repetitions – Levenshtein distance 155
4.4.8 Approximate repetitions – Δ distance 158
4.4.9 Approximate repetitions – Γ distance 161
4.4.10 Approximate repetitions – (Δ, Γ) distance 164
4.4.11 Exact repetitions in one string with don't care symbols 168
4.5 Computation of periods revisited 172

5 Simulation of nondeterministic pattern matching automata – fail function 177
5.1 Searching automata 177
5.2 *MP* and *KMP* algorithms 178
5.3 *AC* algorithm 187

6 Simulation of nondeterministic finite automata – dynamic programming and bit parallelism 195
6.1 Basic simulation method 195
6.1.1 Implementation 196
6.1.1.1 *NFA* without ε-transitions 196
6.1.1.2 *NFA* with ε-transitions 199
6.2 Dynamic programming 202
6.2.1 Algorithm 202
6.2.2 String matching 202
6.2.2.1 Exact string matching 202
6.2.2.2 Approximate string matching using Hamming distance 202
6.2.2.3 Approximate string matching using Levenshtein distance 205
6.2.2.4 Approximate string matching using generalized Levenshtein distance 208
6.2.3 Time and space complexity 210
6.3 Bit parallelism 211
6.3.1 Algorithm 211
6.3.2 String matching 212
6.3.2.1 Exact string matching 212
6.3.2.2 Approximate string matching using Hamming distance 213
6.3.2.3 Approximate string matching using Levenshtein distance 215
6.3.2.4 Approximate string matching using generalized Levenshtein distance 218
6.3.3 Other methods of bit parallelism 221
6.3.4 Time and space complexity 222

References 223

1 Text retrieval systems

Text retrieval systems deal with the representation, storage, organization of, and access to text documents. The organization of text documents should provide the user with easy access to documents of interest. The database of a retrieval system contains a collection of documents. The user can ask for some documents. He must formulate his needs in the form of a query. The query is processed by the search engine and the result is a set of selected documents. This process is depicted in Fig. 1.1. The query is an expression

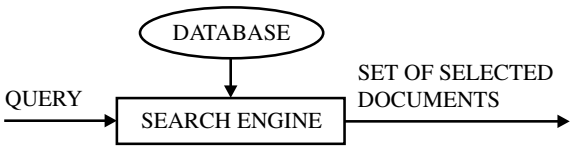


Figure 1.1: Retrieval system

containing keywords as basic elements. The simplest operation of the search engine is the selection of documents containing some keywords of a given query. In this text, we will concentrate on the algorithms used by search engines. The simplest task can be formulated in this way:

Given text string $T = t_1 t_2 \dots t_n$ and pattern (keyword)
$P = p_1 p_2 \dots p_m$, verify if string P is a substring of text T, where t_i and p_i are symbols of the alphabet.

This task is very simple but it is used very frequently. Very fast algorithms are therefore necessary for this task. The design of algorithms that process texts goes back at least to the 1970s. thirty years. In particular, the last decade has produced many new results. We have recognized that finite automata are very useful tools for understanding and solving many text processing problems. We have found in some cases that well known algorithms are in fact simulators of nondeterministic finite automata serving as models of these algorithms.

Because the string is the central notion in this area, *stringology* has become the nickname of this subfield of algorithmic research. To achieve fast text searching, we can prepare either the pattern or the text or both. This preparation is called preprocessing. We can use preprocessing as the criterion for a general classification of text searching approaches. There are four categories in this classification:

1. Neither the pattern nor the text is preprocessed. Elementary algorithms belong in this category.
2. The pattern is preprocessed. Pattern matching automata belong in this category.
3. The text is preprocessed. Factor automata and index methods belong in this category.

4. Both the text and the pattern are preprocessed. Signature methods, pattern matching automata and factor automata belong in this category.

This classification is represented in Fig. 1.2.

		Text preprocessing	
		NO	YES
Pattern preprocessing	NO	Elementary algorithms	Factor automata, index methods
	YES	Pattern matching automata	Pattern matching automata, factor automata, signature methods

Figure 1.2: Classification of text searching approaches

1.1 Basic notions and notations

Some basic notions will be used in the following chapters. This section collects definitions of them.

An *alphabet* is a nonempty finite set of symbols. The alphabet can be either ordered or unordered. The ordering is supposed to be the total. Most operations can be used for either ordered or unordered alphabets (general alphabets). Some specific operations can be used only for totally ordered alphabets.

A *string* over a given alphabet is a finite sequence of symbols. Empty string ε is empty sequence of symbols. We denote by A^* the set of all strings over alphabet A (including empty string ε). This set is always infinite. A set of nonempty strings over alphabet A is denoted by A^+. It holds that $A^* = A^+ \cup \{\varepsilon\}$. The complement of alphabet A for some set of symbols $B, B \subset A$ is denoted $\overline{B} = A \setminus B$. Notation $\overline{a}$ means $A \setminus \{a\}$. The operation concatenation is defined on the set of strings in this way: if x and y are strings over A, then the concatenation of these strings is xy. This operation is associative, i.e. $(xy)z = x(yz)$. On the other hand, it is not commutative, i.e. $xy \neq yx$. Empty string ε is the neutral element: $x\varepsilon = \varepsilon x = x$. The set of strings A^* over alphabet A is a free monoid with ε as the neutral element. The length of string $|x|$ is the number of symbols of x. It holds that $|x| \geq 0, |\varepsilon| = 0$. We will use integer exponents for a string with repetitions: $a^0 = \varepsilon, a^1 = a, a^2 = aa, a^3 = aaa, \ldots$, for $a \in A$ and $x^0 = \varepsilon, x^1 = x, x^2 = xx, x^3 = xxx, \ldots$, for $x \in A^*$.

Definition 1.1
Set *Pref*$(x), x \in A^*$, is the set of all *prefixes* of string x:
Pref$(x) = \{y : x = yu, u, x, y \in A^*\}$. □

Definition 1.2
Set $Suff(x), x \in A^*$, is the set of all *suffixes* of string x:
$Suff(x) = \{y : x = uy, u, x, y \in A^*\}$. □

Definition 1.3
Set $Fact(x), x \in A^*$, is the set of all *substrings (factors)* of string x:
$Fact(x) = \{y : x = uyv, u, v, x, y \in A^*\}$. □

Definition 1.4
Set $Sub(x), x \in A^*$, is the set of all *subsequences* of string x:
$$Sub(x) = \{\, a_1a_2 \dots a_m : x = y_0a_1y_1a_2 \dots a_my_m, \\ y_i \in A^*, i = 0, 1, 2, \dots, m, a_j \in A, j = 1, 2, \dots, m, m \geq 0\}.$$ □

Definition 1.5
The terms proper prefix, proper suffix, proper factor, proper subsequence are used for a prefix, suffix, factor, subsequence of string x which is not equal to x. □

The definitions of sets *Pref, Suff, Fact* and *Sub* can be extended for finite and infinite sets of strings.

Definition 1.6
Set $Pref(X), X \subset A^*$, is the *set of all prefixes* of all strings $x \in X$:
$Pref(X) = \{y : x = yu, x \in X, u, x, y \in A^*\}$. □

Definition 1.7
Set $Suff(X), X \subset A^*$, is the *set of all suffixes* of all strings $x \in X$:
$Suff(X) = \{y : x = yu, x \in X, w, x, y \in A^*\}$. □

Definition 1.8
Set $Fact(X), X \subset A^*$, is the *set of all substrings* (factors) of all strings $x \in X$:
$Fact(X) = \{y : x = yu, x \in X, u, x, y \in A^*\}$. □

Definition 1.9
Set $Sub(X), X \subset A^*$, is the *set of all subsequences* of all strings $x \in X$:
$$Sub(X) = \{\, a_1a_2 \dots a_m : x = y_0a_1y_1a_2 \dots a_my_m, x \in X, y_i \in A^*, \\ i = 0, 1, 2, \dots, m, a_j \in A,\ j = 1, 2, \dots, m,\ m \geq 0\}.$$ □

The definition of the abovementioned sets can also be extended for approximate cases. In the following definitions D is a metrics, k is the distance.

Definition 1.10
The set of approximate prefixes *APref* of string x is:
$APref(x) = \{u : v \in Pref(x),\ D(u, v) \leq k\}$. □

Definition 1.11
The set of approximate suffixes *ASuff* of string x is:
$ASuff(x) = \{u : v \in Suff(x),\ D(u, v) \leq k\}$. □

Definition 1.12
The set of approximate factors $AFact$ of string x is:
$AFact(x) = \{u : v \in Fact(x),\ D(u, v) \leq k\}$. □

Definition 1.13
The set of approximate subsequences $ASub$ of string x is:
$ASub(x) = \{u : v \in Sub(x),\ D(u, v) \leq k\}$. □

The term pattern matching is used for both string matching and sequence matching. The term subpattern matching is used for matching substrings or subsequences of a pattern.

Definition 1.14
The "don't care" symbol is a special universal symbol $\circ$ that matches any other symbol, including itself.

Definition 1.15 (Basic pattern matching problems)
Given text $T = t_1 t_2 \dots t_n$ and pattern $P = p_1 p_2 \dots p_m$, we may define:

1. String matching: verify whether string P is a substring of text T.
2. Sequence matching: verify whether sequence P is a subsequence of text T.
3. Subpattern matching: verify whether a subpattern of P (substring or subsequence) occurs in text T.
4. Approximate pattern matching: verify whether string x occurs in text T so that distance $D(P, x) \leq k$ for given $k < m$.
5. Pattern matching with "don't care" symbols: verify if pattern P containing "don't care" symbols occurs in text T. □

Definition 1.16 (Matching a sequence of patterns)
Given text $T = t_1 t_2 \dots t_n$ and a sequence of patterns (strings and/or sequences) $P_1, P_2, \dots, P_s$. Matching of a sequence of patterns $P_1, P_2, \dots, P_s$ is a verification whether an occurrence of pattern P_i in text T is followed by an occurrence of P_{i+1}, $1 \leq i < s$. □

Definitions 1.15 and 1.16 define pattern matching problems as decision problems, because the output is a Boolean value. A modified version of these problems consists in searching for the first, the last, or all occurrences of a pattern and moreover the result may be the set of positions of the pattern in the text.

Instead of just one pattern, one can consider a finite or infinite set of patterns.

Definition 1.17 (Distances of strings - general alphabets)
Three variants of distances between two strings x and y are defined as the minimum number of editing operations:

1. *replace* (Hamming distance, R-distance),
2. *delete, insert* and *replace* (Levenshtein distance, DIR-distance),
3. *delete, insert, replace* and *transpose* of neighbour symbols (Damerau distance, generalized Levenshtein distance, $DIRT$-distance),

needed to convert string x into string y. □

The Hamming distance is a metrics on a set of strings of equal length. The Levenshtein distance and the generalized Levenshtein distance are metrics on a set of strings not necessarily of equal length.

Definition 1.18 (Distance of strings - ordered alphabet)
Let $A = \{a_1, a_2, \ldots, a_p\}$ be an ordered alphabet. Let a_i, a_j be symbols from alphabet A, then the Δ-distance of a_i, a_j is definined as

$$\Delta(a_i, a_j) = |i - j|$$

1. Δ-distance:
   Let x, y be strings over alphabet A such that $|x| = |y|$, then the Δ-distance of x, y is definined as

$$\Delta(x, y) = \max_{i \in \{1..|x|\}} \Delta(x_i, y_i)$$

2. Γ-distance:
   Let x, y be strings over alphabet A such that $|x| = |y|$, then the Γ-distance of x, y is definined as

$$\Gamma(x, y) = \sum_{i \in \{1..|x|\}} \Delta(x_i, y_i)$$

□

Definition 1.19 (Approximate pattern matching - ordered alphabet)
Given text $T = t_1 t_2 \ldots t_n$ and pattern $P = p_1 p_2 \ldots p_m$ over a given ordered alphabet A, then we define:

1. Δ-matching:
   Find all occurrences of string x in text T so that $|x| = |P|$ and $\Delta(P, x) \leq k$, where k is a given positive integer.

2. Γ-matching:
   Find all occurrences of string x in text T so that $|x| = |P|$ and $\Gamma(P, x) \leq k$, where k is a given positive integer.

3. (Δ, Γ)-matching:
   Find all occurrences of string x in text T so that $|x| = |P|$ and $\Delta(P, x) \leq l$ and $\Gamma(P, x) \leq k$, where k, l are given positive integers such that $l \leq k$. □

1.2 Classification of pattern matching problems

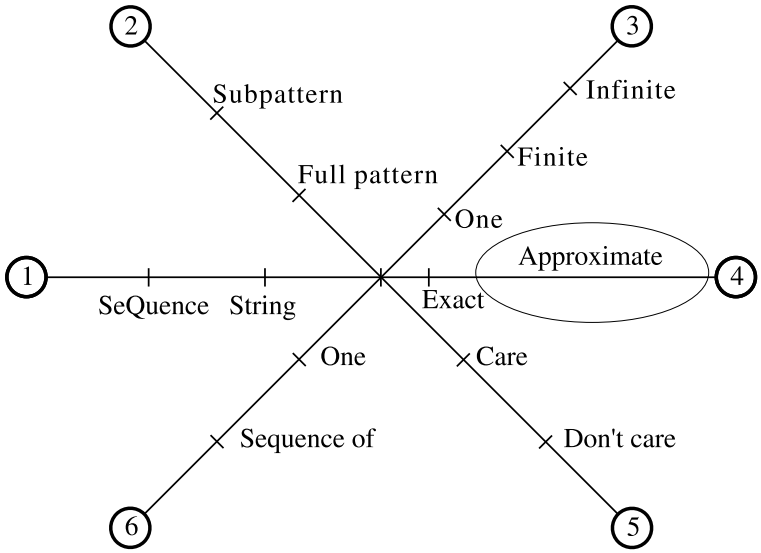
One–dimensional pattern matching problems for a finite size alphabet can be classified according to several criteria. We will use six criteria for a classification leading to a six dimensional space in which one point corresponds to a particular pattern matching problem.

Let us make a list of all dimensions including possible "values" in each dimension:

1. Nature of the pattern:
   - string,
   - sequence.
2. Integrity of the pattern:
   - full pattern,
   - subpattern.
3. Number of patterns:
   - one,
   - finite number greater than one,
   - infinite number.
4. Way of matching:
   - exact,
   - approximate matching with Hamming distance (R-matching),
   - approximate matching with Levenshtein distance (DIR-matching),
   - approximate matching with generalized Levenshtein distance ($DIRT$-matching),
   - Δ-approximate matching,
   - Γ-approximate matching,
   - (Δ, Γ)-approximate matching.
5. Importance of symbols in a pattern:
   - take care above all symbols,
   - don't care above some symbols.
6. Sequences of patterns:
   - one,
   - finite sequence.

The above classification is represented in Figure 1.3. If we count the number of possible pattern matching problems, we obtain $N = 2{\cdot}2{\cdot}3{\cdot}7{\cdot}2{\cdot}2 = 336$.

In order to facilitate references to a particular pattern matching problem, we will use abbreviations for all problems. These abbreviations are summarized in Table 1.1 (D means DIR–matching and G means $DIRT$–matching, generalized Levenshtein distance).



Approximate matching (dimension 4):

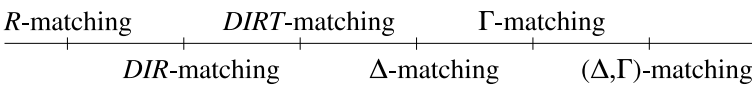


Figure 1.3: Classification of pattern matching problems

Using this method, we can, for example, refer to exact string matching of one string as an *SFOECO* problem.

Instead of a single pattern matching problem we will use the notion of a family of pattern matching problems. In this case we will use symbol ? instead of a particular letter. For example *SFO*??? is the family of all problems concerning one full string matching.

Each pattern matching problem has several instances. For example, an *SFOECO* problem has the following instances:

1. verify whether a given string occurs in the text or not,
2. find the first occurrence of a given string,
3. find the number of all occurrences of a given string,
4. find all occurrences of a given string and their positions.

If we take into account all possible instances, the number of pattern matching problems grows further.

Dimension	1	2	3	4	5	6
	S	F	O	E	C	O
	Q	S	F	R	D	S
			I	D		
				G		
				Δ		
				Γ		
				(Δ, Γ)		

Table 1.1: Abbreviations for pattern matching problems

1.3 Two ways of pattern matching

There are two different ways in which matching of patterns can be performed:

- *forward* pattern matching,
- *backward* pattern matching.

The basic principle of forward pattern matching is depicted in Fig. 1.4. The

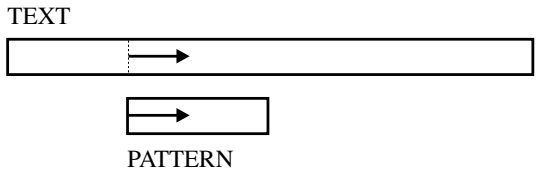


Figure 1.4: Forward pattern matching

text and the pattern are matched in the forward direction. This means that the comparison of symbols is performed from left to right. All algorithms for forward pattern matching must compare each symbol of the text at least once. Therefore the lowest time complexity is equal to the length of the text.

The basic principle of backward pattern matching is depicted in Fig 1.5.

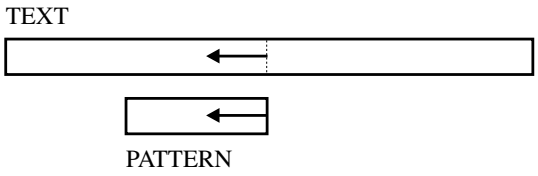


Figure 1.5: Backward pattern matching

The comparison of symbols is performed from right to left. There are three main principles of backward pattern matching:

- looking for a repeated suffix of the pattern,

- looking for a prefix of the pattern,
- looking for an antifactor (a string which is not a factor) of the pattern.

Algorithms for backward pattern matching allow us to skip some part of the text and therefore the number of comparisons can be lower than the length of the text.

1.4 Finite automata

We will use finite automata in all subsequent Chapters as a formalism for the description of various aspects of pattern matching. In this Section we introduce basic notions from the theory of finite automata and we also show some basic algorithms concerning them. The material included is not exhaustive and we recommend using the special literature covering this area in detail.

Definition 1.20 (Deterministic finite automaton)
A deterministic finite automaton (*DFA*) is quintuple $M = (Q, A, \delta, q_0, F)$, where
Q is a finite set of states,
A is a finite input alphabet,
δ is a mapping from $Q \times A$ to Q, $(Q \times A \mapsto Q)$
$q_0 \in Q$ is an initial state,
$F \subset Q$ is the set of final states.

Definition 1.21 (Configuration of FA)
Let $M = (Q, A, \delta, q_0, F)$ be a finite automaton. A pair $(q, w) \in Q \times A^*$ is *a configuration of the finite automaton M*. Configuration (q_0, w) is called *an initial configuration*, configuration (q, ε), where $q \in F$, is called *a final (accepting) configuration* of the finite automaton M.

Definition 1.22 (Transition in DFA)
Let $M = (Q, A, \delta, q_0, F)$ be a deterministic finite automaton. Relation $\vdash_M \in (Q \times A^*) \times (Q \times A^*)$ is called *a transition* in automaton M. If $\delta(q, a) = p$, then $(q, aw) \vdash_M (p, w)$ for each $w \in A^*$. The k-power of the relation $\vdash_M$ will be denoted by $\vdash_M^k$. Symbols $\vdash_M^+$ and $\vdash_M^*$ denote a transitive and a transitive reflexive closure of relation $\vdash_M$, respectively.

Definition 1.23 (Language accepted by DFA)
We will say that input string $w \in A^*$ *is accepted* by finite deterministic automaton $M = (Q, A, \delta, q_0, F)$ if $(q_0, w) \vdash_M^* (q, \varepsilon)$ for some $q \in F$.
Language $L(M) = \{w : w \in T^*, (q_0, w) \vdash^* (q, \varepsilon), q \in F\}$ is *the language accepted* by finite automaton M. String $w \in L(M)$ if it consists only of symbols from the input alphabet and there is a sequence of transitions such that it leads from initial configuration (q_0, w) to final configuration (q, ε), $q \in F$.

Definition 1.24 (Complete DFA)
Finite deterministic automaton $M = (Q, A, \delta, q_0, F)$ is said to be *complete* if the mapping $\delta(q, a)$ is defined for each pair of states $q \in Q$ and input symbols $a \in A$.

Definition 1.25 (Nondeterministic finite automaton)
A nondeterministic finite automaton (*NFA*) is quintuple $M = (Q, A, \delta, q_0, F)$, where
Q is a finite set of states,
A is a finite input alphabet,
δ is a mapping from $Q \times A$ into the set of subsets of Q,
$q_0 \in Q$ is an initial state,
$F \subset Q$ is the set of final states.

Definition 1.26 (Transition in NFA)
Let $M = (Q, A, \delta, q_0, F)$ be a nondeterministic finite automaton. Relation $\vdash_M \subset (Q \times A^*) \times (Q \times A^*)$ will be called *a transition* in automaton M if $p \in \delta(q,a)$ then $(q, aw) \vdash_M (p, w)$, for each $w \in A^*$.

Definition 1.27 (Language accepted by NFA)
String $w \in A^*$ is said to be *accepted by nondeterministic finite automaton* $M = (Q, A, \delta, q_0, F)$, if there exists a sequence of transitions $(q_0, w) \vdash^* (q, \varepsilon)$ for some $q \in F$. Language $L(M) = \{w : w \in A^*, (q_0, w) \vdash^* (q, \varepsilon)$ for some $q \in F\}$ is then the language accepted by nondeterministic finite automaton M.

Definition 1.28 (NFA with ε-transitions)
A nondeterministic finite automaton with ε-transitions is quintuple $M = (Q, A, \delta, q_0, F)$, where
Q is a finite set of states,
A is a finite input alphabet,
δ is a mapping from $Q \times (A \cup \{\varepsilon\})$ into the set of subsets of Q,
$q_0 \in Q$ is an initial state,
$F \subset Q$ is the set of final states.

Definition 1.29 (Transition in NFA with ε-transitions)
Let $M = (Q, A, \delta, q_0, F)$ be a nondeterministic finite automaton with ε-transitions. Relation $\vdash_M \subset (Q \times A^*) \times (Q \times A^*)$ will be called *a transition* in automaton M if $p \in \delta(q,a)$, $a \in A \cup \{\varepsilon\}$, then $(q, aw) \vdash_M (p, w)$, for each $w \in A^*$.

Definition 1.30 ($\varepsilon - CLOSURE$)
Function $\varepsilon - CLOSURE$ for finite automaton $M = (Q, A, \delta, q_0, F)$ is defined as:

$$\varepsilon - CLOSURE(q) = \{p : (q, \varepsilon) \vdash^* (p, \varepsilon), p \in Q\}.$$

Definition 1.31 (NFA with a set of initial states)
Nondeterministic finite automaton M with the set of initial states I is quin-

tuple $M = (Q, A, \delta, I, F)$, where:
Q is a finite set of states,
A is a finite input alphabet,
δ is a mapping from $Q \times A$ into the set of subsets of Q,
$I \subset Q$ is the non-empty set of initial states,
$F \subset Q$ is the set of final states.

Definition 1.32 (Accessible state)
Let $M = (Q, A, \delta, q_0, F)$ be a finite automaton. State $q \in Q$ is called *accessible* if there exists string $w \in A^*$ such that there exists a sequence of transitions from initial state q_0 into state q:

$(q_0, w) \vdash_M (q, \varepsilon)$

A state which is not accessible is called *inaccessible*.

Definition 1.33 (Useful state)
Let $M = (Q, A, \delta, q_0, F)$ be a finite automaton. State $q \in Q$ is called *useful* if there exists a string $w \in A^*$ such that there exists a sequence of transitions from state q into some final state:
$(q, w) \vdash_M (p, \varepsilon)$, $p \in F$.

A state which is not useful is called *useless*.

Definition 1.34 (Finite automaton)
Finite automaton (*FA*) is DFA or NFA.

Definition 1.35 (Equivalence of finite automata)
Finite automata M_1 and M_2 are said to be *equivalent* if they accept the same language, i.e., $L(M_1) = L(M_2)$.

Definition 1.36 (Sets of states)
Let $M = (Q, A, \delta, q_0, F)$ be a finite automaton. Let us define for arbitrary $a \in A$ set $Q(a) \subset Q$ as follows:
$Q(a) = \{q : q \in \delta(p, a), a \in A, p, q \in Q\}$.

Definition 1.37 (Homogenous automaton)
Let $M = (Q, A, \delta, q_0, F)$ be a finite automaton and $Q(a)$ be sets of states for all symbols $a \in T$. If for all pairs of symbols $a, b \in A$, $a \neq b$, it holds $Q(a) \cap Q(b) = \emptyset$, then automaton M is called homogeneous. The collection of sets $\{Q(a) : a \in A\}$ is for the homogeneous finite automaton a decomposition on classes having one of these two forms:

1. $Q = \bigcup_{a \in A} Q(a) \cup \{q_0\}$ in the case that $q_0 \notin \delta(q, a)$ for all $q \in Q$ and all $a \in A$,

2. $Q = \bigcup_{a \in A} Q(a)$ in the case that $q_0 \in \delta(q, a)$ for some $q \in Q, a \in A$. In this case $q_0 \in Q(a)$.

Algorithm 1.38
Construction of a nondeterministic finite automaton without ε-transitions equivalent to a nondeterministic finite automaton with ε-transitions.
Input: Finite automaton $M = (Q, A, \delta, q_0, F)$ with ε-transitions.
Output: Finite automaton $M' = (Q, A, \delta', q_0, F')$ without ε-transitions equivalent to M.
Method:

1. $\delta'(q, a) = \bigcup_{p \in \varepsilon - CLOSURE(q)} \delta(p, a)$.
2. $F' = \{q : \varepsilon - CLOSURE(q) \cap F \neq \emptyset, q \in Q\}$. □

Algorithm 1.39
Construction of a nondeterministic finite automaton with a single initial state equivalent to a nondeterministic finite automaton with several initial states.
Input: Finite automaton $M = (Q, A, \delta, I, F)$ with a nonempty set I.
Output: Finite automaton $M' = (Q', A, \delta', q_0, F)$ with a single initial state q_0.
Method: Automaton M' will be constructed using the following two steps:

1. $Q' = Q \cup \{q_0\}$, $q_0 \notin Q$,
2. $\delta'(q_0, \varepsilon) = I$,
   $\delta'(q, a) = \delta(q, a)$ for all $q \in Q$ and all $a \in A$. □

The next Algorithm constructs a deterministic finite automaton equivalent to a given nondeterministic finite automaton. The construction used is called a *subset construction.*

Algorithm 1.40
Transformation of a nondeterministic finite automaton to a deterministic finite automaton.
Input: Nondeterministic finite automaton $M = (Q, A, \delta, q_0, F)$.
Output: Deterministic finite automaton $M' = (Q', A, \delta', q_0', F')$ such that $L(M) = L(M')$.
Method:

1. Set $Q' = \{\{q_0\}\}$ will be defined, state $q_0' = \{q_0\}$ will be treated as unmarked. (Please note that each state of a deterministic automaton consists of a set of state of a nondeterministic automaton.)
2. If each state in Q' is marked then continue with step 4.
3. Unmarked state q' will be chosen from Q' and the following operations will be executed:

(a) $\delta'(q', a) = \bigcup \delta(p, a)$ for $p \in q'$ and for all $a \in A$,
(b) $Q' = Q' \cup \delta'(q', a)$ for all $a \in A$,
(c) state $q' \in Q'$ will be marked,
(d) continue with step 2.

4. $q'_0 = \{q_0\}$.
5. $F' = \{q' : q' \in Q', q' \cap F \neq \emptyset\}$. □

Note: Let us mention that all states of the resulting deterministic finite automaton M' are accessible states. (See Def. 1.32)

Definition 1.41 (d–subset)
Let $M_1 = (Q_1, A, \delta_1, q_{01}, F_1)$ be a nondeterministic finite automaton. Let $M_2 = (Q_2, A, \delta_2, q_{02}, F_2)$ be the deterministic finite automaton equivalent to automaton M_1. Automaton M_2 is constructed using the standard determinization algorithm based on subset construction (see Alg. 1.40). Every state $q \in Q_2$ corresponds to some subset d of Q_1. This subset will be called a *d–subset* (deterministic subset). □

Notational convention:
A d-subset created during the determinization of a nondeterministic finite automaton has the form: $\{q_{i1}, q_{i2}, \ldots, q_{in}\}$. If no confusion arises we will write such d-subset as $q_{i1}q_{i2}\ldots q_{in}$.

Definition 1.42
A d-subset is *simple* if it contains just one element. The corresponding state to it is called a *simple state*. A d-subset is *multiple* if it contains more than one element. The corresponding state to it will be called a *multiple state*.□

Algorithm 1.43
Construction of a finite automaton for an union of languages.
Input: Two finite automata M_1 and M_2.
Output: Finite automaton M accepting the language $L(M) = L(M_1) \cup L(M_2)$.
Method:

1. Let $M_1 = (Q_1, A, \delta_1, q_{01}, F_1),\ M_2 = (Q_2, A, \delta_2, q_{02}, F_2), Q_1 \cap Q_2 = \emptyset$.
2. The resulting automaton $M = (Q, A, \delta, q_0, F)$ is constructed using the following steps:
   (a) $Q = Q_1 \cup Q_2 \cup \{q_0\},\ q_0 \notin Q_1 \cup Q_2$,
   (b) $\delta(q_0, \varepsilon) = \{q_{01}, q_{02}\}$,
   $\delta(q, a) = \delta_1(q, a)$ for all $q \in Q_1$ and all $a \in A$,
   $\delta(q, a) = \delta_2(q, a)$ for all $q \in Q_2$ and all $a \in A$.

3. $F = F_1 \cup F_2$. □

Algorithm 1.44
Construction of a finite automaton for the intersection of two languages.
Input: Two finite automata $M_1 = (Q_1, A, \delta_1, q_{01}, F_1), M_2 = (Q_2, A, \delta_2, q_{02}, F_2)$.
Output: Finite automaton $M = (Q, A, \delta, q_0, F)$, accepting language $L(M) = L(M_1) \cap L(M_2)$.
Method:

1. Let $Q = \{(q_{01}, q_{02})\}$. State (q_{01}, q_{02}) will be treated as unmarked.
2. If all states in Q are marked go to step 4.
3. Take any unmarked state $q = (q_{n1}, q_{m2})$ from Q and perform these operations:
   (a) determine $\delta((q_{n1}, q_{m2}), a) = (\delta_1(q_{n1}, a), \delta_2(q_{m2}, a))$ for all $a \in A$,
   (b) if both transitions $\delta_1(q_{n1}, a)$ and $\delta_2(q_{m2}, a)$ are defined then $Q = Q \cup (\delta_1(q_{n1}, a), \delta_2(a_{m2}, a))$ and state $(\delta_1(q_{n1}, a), \delta_2(q_{m2}, a))$ will be treated as unmarked only if it is a new state in Q,
   (c) state (q_{n1}, q_{m2}) in Q will be treated as marked,
   (d) go to step 2.
4. $q_0 = (q_{01}, q_{02})$.
5. $F = \{q : q \in Q,\ q = (q_{n1}, q_{m2}),\ q_{n1} \in F,\ q_{m2} \in F\}$. □

1.5 Regular expressions

1.5.1 Definition of regular expressions

Definition 1.45
Regular expression V over alphabet A is defined as follows:

1. $\emptyset, \varepsilon, a$ are regular expressions for all $a \in A$.
2. If x, y are regular expressions over A then:
   (a) $(x + y)$ (union)
   (b) $(x \cdot y)$ (concatenation)
   (c) $(x)^*$ (closure)

   are regular expressions over A.

Definition 1.46
The *value h(x) of regular expression x* is defined as follows:

1. $h(\emptyset) = \emptyset,\ h(\varepsilon) = \{\varepsilon\},\ h(a) = \{a\}$,
2. $h(x + y) = h(x) \cup h(y)$,
   $h(x \cdot y) = h(x) \cdot h(y)$,
   $h(x^*) = (h(x))^*$.

The value of any regular expression is a regular language, and each regular language can be represented by some regular expression. Unnecessary parentheses in regular expressions can be avoided by the convention that precedence is given to regular operations. The closure operator has the highest precedence, and the union operator has the lowest precedence.

The following axioms are defined for regular expressions:

$A_1 : x + (y + z) = (x + y) + z$ (union associativity),
$A_2 : x \cdot (y \cdot z) = (x \cdot y) \cdot z$ (concatenation associativity),
$A_3 : x + y = y + x$ (union commutativity),
$A_4 : (x + y) \cdot z = x \cdot z + y \cdot z$ (distributivity from the right),
$A_5 : x \cdot (y + z) = x \cdot y + x \cdot z$ (distributivity from the left),
$A_6 : x + x = x$ (union idempotention),
$A_7 : \varepsilon x = x$ (ε is a unitary element for the operation concatenation),
$A_8 : \emptyset x = \emptyset$ ($\emptyset$ is a zero element for the operation concatenation),
$A_9 : x + \emptyset = x$ ($\emptyset$ is a zero element for the operation union),
$A_{10} : x^* = \varepsilon + x^* x$
$A_{11} : x^* = (\varepsilon + x)^*$
$A_{12} : x = x\alpha + \beta \Rightarrow x = \beta\alpha^*$ (solution of left regular equation),
$A_{13} : x = \alpha x + \beta \Rightarrow x = \alpha^*\beta$ (solution of right regular equation).

It has been proven that all other equalities between regular expressions can be derived from these axioms.

1.5.2 The relation between regular expressions and finite automata

It is possible to construct for each regular expression V an equivalent finite automaton M, which means for such on automaton that $h(V) = L(M)$. There are several techniques for building up a finite automaton for a given regular expression. The method shown here is based on the notion of adjacent symbols.

Algorithm 1.47
Construction of an equivalent finite automaton for a given regular expression.

Input: Regular expression V.
Output: Finite automaton $M = (Q, T, \delta, q_0, F)$ such that $h(V) = L(M)$.
Method: Let A be an alphabet, over which expression V is defined.

1. Number by numbers $1, 2, ..., n$ all occurrences of symbols from A in expression V so that each two occurrences of the same symbol will be numbered by different numbers. The resultant regular expression will be denoted by V'.
2. Build up the set of start symbols
   $Z = \{x_i : x \in A$, some string from $h(V')$ may start with symbol $x_i\}$.
3. Construct the set P of adjacent symbols:
   $P = \{x_i y_j$: symbols x_i and y_j may be adjacent in some string from $h(V')\}$.
4. Construct set of final symbols F in the following way:
   $F = \{x_i$: some string from $h(V')$ may end with symbol $x_i\}$.
5. Set of states of the finite automaton
   $Q = \{q_0\} \cup \{x_i : x \in T, i \in <1, n>\}$.
6. Mapping δ will be constructed in the following way:
   (a) $\delta(q_0, x)$ includes x_i for each $x_i \in Z$ such that x_i was created by the numbering of x.
   (b) $\delta(x_i, y)$ includes y_j for each couple $x_i y_j \in P$ such that y_j was created by the numbering of y.
   (c) Set F is the set of final states.

2 Forward pattern matching

The basic principles of the forward pattern matching approach are discussed in this Chapter. We will discuss two approaches from the classification shown in Fig. 1.2:

1. Neither the pattern nor the text is preprocessed. We will introduce two programs implementing elementary algorithms for exact and approximate pattern matching using Hamming distance, in both cases for a single pattern.
2. The pattern is preprocessed but the text is not preprocessed.

The preprocessing of the pattern is divided into several steps. The first step is the construction of a nondeterministic finite automaton which will serve as a model of the solution of the pattern matching problem in question. Using this model as a basis for the next step, we can construct either ae deterministic finite automaton or a simulator, both equivalent to the basic model. If the result of the preprocessing is a deterministic finite automaton then the pattern matching is preformed so that the text is read as the input of the automaton. Both the nondeterministic finite automaton as a model and the equivalent deterministic finite automaton are constructed as automata that are able to read any text. An occurrence of the pattern in the text is found when the automaton is reaching a final state. Finding the pattern is then reported and reading of the text continues in order to find all following occurrences of the pattern including overlapping cases.

This approach using deterministic finite automata has one advantage: each symbol of the text is read just once. If we take the number of steps performed by the automaton as a measure of the time complexity of the forward pattern matching then it is equal to the length of the text. On the other hand, the use of deterministic finite automata may have space problems with space complexity. The number of states of a deterministic finite automaton may in some cases be very large in comparison with the length of the pattern. Approximate pattern matching in an example of this case. It is a limitation of this approach. A solution of this "space" problem is the use of simulators of nondeterministic finite automata. We will show three types of such simulators in the next Chapters:

1. use of the "fail" function,
2. dynamic programming, and
3. bit parallelism.

The space complexity of all of these simulators is acceptable. The time complexity is greater than for deterministic finite automata in almost all cases. It is even quadratic for dynamic programming.

Let us recall that text $T = t_1 t_2 \dots t_n$ and pattern $P = p_1 p_2 \dots p_m$, and all symbols of both the text and the pattern are from alphabet A.

2.1 Elementary algorithm

The elementary algorithm compares the symbols of the pattern with the symbols of the text. The principle of this approach is shown in Fig. 2.1.

```
var TEXT[1..N] of char;
    PATTERN[1..M] of char;
    I,J: integer;
begin
    I:=0;
    while I≤ N-M do
    begin
        J:=0;
        while J<M and PATTERN[J+1]=TEXT[I+J+1] do J:=J+1;
        if J=M then output(I+1);
        I:=I+1; {length of shift=1}
    end;
end;
```

Figure 2.1: Elementary algorithm for exact matching of one pattern

The program presented here implements the algorithm performing the exact matching of one pattern. The meanings of the variables used in the program are represented in Fig. 2.2. When the pattern is found then the

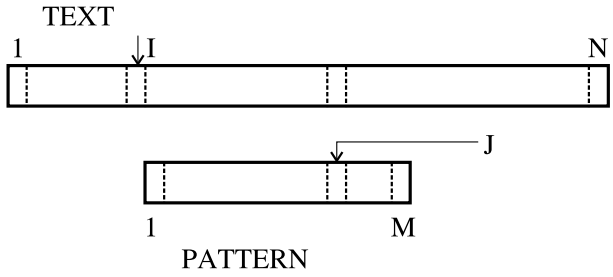


Figure 2.2: Meaning of variables in the program from Fig. 2.1

value of variable I is the index of the position just before the first symbol of the occurrence of the pattern in the text. Comment {length of shift=1} means that the pattern is shifted one position to the right after each mismatch or finding the pattern. The term *shift* will be used later.

We will use the number of symbol comparisons (see expression PATTERN[J+1]=TEXT[I+J+1]) as the measure for the complexity of the algorithm. The maximum number of symbol comparisons for the elementary algorithm is

$$NC = (n - m + 1) * m, \tag{1}$$

where n is the length of the text and m is the length of the pattern. We

assume that $n \gg m$. The time complexity is $\mathcal{O}(n * m)$. The maximum number of comparisons NC is reached for text $T = a^{n-1}b$ and for pattern $P = a^{m-1}c$, where $a, b, c \in A$, $c \neq a$. Elementary algorithm has no extra space requirements.

The experimental measurements show that the number of comparisons of elementary algorithm for texts written in natural languages is linear with respect to the length of the text. It has been observed that a mismatch of the symbols is reached very soon (at the first or second symbol of the pattern). The number of comparisons in this case is:

$$NC_{nat} = C_L * (n - m + 1), \tag{2}$$

where C_L is a constant given by the experiments for given language L. The value of this constant for English is $C_E = 1.07$. Thus, the elementary algorithm has linear time complexity ($\mathcal{O}(n)$) for the pattern matching in natural language texts.

The elementary algorithm can be used for matching a finite set of patterns. In this case, the algorithm is used for each pattern separately. The time complexity is

$$\mathcal{O}(n * \sum_{i=1}^{s} m_i),$$

where s is the number of patterns in the set and m_i is the length of the i–th pattern, $i = 1, 2, \ldots, s$.

The next variant of the elementary algorithm is for approximate pattern matching of one pattern using Hamming distance. It is shown in Fig. 2.3.

2.2 Pattern matching automata

In this Section, we will show basic models of pattern matching algorithms. Moreover, we will show how to construct models for more complicated problems using models of simple problems.

Notational convention:
We replace names of states q_i, q_{ij} by i, ij, respectively, in subsequent transition diagrams. The reason for this is to improve the readability.

2.2.1 Exact string and sequence matching

The model of the algorithm for exact string matching (*SFOECO* problem) for pattern $P = p_1p_2p_3p_4$ is shown in Fig. 2.4. The *SFOECO* nondeterministic finite automaton is constructed in this way:

1. Create the automaton accepting pattern P.
2. Insert the selfloop for all symbols from alphabet A in the initial state.

```
var TEXT[1..N] of char;
    PATTERN[1..M] of char;
    I,J,K,NERR: integer;
    K:=number of errors allowed;
begin
    I:=0;
    while I≤ N-M do
    begin
        J:=0;
        NERR:=0
        while J<M and NERR<K do
        begin if PATTERN[J+1]≠ TEXT[I+J+1] then NERR:=NERR+1;
            J:=J+1
        end;
        if J=M then output(I+1);
        I:=I+1; {length of shift=1}
    end;
end;
```

Figure 2.3: Elementary algorithm for approximate matching of one pattern using Hamming distance

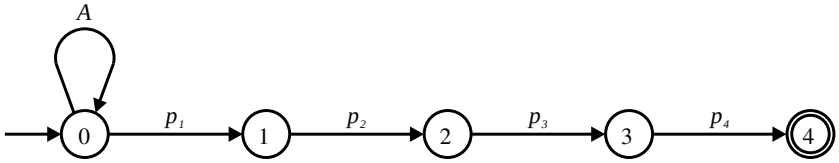


Figure 2.4: Transition diagram of *NFA* for exact string matching (*SFOECO* automaton) for pattern $P = p_1p_2p_3p_4$

Algorithm 2.1 describes the construction of the *SFOECO* automaton in detail.

Algorithm 2.1
Construction of the *SFOECO* automaton.
Input: Pattern $P = p_1p_2 \dots p_m$.
Output: *SFOECO* automaton M.
Method: *NFA* $M = (\{q_0, q_1, \dots, q_m\}, A, \delta, q_0, \{q_m\})$, where mapping δ is constructed in the following way:

1. $q_{i+1} \in \delta(q_i, p_{i+1})$ for $0 \leq i < m$,
2. $q_0 \in \delta(q_0, a)$ for all $a \in A$. □

The *SFOECO* automaton has $m + 1$ states for a pattern of length m.

A model of the algorithm for exact sequence matching (*QFOECO* problem) for pattern $P = p_1p_2p_3p_4$ is shown in Fig. 2.5. The *QFOECO* nonde-

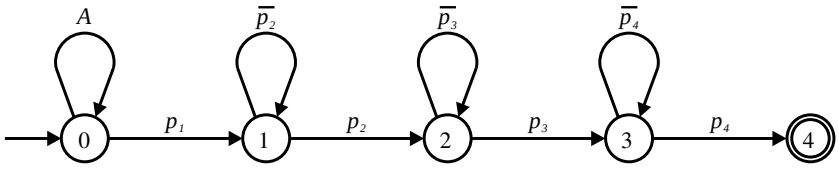


Figure 2.5: Transition diagram of *NFA* for exact sequence matching (*QFOECO* automaton) for pattern $P = p_1p_2p_3p_4$

terministic finite automaton is constructed as the *SFOECO* automaton with the addition of some new selfloops. The new selfloops are added in all states but the initial and final ones for all symbols, with the exception of the symbol for which there is already the transition to the next state. Algorithm 2.2 describes the construction of the *QFOECO* automaton in detail.

Algorithm 2.2
Construction of the *QFOECO* automaton.
Input: Pattern $P = p_1p_2 \dots p_m$.
Output: *QFOECO* automaton M.
Method: *NFA* $M = (\{q_0, q_1, \dots, q_m\}, A, \delta, q_0, \{q_m\})$, where mapping δ is constructed in the following way:

1. $q_i \in \delta(q_i, a)$ for $0 < i < m$ and all $a \in A$ and $a \neq p_{i+1}$.
2. $q_0 \in \delta(q_0, a)$ for all $a \in A$,
3. $q_{i+1} \in \delta(q_i, p_{i+1})$ for $0 \leq i < m$. □

The *QFOECO* automaton has $m + 1$ states for a pattern of length m.

2.2.2 Substring and subsequence matching

A model of the algorithm for exact substring matching (*SSOECO* problem) for pattern $P = p_1p_2p_3p_4$ is shown in Fig. 2.6.

Notational convention:
The following nondeterministic finite automata have a regular structure. For clarity of expandion, we will use the following terminology:
State q_{ij} is at *depth* i (a position in the pattern) and on *level* j.

The *SSOECO* nondeterministic finite automaton is constructed by composing the collection of m copies of *SFOECO* automata. The composition is done by inserting ε-transitions. These ε-transitions are inserted in the "diagonal" direction. They start from the initial state of the zero level and are directed to the next level of it. The next ε-transitions always start from the end state of the previous ε-transition. As the final step, the inaccessible states are removed. Algorithm 2.3 describes the construction of the *SSOECO* automaton in detail.

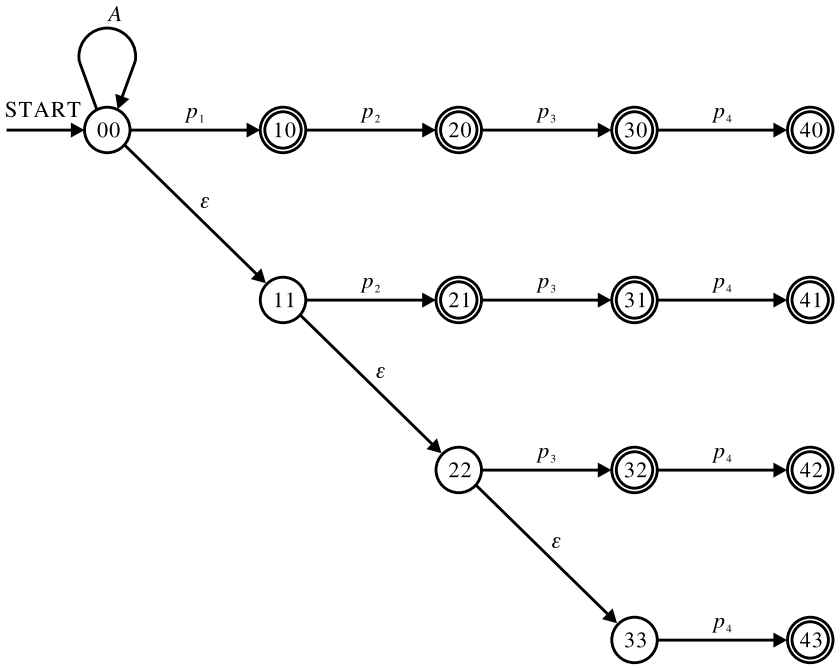


Figure 2.6: Transition diagram of *NFA* for exact substring matching (*SSOECO* automaton) for pattern $P = p_1p_2p_3p_4$

Algorithm 2.3
Construction of the *SSOECO* automaton.
Input: Pattern $P = p_1p_2 \ldots p_m$, *SFOECO* automaton $M' = (Q', A, \delta', q_0', F')$ for P.
Output: *SSOECO* automaton M.
Method:

1. Create a sequence of m instances of *SFOECO* automata for pattern P $M_j' = (Q_j', A, \delta_j', q_{0j}, F_j')$ for $j = 0, 1, 2, \ldots, m-1$. Let the states in Q_j' be
   $q_{0j}, q_{1j}, \ldots, q_{mj}$.
2. Construct automaton $M = (Q, A, \delta, q_0, F)$ as follows:
   $Q = \bigcup_{j=0}^{m-1} Q_j'$,

$\delta(q,a) = \delta_j^{'}(q,a)$ for all $q \in Q$, $a \in A$, $j = 0,1,2,\ldots,m-1$,
$\delta(q_{00},\varepsilon) = \{q_{11}\}$,
$\delta(q_{11},\varepsilon) = \{q_{22}\}$,
$\vdots$
$\delta(q_{m-2,m-2},\varepsilon) = \{q_{m-1,m-1}\}$,
$q_0 = q_{00}$,
$F = Q \setminus \{q_{00}, q_{11}, \ldots, q_{m-1,m-1}\}$.

3. Remove all states which are inaccessible from state q_0. □

The *SSOECO* automaton has $(m+1)+m+(m-1)+\ldots+2 = \frac{m(m+3)}{2}$ states. The *SSOECO* automaton can be minimized. The direct construction of the main part of the minimized version of this automaton is described by Algorithm 3.14 (construction of factor automaton) and shown in Example 3.15. It is enough to add selfloops in the initial state for all symbols of the alphabet in order to obtain the *SSOECO* automaton. The advantage of the nonminimized *SSOECO* automaton is that the unique state corresponds to each substring of the pattern.

A model of the algorithm for exact subsequence matching (*QSOECO* problem) for pattern $P = p_1p_2p_3p_4$ is shown in Fig. 2.7. Construction of the *QSOECO* nondeterministic finite automaton starts in the same way as for the *SSOECO* automaton. The final part of this construction is addition of the ε-transitions. The "diagonal" ε-transitions star in all states having transitions to following states on levels from 0 to $m-1$. Algorithm 2.4 describes the construction of the *QSOECO* automaton in detail.

Algorithm 2.4
Construction of the *QSOECO* automaton.
Input: Pattern $P = p_1p_2\ldots p_m$, *QFOECO* automaton $M' = (Q', A, \delta', q_0', F')$ for P.
Output: *QSOECO* automaton M.
Method:

1. Create a sequence of m instances of *QFOECO* automata for pattern P $M_j' = (Q_j', A, \delta_j', q_{0j}, F_j')$ for $j = 0,1,2,\ldots,m-1$. Let the states in Q_j' be
$q_{0j}, q_{1j}, \ldots, q_{mj}$.
2. Construct automaton $M = (Q, A, \delta, q_0, F)$ as follows:
$Q = \bigcup_{j=0}^{m-1} Q_j'$,
$\delta(q,a) = \delta_j^{'}(q,a)$ for all $q \in Q$, $a \in A$, $j = 0,1,2,\ldots,m-1$,
$\delta(q_{ij},\varepsilon) = \{q_{i+1,j+1}\}$, for $i = 0,1,\ldots,m-1$, $j = 0,1,2,\ldots,m-1$,
$q_0 = q_{00}$,
$F = Q \setminus \{q_{00}, q_{11}, \ldots, q_{m-1,m-1}\}$.
3. Remove the states which are inaccessible from state q_0. □

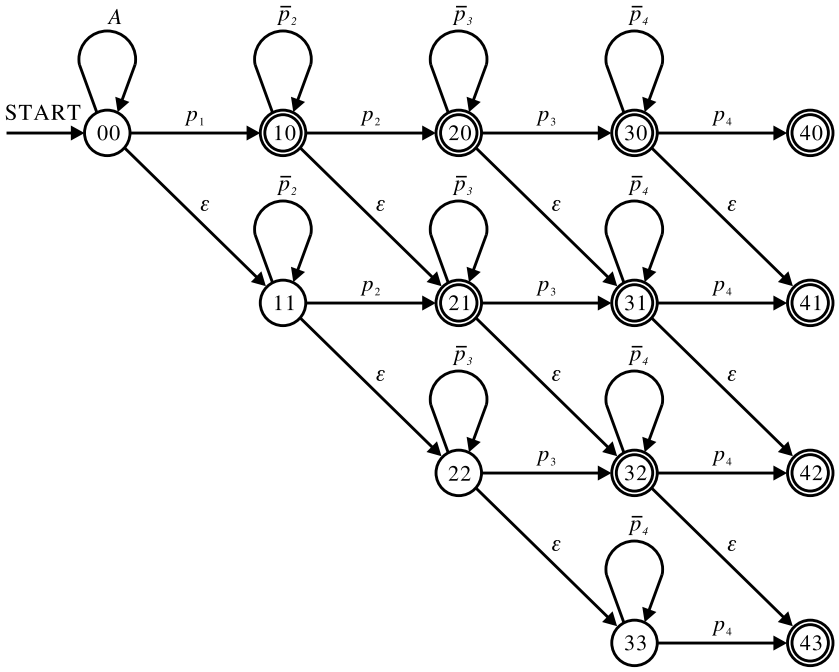


Figure 2.7: Transition diagram of *NFA* for exact subsequence matching (*QSOECO* automaton) for pattern $P = p_1p_2p_3p_4$

The *QSOECO* automaton has $(m+1) + m + (m-1) + \ldots + 2 = \frac{m(m+3)}{2}$ states.

2.2.3 Approximate string matching - general alphabet

We will discuss three variants of approximate string matching corresponding to the three definitions of distances between strings in the general alphabet: Hamming distance, Levenshtein distance, and generalized Levenshtein distance.

Note:
The notion *level* of the state corresponds to the number of errors in the nondeterministic finite automata for approximate pattern matching.

2.2.3.1 Hamming distance Let us recall that the Hamming distance (R-distance) between strings x and y is equal to the minimum number of editing operations replace which are necessary to convert string x into string y (see Def. 1.17). This type of string matching using R-distance is called string R-matching.

A model of the algorithm for string R-matching (*SFORCO* problem) for string $P = p_1p_2p_3p_4$ is shown in Fig. 2.8. The construction of the *SFORCO*

nondeterministic finite automaton again uses a composition of *SFOECO* automata similarly to the construction of *SSOECO* or *QSOECO* automata. The composition is done in this case by inserting "diagonal" transitions starting in all states having transitions to next states on levels from 0 to $k-2$. The diagonal transitions are labelled by all symbols for which no transition to the next state exists. These transitions represent replace operations. Algorithm 2.5 describes the construction of the *SFORCO* automaton in detail.

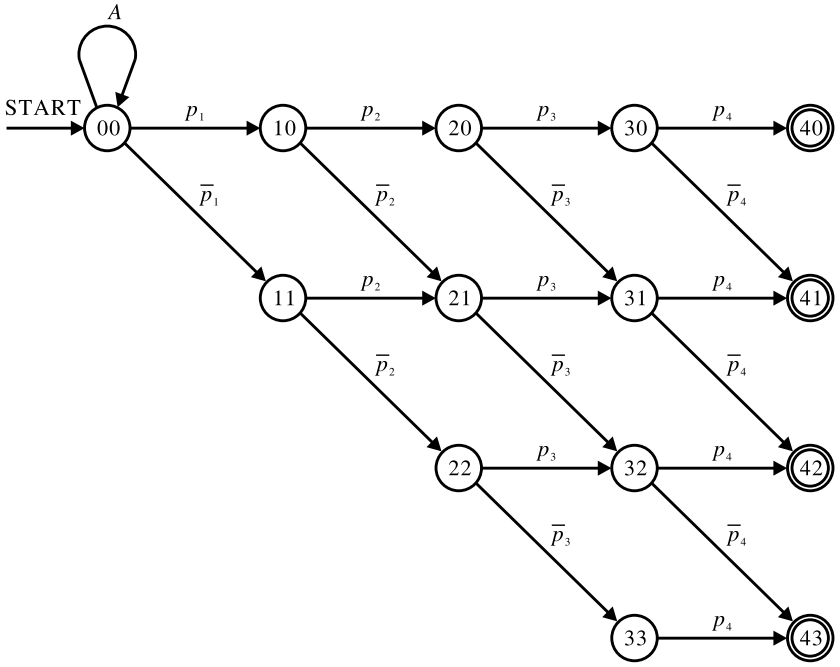


Figure 2.8: Transition diagram of *NFA* for string R-matching ($SFORCO$ automaton) for pattern $P = p_1p_2p_3p_4,\ k = 3$

Algorithm 2.5
Construction of the $SFORCO$ automaton.
Input: Pattern $P = p_1p_2 \ldots p_m, k$, *SFOECO* automaton $M' = (Q', A, \delta', q_0', F')$ for P.
Output: $SFORCO$ automaton M.
Method:

1. Create a sequence of $k+1$ instances of *SFOECO* automata $M_j' = (Q_j', A, \delta_j', q_{0j}', F_j')$ for $j = 0, 1, 2, \ldots, k$. Let states in Q_j' be $q_{0j}, q_{1j}, \ldots, q_{mj}$.
2. Construct $SFORCO$ automaton $M = (Q, A, \delta, q_0, F)$ as follows: $Q = \bigcup_{j=0}^{k} Q_j'$,

$\delta(q,a) = \delta'_j(q,a)$ for all $q \in Q$, $a \in A$, $j = 0,1,2,\ldots,k$,
$\delta(q_{ij},a) = \{q_{i+1,j+1}\}$, for all $i = 0,1,\ldots,m-1$, $j = 0,1,2,\ldots,k-1$,
$a \in A \setminus \{p_{i+1}\}$,
$q_0 = q_{00}$,
$F = \bigcup_{j=0}^{k} F'_j$.

3. Remove all states which are inaccessible from state q_0. □

The $SFORCO$ automaton has $(m+1)+m+(m-1)+\ldots+m-k+1 = \frac{m(k+1)+1-(k(k-1))}{2}$ states.

2.2.3.2 Levenshtein distance Let us recall that the Levenshtein distance (DIR distance) between strings x and y is equal to the minimum number of editing operations delete, insert and replace which are necessary to convert string x into string y (see Def. 1.17). This type of string matching using DIR-distance is called string DIR-matching. A model of the algorithm for string DIR-matching ($SFODCO$ problem) for the string $P = p_1p_2p_3p_4$ is shown in Fig. 2.9. Construction of the $SFODCO$ nondeterministic finite automaton is performed by an extension of the $SFORCO$ automaton. The extension is done by the following two operations:

1. Adding ε-transitions parallel to the "diagonal" transition of the $SFORCO$ automaton. These represent delete operations.
2. Adding "vertical" transitions starting in all states as ε-transitions. Labelling added "vertical" transitions is the same as for diagonal transitions. The "vertical" transitions represent insert operations.

Algorithm 2.6 describes the construction of the $SFODCO$ automaton in detail.

Algorithm 2.6
Construction of the $SFODCO$ automaton.
Input: Pattern $P = p_1p_2\ldots p_m, k$, $SFORCO$ automaton $M' = (Q, A, \delta', q_0, F)$ for P.
Output: $SFODCO$ automaton M for P.
Method: Let states in Q be

$$\begin{array}{l} q_{00}, q_{10}, q_{20}, \ldots, q_{m0}, \\ \quad q_{11}, q_{21}, \ldots, q_{m1}, \\ \qquad \vdots \\ \qquad q_{k,k}, \ldots, q_{mk}. \end{array}$$

Construct $SFODCO$ automaton $M = (Q, A, \delta, q_0, F)$ as follows:
$\delta(q,a) = \delta'(q,a)$ for all $q \in Q$, $a \in A$,
$\delta(q_{ij},\varepsilon) = \{q_{i+1,j+1}\}$, for $i = 0,1,\ldots,m-1$, $j = 0,1,2,\ldots,k-1$,
$\delta(q_{ij},a) = \{q_{i,j+1}\}$, for all $i = 1,2,\ldots,m-1$, $j = 0,1,2,\ldots,k-1$, $a \in A \setminus \{p_{i+1}\}$. □

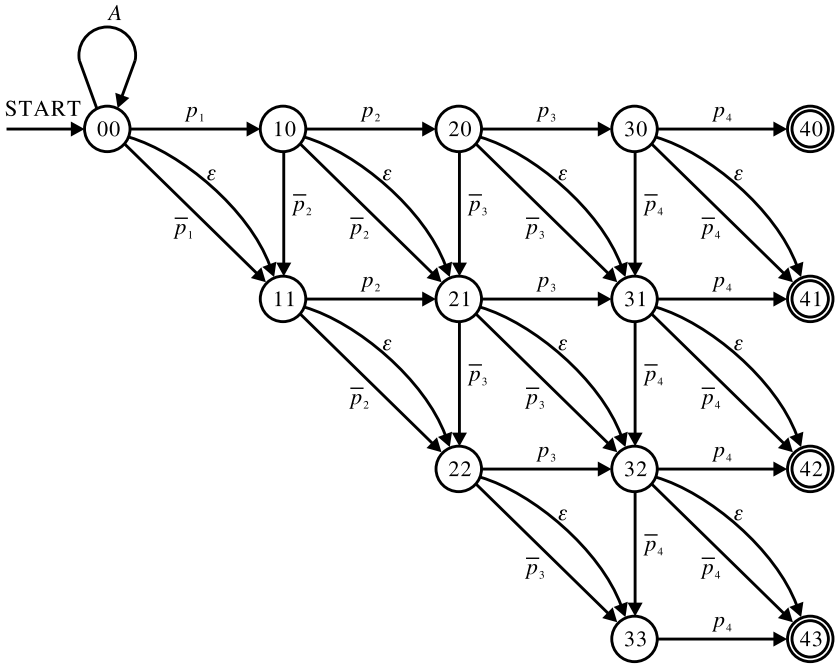


Figure 2.9: Transition diagram of *NFA* for string DIR-matching ($SFODCO$ automaton) for pattern $P = p_1p_2p_3p_4,\ k = 3$

2.2.3.3 Generalized Levenshtein distance Let us recall that the generalized Levenshtein distance ($DIRT$-distance) between strings x and y is equal to the minimum number of editing operations delete, insert, replace and transpose which are necessary to convert string x into string y (see Def 1.17). This type of string matching using $DIRT$-distance is called string $DIRT$-matching.

A model of the algorithm for string $DIRT$-matching ($SFOGCO$ problem) for pattern $P = p_1p_2p_3p_4$ is shown in Fig. 2.10. Construction of the $SFOGCO$ nondeterministic finite automaton is performed by an extension of the
$SFODCO$ automaton. The extension is done by adding the "flat diagonal" transitions representing transpose operations of neighbor symbols. Algorithm 2.7 describes the construction of the $SFOGCO$ automaton in detail.

Algorithm 2.7
Construction of the $SFOGCO$ automaton.
Input: Pattern $P = p_1p_2 \dots p_m, k$, $SFODCO$ automaton $M^{'} = (Q^{'}, A, \delta^{'}, q_0, F)$ for P.
Output: $SFOGCO$ automaton M for P.
Method: Let states in $Q^{'}$ be

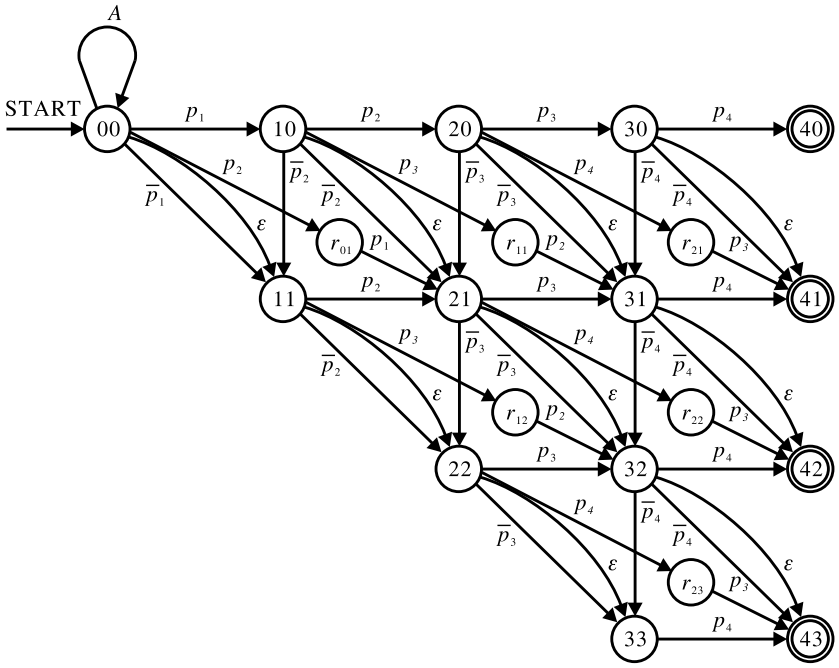


Figure 2.10: Transition diagram of *NFA* for string *DIRT*-matching (*SFOGCO* automaton) for pattern $P = p_1p_2p_3p_4$, $k = 3$

$q_{00}, q_{10}, q_{20}, \ldots, q_{m0},$
$q_{11}, q_{21}, \ldots, q_{m1},$
$\vdots$
$q_{kk}, \ldots, q_{mk}.$

Construct *SFOGCO* automaton $M = (Q, A, \delta, q_0, F)$ as follows:
$Q = Q^{'} \cup \{r_{ij} : j = 1, 2, \ldots, k,\ i = j-1, j, \ldots, m-2\}$,
$\delta(q, a) = \delta^{'}(q, a)$ for all $q \in Q$, $a \in A \cup \{\varepsilon\}$,
$\delta(q_{ij}, a) = r_{ij}$, $j = 1, 2, \ldots, k$, $i = j-1, j, \ldots, m-2$, if $\delta(q_{i+1,j}, a) = q_{i+2,j}$,
$\delta(r_{ij}, a) = q_{i+2,j+1}$, $j = 1, 2, \ldots, k$, $i = j-1, j, \ldots, m-2$, if $\delta(q_{ij}, a) = q_{i+1,j}$.
□

2.2.4 Approximate string matching - ordered alphabet

We will discuss three variants of approximate string matching corresponding to the three definitions of distances between strings in ordered alphabets: Δ-distance, Γ-distance, and (Δ, Γ)-distance. The notation introduced in the following definition will be used in this Section.

Definition 2.8
Let A be an ordered alphabet, $A = \{a_1, a_2, \ldots, a_{|A|}\}$. We denote following sets in this way:
$a_i^j = \{a_{i-j}, a_{i+j}\}$,

$a_i^{j+} = \{a_{i-1}, a_{i-2}, \ldots, a_{i-j}, a_{i+1}, a_{i+2}, \ldots, a_{i+j}\}$,
$a_i^{j*} = a_i^{j+} \cup \{a_i\}$.

Some elements of these sets may be missing when a_i is close either to the beginning or to the end of ordered alphabet A. □

The definition is presented in Fig. 2.11.

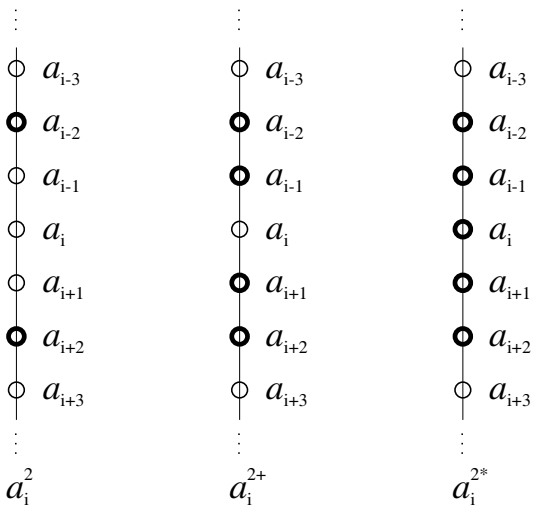


Figure 2.11: Visualisation of $a_i^j, a_i^{j+}, a_i^{j*}$ from Definition 2.8, $j = 2$

2.2.4.1 Δ–distance Let us note that two strings x and y have Δ-distance equal to k if the symbols on the equal positions have maximum Δ-distance equal to k (see Def. 1.18). This type of string matching using Δ-distance is called string Δ-matching (see Def. 1.19).

A model of the algorithm for string Δ-matching ($SFO\Delta CO$ problem) for string $P = p_1p_2p_3p_4$ is shown in Fig. 2.12.

The construction of the $SFO\Delta CO$ nondeterministic finite automaton is based on the composition of $k+1$ copies of the $SFOECO$ automaton with a simple modification. The modification consists in changing the "horizontal" transitions in all copies but the first change is to transitions for all symbols in p_i^*, $i = 2, 3, \ldots, m$. The composition is done by inserting "diagonal" transitions having different "angles" and starting in all non-final states of all copies but the last one. They lead to all following next copies. The inserted transitions represent replace operations. Algorithm 2.9 describes the construction of the $SFO\Delta CO$ automaton in detail.

Algorithm 2.9
Construction of $SFO\Delta CO$ automaton

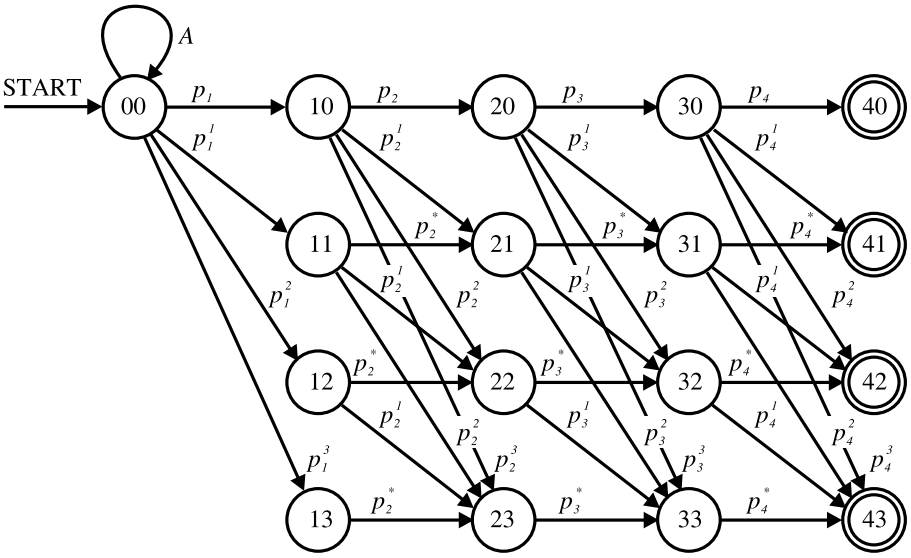


Figure 2.12: Transition diagram of *NFA* for string Δ-matching (*SFO* Δ *CO* problem) for pattern $P = p_1p_2p_3p_4$

Input: Pattern $P = p_1p_2 \ldots p_m$, k, *SFOECO* automaton $M' = (Q', A, \delta', q_0', F')$ for P.
Output: *SFO* Δ *CO* automaton M.
Method:

1. Create a sequence of $k+1$ instances of *SFOECO* automata $M_j' = (Q_j', A, \delta_j', q_{0j}', F_j')$ for $j = 0, 1, 2, \ldots, k$. Let states in Q_j' are $q_{0j}, q_{1j}, \ldots, q_{mj}$.
2. Construct *SFO* Δ *CO* automaton $M = (Q, A, \delta, q_0, F)$ as follows:
   $Q = \bigcup_{j=0}^{k} Q_j'$,
   $\delta(q, a) = \delta_0'(q, a)$ for all $q \in Q_0'$, $a \in A$,
   $\delta(q_{ij}, b) = \delta_j'(q_i, p_{i+1})$ for all $b \in p_{i+1}^*$, $i = 1, 2, \ldots, m-1$, $j = 1, 2, \ldots, k-1$,
   $\delta(q_{ij}, a) = \{q_{i+1,j+1}, q_{i+1,j+2}, \ldots, q_{i+1,k}\}$, for all $i = 0, 1, \ldots, m-1$, $j = 0, 1, 2, \ldots, k-1$, $\quad a \in p_{i+1}^k$,
   $q_0 = q_{00}$,
   $F = \bigcup_{j=0}^{m} F_j'$.
3. Remove all states which are inaccessible from state q_0. □

The *SFO* Δ *CO* automaton has $m(k+1)+1$ states.

2.2.4.2 Γ-distance Let us note that two strings x and y have Γ-distance equal to k if the symbols on the equal positions have Δ-distance less or equal to k and the sum of all these Δ-distances is less or equal to k. The Δ-distance

may be equal to Γ-distance (see Def. 1.18). This type of string matching using Γ-distance is called string Γ-matching (see Def. 1.19).

Model of algorithm for string Γ-matching ($SFO\Gamma CO$ problem) for string $P = p_1p_2p_3p_4$ is shown in Fig. 2.13. Construction of the $SFO\Gamma CO$ nondeterministic finite automaton is based on composition of $k+1$ copies of the $SFOECO$ automaton. The composition is done by insertion of "diagonal" transitions having different "angles" and starting in all non-final states of the all copies but the last one. They are leading to all next copies. The inserted transitions represent replace operations. Algorithm 2.11 describes the construction of the $SFO\Gamma CO$ automaton in detail.

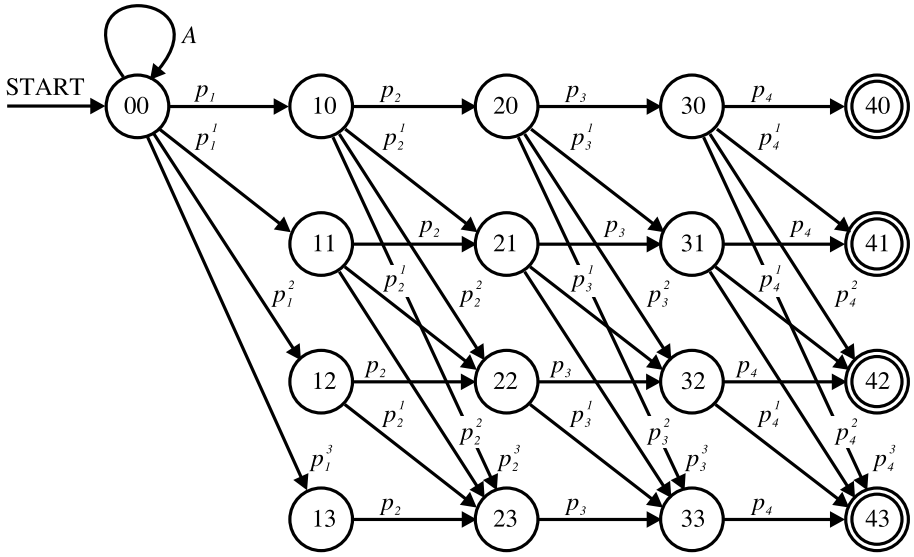


Figure 2.13: Transition diagram of *NFA* for string Γ-matching ($SFO\Gamma CO$ problem) for pattern $P = p_1p_2p_3p_4$, $k = 3$

Algorithm 2.10
Construction of $SFO\Gamma CO$ automaton.
Input: Pattern $P = p_1p_2 \dots p_m$, k, $SFOECO$ automaton $M' = (Q', A, \delta', q_0', F')$ for P.
Output: $SFO\Gamma CO$ automaton M.
Method:

1. Create a sequence of $k+1$ instances of $SFOECO$ automata $M_j' = (Q_j', A, \delta_j', q_{0j}', F_j')$ for $j = 0, 1, 2, \dots, k$. Let states in Q_j' are $q_{0j}, q_{1j}, \dots, q_{mj}$.
2. Construct $SFO\Gamma CO$ automaton $M = (Q, A, \delta, q_0, F)$ as follows:
   $Q = \bigcup_{j=0}^{k} Q_j'$,
   $\delta(q, a) = \delta_j'(q, a)$ for all $q \in Q$, $a \in A$, $j = 0, 1, 2, \dots, k$,

$\delta(q_{ij}, a) = \{q_{i+1,j+1}, q_{i+1,j+2}, \ldots, q_{i+1,k}\}$, for all $i = 0, 1, \ldots, m-1$, $j = 0, 1, 2, \ldots, k-1, \quad a \in p^k_{i+1}$,
$q_0 = q_{00}$,
$F = \bigcup_{j=0}^{m} F'_j$.

3. Remove all states which are inaccessible from state q_0. □

The $SFO\Gamma CO$ automaton has $m(k+1)+1$ states.

2.2.4.3 (Δ, Γ)-distance Let us note that two strings x and y have Γ distance if the symbols on the equal positions have Δ-distance less or equal to l and the sum of these Δ-distances is less or equal to k. The Δ-distance is strictly less than the Γ-distance (see Def. 1.18). This type of string matching using (Δ, Γ) distance is called string (Δ, Γ) matching (see Def. 1.19). A model of the algorithm for string (Δ, Γ) matching ($SFO(\Delta, \Gamma)CO$ problem) for the string $P = p_1p_2p_3p_4$ is shown in Fig. 2.14. Construction of the

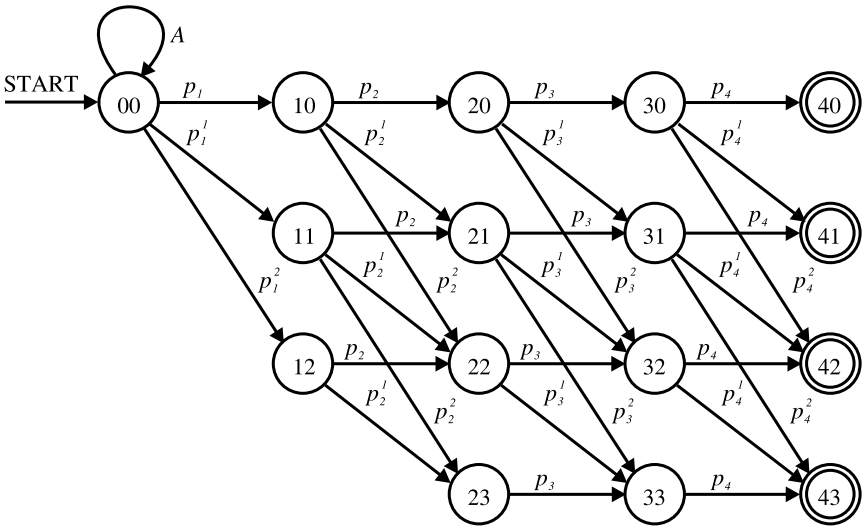


Figure 2.14: Transition diagram of *NFA* for string (Δ, Γ)-matching ($SFO(\Delta, \Gamma)CO$ problem) for pattern $P = p_1p_2p_3p_4$, $l = 2$, $k = 3$

$SFO(\Delta, \Gamma)CO$ nondeterministic finite automaton is similar to the construction of the $SFO\Gamma CO$ automaton. The only difference is that the number of "diagonal" transitions is limited by l. Algorithm 2.11 describes the construction of the $SFO(\Delta, \Gamma)CO$ automaton in detail.

Algorithm 2.11
Construction of the $SFO(\Delta, \Gamma)CO$ automaton.
Input: Pattern $P = p_1p_2 \ldots p_m, k, l$, $SFOECO$ automaton for P.
Output: $SFO(\Delta, \Gamma)CO$ automaton M.

Method: Let $M^{'} = (Q^{'}, A, \delta^{'}, q_0^{'}, F^{'})$ be a *SFOECO* automaton for given pattern P.

1. Create a sequence of $k+1$ instances of *SFOECO* automata $M_j^{'} = (Q_j^{'}, A, \delta_j^{'}, q_{0j}^{'}, F_j^{'})$ for $j = 0, 1, 2, \ldots, k$. Let states in $Q_j^{'}$ be $q_{0j}, q_{1j}, \ldots, q_{mj}$.
2. Construct the $SFO(\Delta, \Gamma)CO$ automaton $M = (Q, A, \delta, q_0, F)$ as follows:
   $Q = \bigcup_{j=0}^{k} Q_j^{'}$,
   $\delta(q, a) = \delta_j^{'}(q, a)$ for all $q \in Q$, $a \in A$, $j = 1, 2, \ldots, m$,
   $\delta(q_{ij}, a) = \{q_{i+1,j+1}, q_{i+1,j+2}, \ldots, q_{i+1,j+l}\}$, for all $i = 0, 1, \ldots, m-1$, $j = 0, 1, 2, \ldots, l-1, \quad a \in p_{i+1}^{l}$,
   $q_0 = q_{00}$,
   $F = \bigcup_{j=0}^{m} F_j^{'}$.
3. Remove all states which are inaccessible from state q_0.

The $SFO(\Delta, \Gamma)CO$ automaton has less than $m(k+1)+1$ states. □

2.2.5 Approximate sequence matching

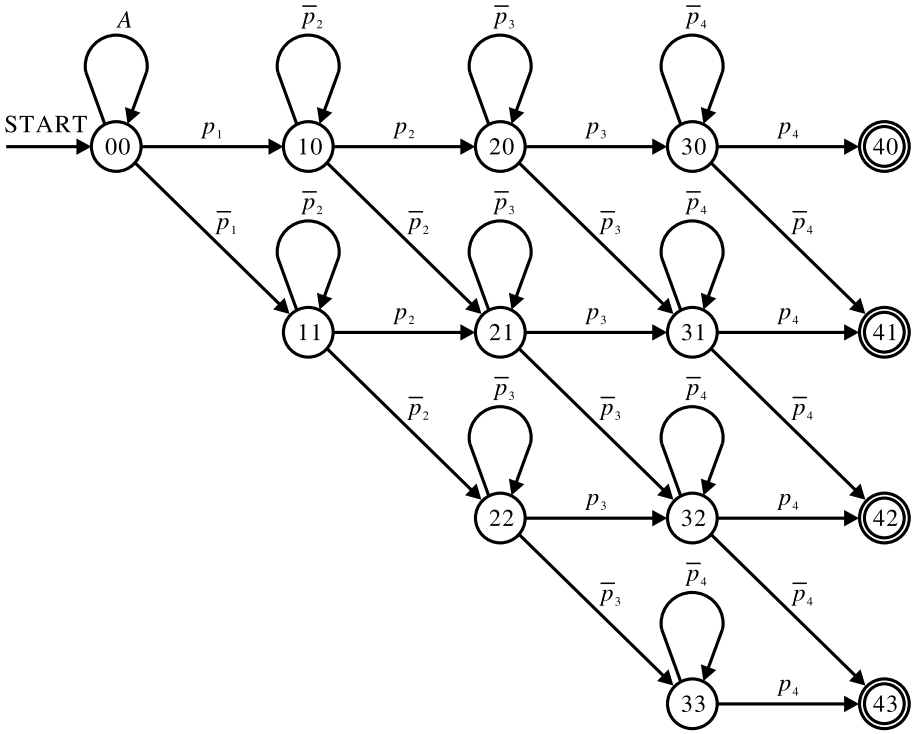


Figure 2.15: Transition diagram of *NFA* for sequence R-matching ($QFORCO$ automaton)

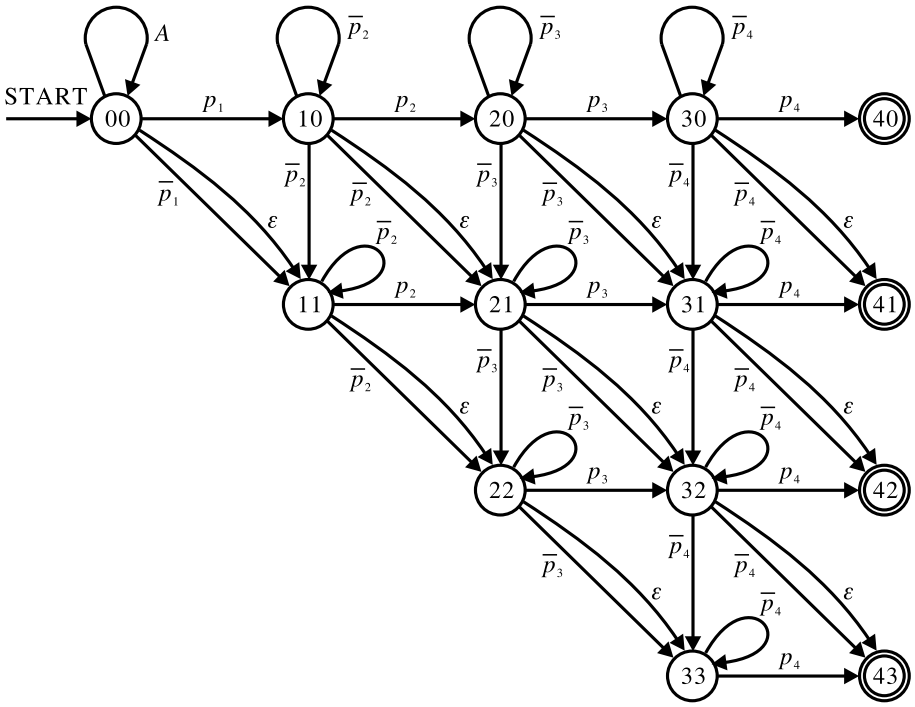


Figure 2.16: Transition diagram of *NFA* for sequence DIR-matching ($QFODCO$ automaton)

Here we discuss six variants of approximate sequence matching in which the following distances are used: Hamming distance, Levenshtein distance, generalized Levenshtein distance, Δ-distance, Γ-distance, and (Δ, Γ)-distance. There are two ways of constructing nondeterministic finite automata for approximate sequence matching:

- The first way is to construct a $QFO?CO$ nondeterministic finite automaton by the corresponding algorithms for approximate string matching, to which we give the $QFOECO$ automaton as the input.
- The second way is to transform a $SFO?CO$ automaton to a $QFO?CO$ automaton by adding self loops for all symbols to all non-initial states that have at least one outgoing transition, as shown in Algorithm 2.12.

Algorithm 2.12
Transformation of an $SFO?CO$ automaton to a $QFO?CO$ automaton.
Input: $SFO?CO$ automaton $M' = (Q', A, \delta', q_0', F')$.
Output: $QFO?CO$ automaton M.
Method: Construct automaton $M = (Q, A, \delta, q_0, F)$ as follows:

1. $Q = Q'$.
2. $\delta(q, a) = \delta'(q', a)$ for each $q \in Q$, $a \in A \cup \{\varepsilon\}$.

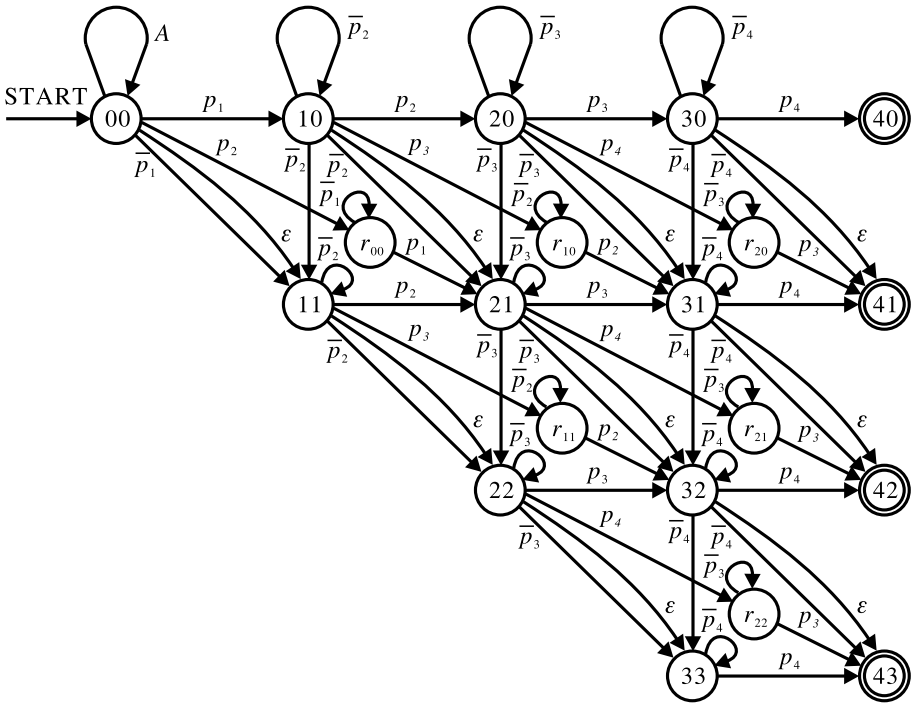


Figure 2.17: Transition diagram of *NFA* for sequence $DIRT$-matching (*QFOGCO* automaton)

3. $\delta(q,a) = \delta'(q',a) \cup \{q\}$ for all such $q \in Q$ that $\delta(q,a) \neq \emptyset$, $a \in \overline{p_i}$, where p_i is the label of the outgoing transition from state q to the next state at the same level.
4. $q_0 = q'_0$.
5. $F = F'$. □

Transition diagrams of the resulting automata are depicted in Figs. 2.15, 2.16 and 2.17 for pattern $P = p_1p_2p_3p_4,\ k = 3$.

2.2.6 Matching of finite and infinite sets of patterns

A model of the algorithm for matching a finite set of patterns is constructed as the union of *NFA* for matching the individual patterns.

As an example we show the model for exact matching of the set of patterns $P = \{p_1p_2p_3, p_4p_5, p_6p_7p_8\}$ (*SFFECO* problem). This is shown in Fig. 2.18. This automaton is in some contexts called dictionary matching automaton. A dictionary is a finite set of strings.

The operation union of nondeterministic finite automata is the general approach for the family of matching of finite set of patterns (??F??? family). Moreover, the way of matching of each individual pattern must be defined.

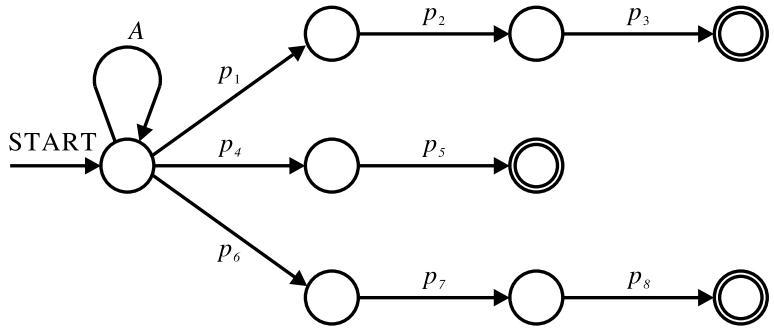


Figure 2.18: Transition diagram of the nondeterministic finite automaton for exact matching of the finite set of strings $P = \{p_1p_2p_3, p_4p_5, p_6p_7p_8\}$ (*SFFECO* automaton)

The next algorithm describes this approach and assumes that the way of matching is fixed (exact, approximate, ...).

Algorithm 2.13
Construction of the ??*F*??? automaton.
Input: A set of patterns with a specification of the way of matching $P = \{P_1(w_1), P_2(w_2), \ldots, P_r(w_r)\}$, where $P_1, P_2, \ldots, P_r$ are patterns and $w_1, w_2, \ldots, w_r$ are specifications of the ways of matching them.
Output: The ??*F*??? automaton.
Method:

1. Construct an *NFA* for each pattern P_i, $1 \leq i \leq r$, with respect to the specification of matching w_i.
2. Create the *NFA* for the language which is the union of all input languages of the automata constructed in step 1. The resulting automaton is the ??*F*??? automaton. □

Example 2.14
Let the input to Algorithm 2.13 be $P = \{abc(SFOECO), def(QFOECO), xyz(SSOECO)\}$. The result of step 1 of Algorithm 2.13 is shown in Fig. 2.19.

The transition diagram of the final $\left\{ \begin{array}{c} S \\ Q \end{array} \right\} \left\{ \begin{array}{c} F \\ S \end{array} \right\} OECO$ automaton is shown in Fig. 2.20. □

The model of the algorithm for matching an infinite set of patterns is based on a finite automaton accepting this set. The infinite set of patterns is in this case defined by a regular expression. Let us present the exact matching of an infinite set of strings (*SFIECO* problem). The construction of the *SFIECO* nondeterministic finite automaton is performed in two steps.

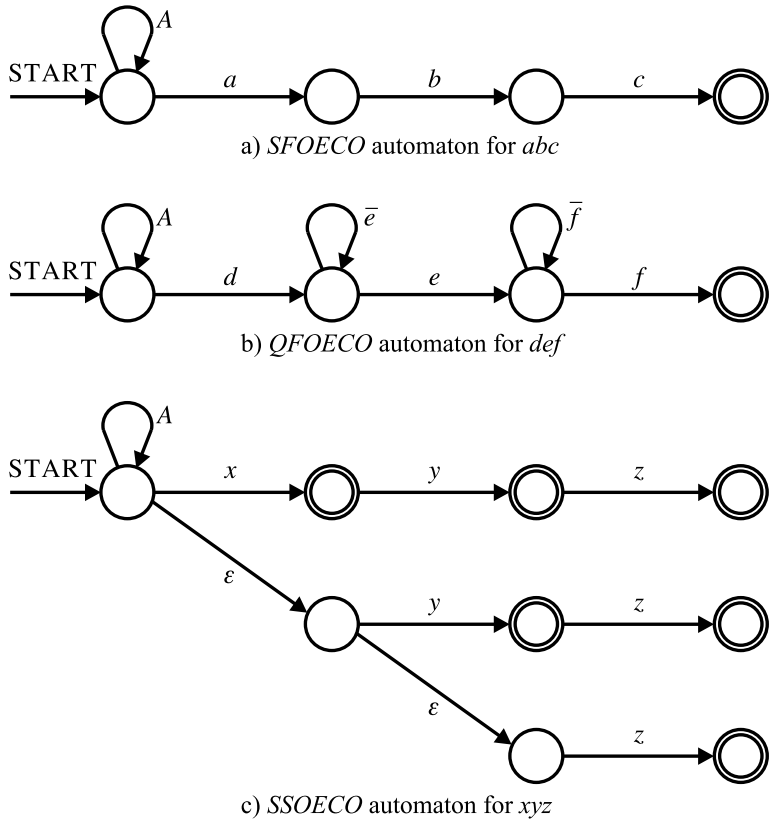


Figure 2.19: Transition diagram of nondeterministic finite automata for individual patterns from Example 2.14

In the first step, a finite automaton accepting the language defined by the given regular expression is constructed. The selfloop in its initial state for all symbols from the alphabet is added in the second step. Algorithm 2.15 describes the construction of the *SFIECO* automaton in detail.

Algorithm 2.15
Construction of the *SFIECO* automaton.
Input: Regular expression R describing a set of strings over alphabet A.
Output: *SFIECO* automaton M.
Method:

1. Construct finite automaton $M' = (Q, A, \delta', q_0, F)$ such that $L(M') = h(R)$, where $h(R)$ is the value of the regular expression R.
2. Construct nondeterministic finite automaton $M = (Q, A, \delta, q_0, F)$, where $\delta(q, a) = \delta'(q, a)$ for all $q \in Q \backslash \{q_0\}$, $a \in A \cup \{\varepsilon\}$,
   $\delta(q_0, a) = \delta'(q_0, a) \cup \{q_0\}$ for all $a \in A$. □

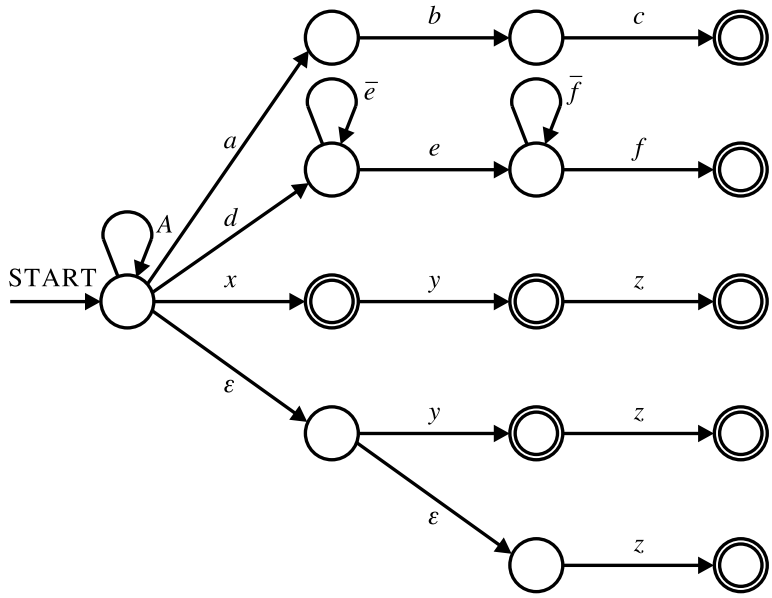


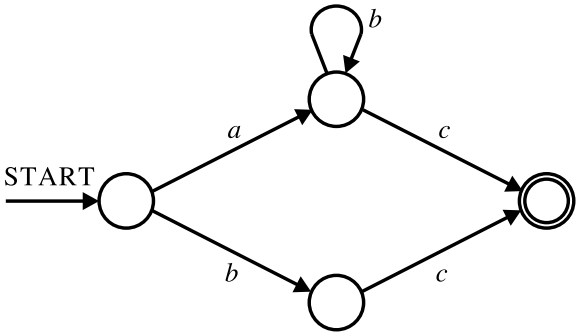
Figure 2.20: Transition diagram of the resulting finite automaton for the set of patterns P from Example 2.14

Example 2.16
Let the regular expression $R = ab^*c+bc$ be given over alphabet $A = \{a, b, c\}$. The result of step 1 of Algorithm 2.15 is shown in Fig. 2.21a. The final result of Algorithm 2.15 is shown in Fig. 2.21b. □

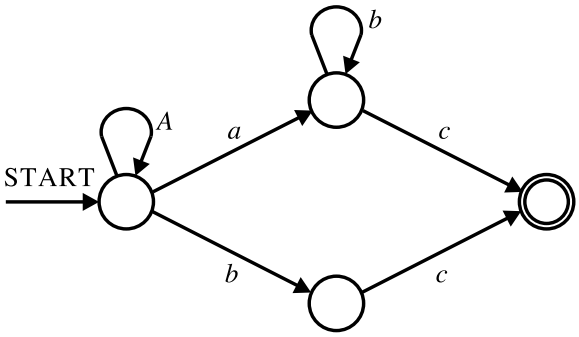
The construction of the *QFIECO* nondeterministic finite automaton starts, as for the *SFIECO* automaton, by the construction of the finite automaton accepting the language defined by the given regular expression. There are added selfloops for all symbols from the alphabet to the initial state and to all states having outgoing transitions for more than one symbol. The selfloops to all states having only one outgoing transition (for symbol a) are added for all symbols but symbol a. Algorithm 2.17 describes the construction of the *QFIECO* automaton in detail.

Algorithm 2.17
Construction of the *QFIECO* automaton.
Input: Regular expression R describing a set of strings over alphabet A.
Output: The *QFIECO* automaton.
Method:

1. Construct finite automaton $M^{'} = (Q, A, \delta^{'}, q_0, F)$ such that $L(M') = h(R)$, where $h(R)$ is the value of the regular expression R.
2. Construct nondeterministic finite automaton $M = (Q, A, \delta, q_0, F)$ where $\delta(q, a) = \delta^{'}(q, a)$ for all $q \in Q$, $a \in A$,



a) Finite automaton for $R=ab^*c+bc$



b) *SFIECO* automaton

Figure 2.21: Transition diagram of the finite automata of Example 2.16

$\delta(q,a) = \delta'(q,a) \cup \{q\}$ for all $q \in Q$, such that $\delta(q,a) \neq \emptyset$, $a \in A \setminus \{a\}$, where there is an outgoing transition from state q for symbol a only, $\delta(q,a) = \delta'(q,a) \cup \{q\}$ for all $q \in Q$ and $a \in A$ such that $\delta(q,a) \neq \emptyset$ for more than one symbol $a \in A$. □

Example 2.18
Let a regular expression be $R = ab^*c + bc$ over alphabet $A = \{a, b, c\}$. The result of Algorithm 2.17 is the automaton having transition diagram depicted in Fig. 2.22. □

2.2.7 Pattern matching with "don't care" symbols

Let us recall that the "don't care" symbol $\circ$ is the symbol matching any other symbol from alphabet A including itself. The transition diagram of the nondeterministic finite automaton for exact string matching with the don't care symbol (*SFOEDO* problem) for pattern $P = p_1 p_2 \circ p_4$ is shown in Fig. 2.23.

An interesting point of this automaton is the transition from state 2 to state 3 corresponding to the don't care symbol. This is in fact a set of

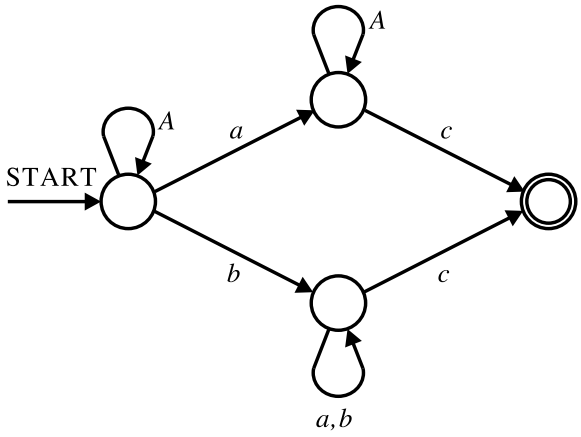


Figure 2.22: Transition diagram of the resulting *QFIECO* automaton from Example 2.18

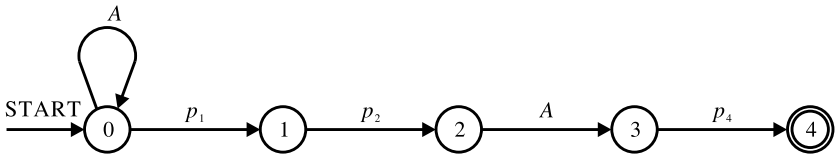


Figure 2.23: Transition diagram of the nondeterministic finite automaton for exact string matching with the don't care symbol ($SFOEDO$ automaton) for pattern $P = p_1 p_2 \circ p_4$

transitions for all symbols of alphabet A. The rest of the automaton is the same as for the *SFOECO* automaton.

The transition diagram of the nondeterministic finite automaton for exact sequence matching with the don't care symbol ($QFOEDO$ problem) for the pattern $P = p_1 p_2 \circ p_4$ is shown in Fig. 2.24.

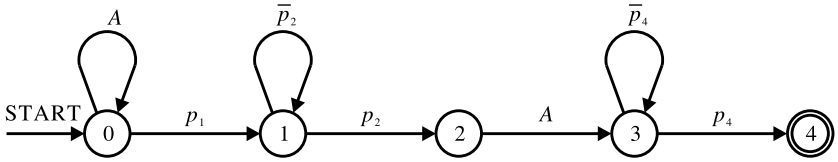


Figure 2.24: Transition diagram of the nondeterministic finite automaton for exact sequence matching with don't care symbol ($QFOEDO$ automaton) for pattern $P = p_1 p_2 \circ p_4$

The transition for the don't care symbol is the same as for string matching. However, the rest of the automaton is a slightly changed *QFOECO* automaton. The selfloop in state 2 is missing. The reason for this is that the symbol following symbol p_2 is always the third element of the given sequence, because we "do not care" about.

The construction of automata for other problems with don't care symbols uses the principle of insertion of the sets of transitions for all symbols of the alphabet to the place corresponding to the positions of the don't care symbols.

Example 2.19
Let pattern $P = p_1 p_2 \circ p_4$ be given. We construct the *SFORDO* automaton (approximate string matching of one full pattern using Hamming distance) for Hamming distance $k = 3$. The transition diagram of the *SFORDO* automaton is depicted in Fig. 2.25. Let us note that transitions labelled by $\overline{A}$ refer to a transition for an empty set of symbols, and may be removed.

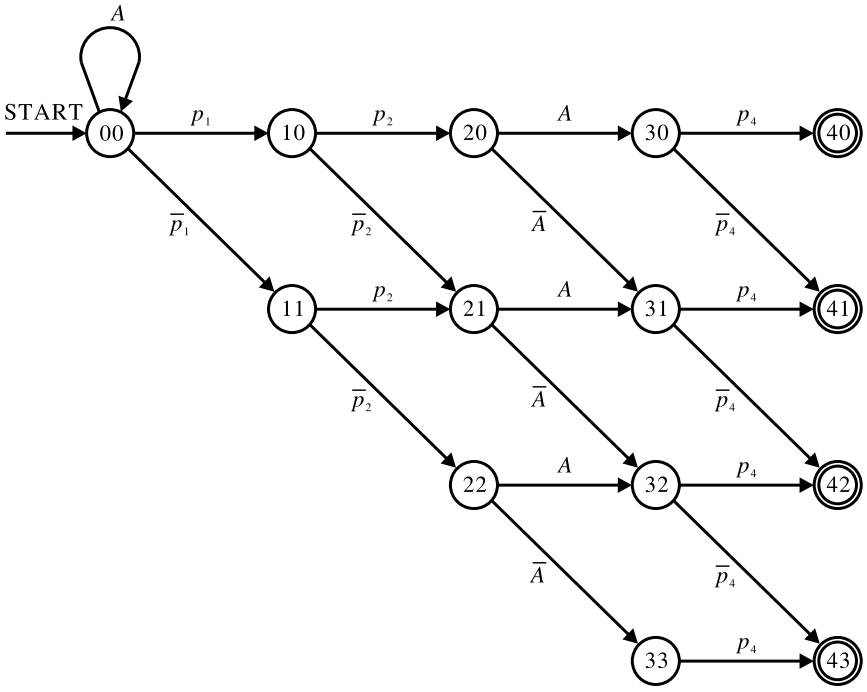


Figure 2.25: Transition diagram of the nondeterministic finite automaton for the *SFORDO* problem for pattern $P = p_1 p_2 \circ p_4$ (transitions labelled by $\overline{A}$ may be removed)

2.2.8 Matching a sequence of patterns

Matching a sequence of patterns is defined by Definition 1.16. The nondeterministic finite automaton for matching a sequence of patterns is constructed by making a "cascade" of automata for patterns in the given sequence. The construction of the nondeterministic finite automaton for a sequence of patterns starts by constructing finite automata for matching all elements of the sequence. The next operation (making a "cascade") is the insertion of ε-transitions from all final states of the automaton in the sequence to the initial state of the next automaton in the sequence, if it exists. The following algorithm describes this construction.

Algorithm 2.20
Construction of the $?????S$ automaton.
Input: Sequence of patterns $P_1(w_1), P_2(w_2), \ldots, P_s(w_s)$, where $P_1, P_2, \ldots, P_s$ are patterns and $w_1, w_2, \ldots, w_S$ are specifications of their matching.
Output: $?????S$ automaton.
Method:

1. Construct a *NFA* $M_i = (Q_i, A_i, \delta_i, q_{0i}, F_i)$ for each pattern $P_i(w_i)$, $1 \leq i \leq s, s > 1$, with respect to the specification w_i.
2. Create automaton $M = (Q, A, \delta, q_0, F)$ as a cascade of automata in this way:
   $Q = \bigcup_{i=1}^{s} Q_i$,
   $A = \bigcup_{i=1}^{s} A_i$,
   $\delta(q, a) = \delta_i(q, a)$ for all $q \in Q_i$, $a \in A_i \cup \{\varepsilon\}$, $i = 1, 2, \ldots, s$,
   $\delta(q, \varepsilon) = (q_{0,i+1})$, for all $q \in F_i$, $1 \leq i \leq s-1$,
   $q_0 = q_{01}$,
   $F = F_s$. □

The main point of Algorithm 2.20 is the insertion of ε-transitions from all final states of automaton M_i to the initial state of automaton M_{i+1}.

Example 2.21
Let the input to Algorithm 2.20 be the sequence *abc, def, xyz*. The resulting *SFOECS* automaton is shown in Fig. 2.26. □

2.3 Some deterministic pattern matching automata

In this Section we will show some deterministic finite automata obtained by determinising the of nondeterministic finite automata of the previous Section. A deterministic pattern matching automaton needs at most n steps for pattern matching in the text $T = t_1 t_2 \ldots t_n$. This means that the time complexity of searching is linear ($\mathcal{O}(n)$) for all problems in the classification described in Section 1.2.

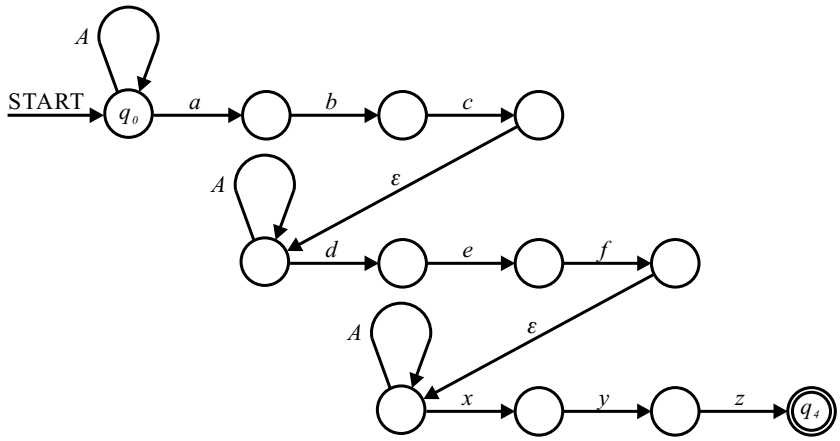


Figure 2.26: Transition diagram of the nondeterministic finite automaton for matching the sequence of patterns $P = abc(SFOECO)$, $def(SFOECO)$, $xyz(SFOECO)$ (*SFOECS* automaton)

On the other hand, the use of deterministic finite automata has two drawbacks:

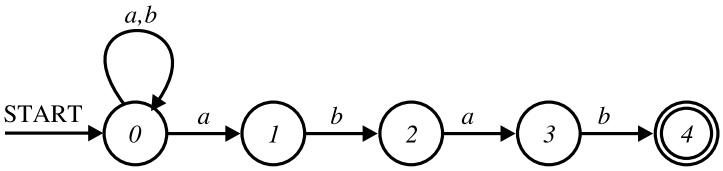
1. The size of the pattern matching automaton depends on the cardinality of the alphabet. Therefore this approach is suitable primarily for small alphabets.
2. The number of states of a deterministic pattern matching automaton can be much greater than the number of states of its nondeterministic equivalent.

These drawbacks, the time and space complexity of the construction of a deterministic finite automaton, are the Price we have to pay for fast searching. Several methods for simulating the original nondeterministic pattern matching automata have been designed to overcome these drawbacks. They will be discussed in the following Chapters.

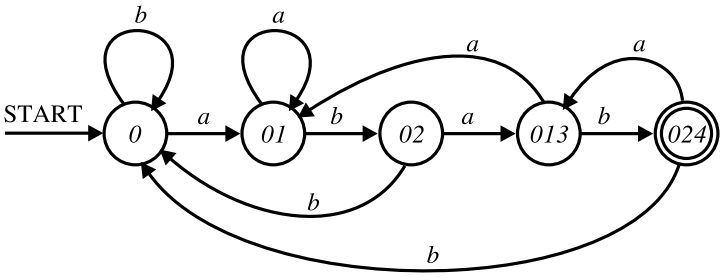
2.3.1 String matching

The deterministic *SFOECO* finite automaton for the pattern $P = p_1p_2 \dots p_m$ is the result of determinisation of the nondeterministic *SFOECO* automaton.

Example 2.22
Let us have pattern $P = abab$ over alphabet $A = \{a, b\}$. A transition diagram of the *SFOECO*($abab$) automaton and its deterministic equivalent are depicted in Fig. 2.27. Transition tables of both automata are shown in Table 2.1. The deterministic *SFOECO* automaton is a complete automaton and has just $m + 1$ states for a pattern of length m. The number of steps



a) Nondeterministic *SFOECO(abab)* automaton



b) Deterministic *SFOECO(abab)* automaton

Figure 2.27: Transition diagrams of the nondeterministic and deterministic *SFOECO* automata for pattern $P = abab$ from Example 2.22

(number of transitions) made during pattern matching in a text of length n is just n. □

2.3.2 Matching of a finite set of patterns

The deterministic *SFFECO* automaton for matching a set of patterns $S = \{P_1, P_2, \ldots, P_s\}$ is the result of the determinisation of nondeterministic *SFFECO* automaton.

Example 2.23
Let us have set of patterns $S = \{ab, bb, babb\}$ over alphabet $A = \{a, b\}$. Transition diagram of the *SFFECO*(S) automaton and its deterministic equivalent are depicted in Fig. 2.28. Transition tables of both automata are shown in Table 2.2. □

The deterministic pattern matching automaton for a finite set of patterns has less than $|S| + 1$ states, where $|S| = \sum_{i=1}^{s} |P_i|$ for $P = \{P_1, P_2, \ldots, P_s\}$.

The maximum number of states (equal to $|S|+1$) is reached in the case when no two patterns have common prefix. In Example 2.23 holds that $|S|+1 = 9$ and patterns bb and $babb$ have common prefix b. Therefore the number of states is 8 which is less than 9.

	a	b
0	$0, 1$	0
1		2
2	3	
3		4
4		

a) Nondeterministic $SFOECO(abab)$ automaton

	a	b
0	01	0
01	01	02
02	013	0
013	01	024
024	013	0

b) Deterministic $SFOECO(abab)$ automaton

Table 2.1: Transition tables of $SFOECO(abab)$ automata from Example 2.22

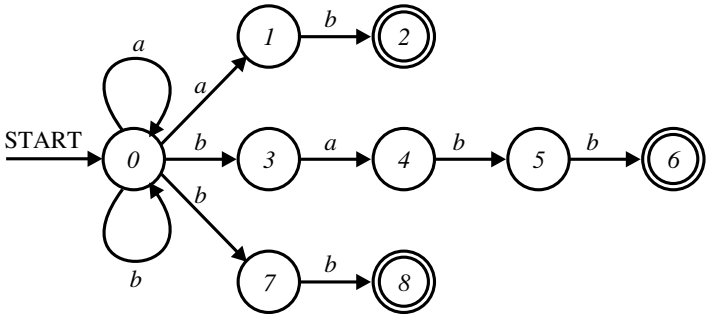
	a	b
0	$0, 1$	$0, 3, 7$
1		2
2		
3	4	
4		5
5		6
6		
7		8
8		

a) Nondeterministic $SFFECO$ ($\{ab, bb, babb\}$) automaton

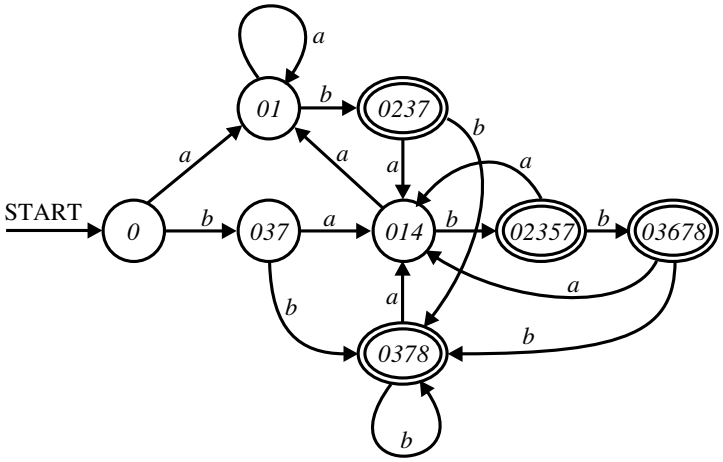
	a	b
0	01	037
01	01	0237
037	014	0378
0237	014	0378
014	01	02357
0378	014	0378
02357	014	03678
03678	014	0378

b) Deterministic $SFFECO$ ($\{ab, bb, babb\}$) automaton

Table 2.2: Transition tables of $SFFECO(\{ab, bb, babb\})$ automata from Example 2.23



a) Nondeterministic *SFFECO* ({*ab,bb,babb*}) automaton



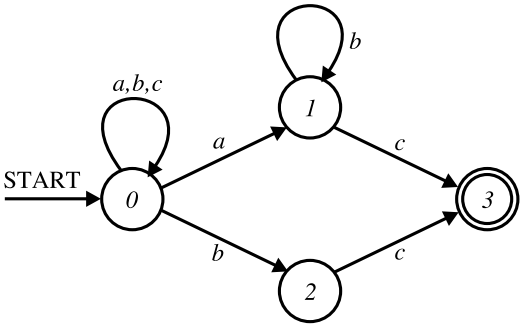
b) Deterministic *SFFECO* ({*ab,bb,babb*}) automaton

Figure 2.28: Transition diagrams of the nondeterministic and deterministic *SFFECO* automata for $S = (\{ab, bb, babb\})$ from Example 2.23

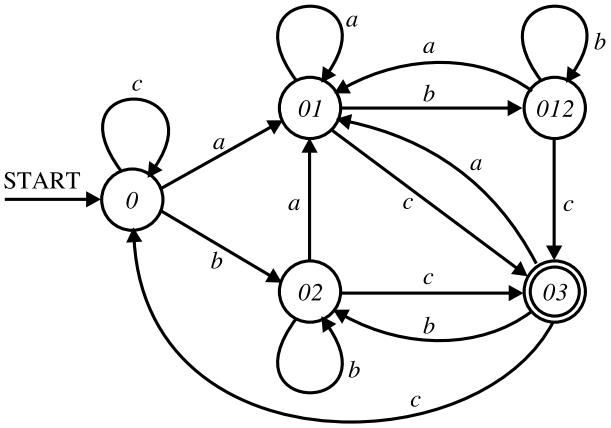
2.3.3 Regular expression matching

The deterministic finite automaton for matching an infinite set of patterns is the result of the determinisation of *SFIECO* automaton.

Example 2.24
Let us have regular expression $R = ab^*c + bc$ over alphabet $A = \{a, b, c\}$ (see also Example 2.16). Transition diagram of the *SFIECO*(R) automaton and its deterministic equivalent are shown depicted in Fig. 2.29. Transition



a) Nondeterministic *SFIECO*(ab^*c+bc) automaton



b) Deterministic *SFIECO*(ab^*c+bc) automaton

Figure 2.29: Transition diagrams of the nondeterministic and deterministic *SFIECO* automata for $R = ab^*c + bc$ from Example 2.24

tables of both automata are shown in Table 2.3. □

The space complexity of matching of regular expression can vary from linear to exponential. The example of linear space complexity is matching of regular expression describing language containing one string. The exponential space complexity is reached for example for expression:

$$R = a(a + b)^{m-1}$$

	a	b	c
0	$0,1$	$0,2$	0
1		1	3
2			3
3			

a) Nondeterministic *SFIECO* $(ab^*c + bc)$ automaton

	a	b	c
0	01	02	0
01	01	012	03
02	01	02	03
012	01	012	03
03	01	02	0

b) Deterministic *SFIECO* $(ab^*c + bc)$ automaton

Table 2.3: Transition tables of *SFIECO*$(ab^*c + bc)$ automata from Example 2.24

	a	b
0	$0,1$	0
1	2	2
2	3	3
3		

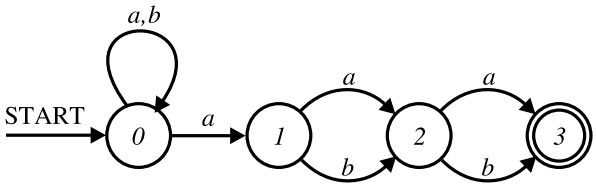
a) Nondeterministic *SFIECO* $(a(a + b)(a + b))$ automaton

	a	b
0	01	0
01	012	02
012	0123	023
0123	0123	023
02	013	03
023	013	03
013	012	02
03	01	0

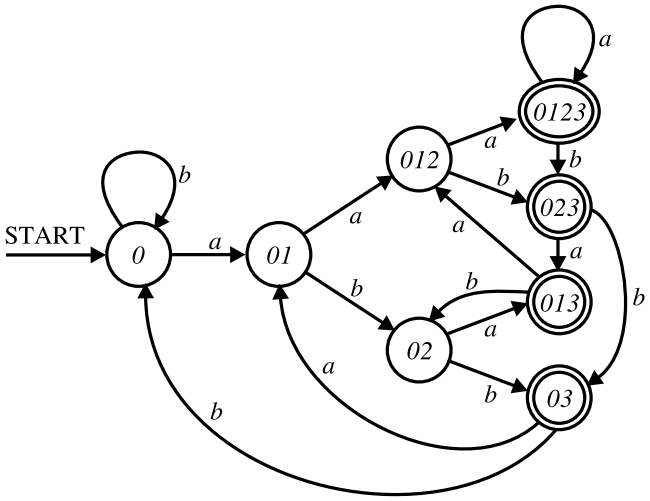
b) Deterministic *SFIECO* $(a(a + b)(a + b))$ automaton

Table 2.4: Transition tables of *SFIECO*$(a(a + b)(a + b))$ automata from Example 2.25

Example 2.25
Let us show example of deterministic regular expression matching automaton for $R = a(a + b)(a + b)$ over alphabet $A = \{a, b\}$. Transition diagram of the *SFIECO*$(a(a + b)(a + b))$ automaton and its deterministic equivalent are depicted in Fig. 2.30. Transition tables of both automata are shown in Table 2.4. We can see that the resulting deterministic automaton has $2^3 = 8$ states. □



a) Nondeterministic *SFIECO*(*a*(*a*+*b*)(*a*+*b*)) automaton



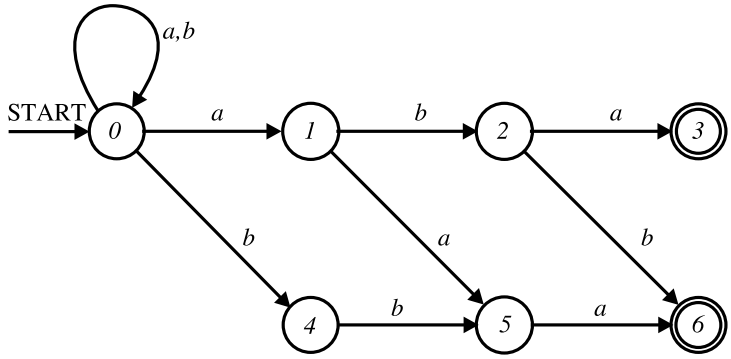
b) Deterministic *SFIECO*(*a*(*a*+*b*)(*a*+*b*)) automaton

Figure 2.30: Transition diagrams of the nondeterministic and deterministic *SFIECO* automata for $R = a(a + b)(a + b)$ from Example 2.25

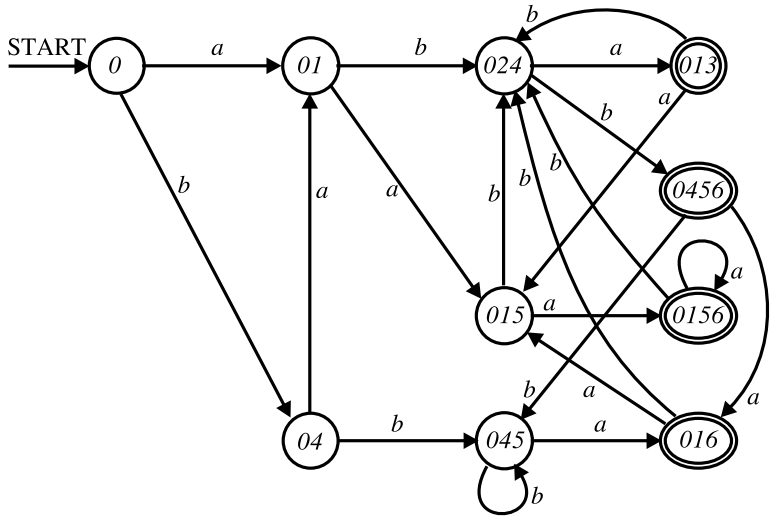
2.3.4 Approximate string matching – Hamming distance

The deterministic finite automaton for approximate string matching using Hamming distance is the result of the determinisation of $SFORCO$ automaton.

Example 2.26
Let us have pattern $P = aba$ over alphabet $A = \{a, b\}$ and Hamming distance $k = 1$. Transition diagram of the $SFORCO(aba, 1)$ automaton and its deterministic equivalent are depicted in Fig. 2.31. Transition tables of both automata are shown in Table 2.5. □



a) Nondeterministic *SFORCO*(*aba,1*) automaton



b) Deterministic *SFORCO*(*aba,1*) automaton

Figure 2.31: Transition diagrams of the nondeterministic and deterministic $SFORCO(aba, 1)$ automata from Example 2.26

	a	b
0	0,1	0,4
1	5	2
2	3	6
3	$\emptyset$	$\emptyset$
4	$\emptyset$	5
5	6	$\emptyset$
6	$\emptyset$	$\emptyset$

	a	b
0	01	04
01	015	024
024	013	0456
013	015	024
04	01	045
015	0156	024
045	016	045
0456	016	045
0156	0156	024
016	015	024

Table 2.5: Transition tables of deterministic and nondeterministic $SFORCO(aba, 1)$ automata from Example 2.26

2.3.5 Approximate string matching – Levenshtein distance

The deterministic finite automaton for approximate string matching using Levenshtein distance is the result of the determinisation of *SFODCO* automaton.

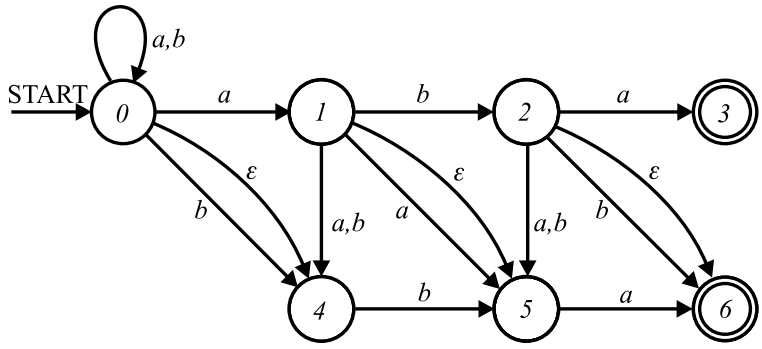
Example 2.27
Let us have pattern $P = aba$ over alphabet $A = \{a, b\}$ and Levenshtein distance $k = 1$. Transition diagram of the $SFODCO(aba, 1)$ automaton and its deterministic equivalent are depicted in Fig. 2.32. Transition tables of both automata are shown in Table 2.6. □

2.4 The state complexity of the deterministic pattern matching automata

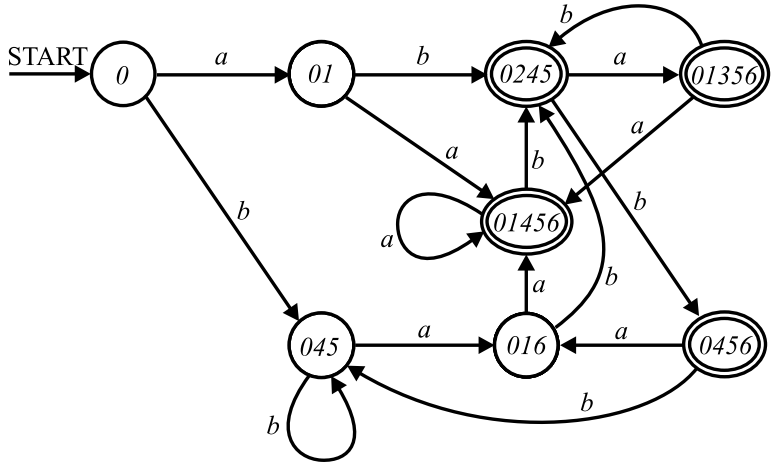
The states of a *SFFECO* automaton (dictionary matching automaton, finite set of strings automaton) correspond to the prefixes of strings in a finite set U. To formalize this automaton, mapping h_U is defined for each language U as follows:

$$h_U(v) = \text{the longest suffix of } v \text{ that belongs to } \mathit{Pref}(U),$$

for each $v \in A^*$.



a) Nondeterministic *SFODCO*(*aba,1*) automaton



b) Deterministic *SFODCO*(*aba,1*) automaton

Figure 2.32: Transition diagrams of the nondeterministic and deterministic $SFODCO(aba, 1)$ automata from Example 2.27

Definition 2.28
Strings v and w are denoted by uw^{-1} and $v^{-1}u$ when $u = vw$.

The following Theorem 2.29 adopted from [CH97b] is necessary for dictionary matching automata construction.

Theorem 2.29
Let $U \subset A^*$. Then

1. for each $v \in A^*$ $v \in A^*U$ iff $h_U(v) \in A^*U$,
2. $h_U(\varepsilon) = \varepsilon$,
3. for each $v \in A^*$, $a \in A$ $h_U(va) = h_U(h_U(v)a)$.

	a	b	ε
0	0,1	0,4	4
1	4,5	2,4	5
2	3,5	5,6	6
3			
4		5	
5	6		
6			

	a	b
0	01	045
01	01456	0245
045	016	045
01456	01456	0245
0245	01356	0456
01356	01456	0245
0456	016	045
016	01456	0245

Table 2.6: Transition tables of $SFODCO(aba, 1)$ automata from Example 2.27

Proof
If $v \in A^*U$, then v is in the form wu, where $w \in A^*$ and $u \in U$. By the definition of h_U, u is necessarily a suffix of $h_U(v)$; therefore $h_U(v) \in A^*U$. Conversely, if $h_U(v) \in A^*U$, we have also $v \in A^*U$, because $h_U(v)$ is a suffix of v. Which proves (1).

Property (2) clearly holds.

It remains to prove (3). Both words $h_U(va)$ and $h_U(v)a$ are suffixes of va, and therefore one of them is a suffix of the other. Then two cases are distinguished according to which word is a suffix of the other.

First case: $h_U(v)a$ is a proper suffix of $h_U(va)$ (hence $h_U(va) \neq \varepsilon$). Consider the word w defined by $w = h_U(va)a^{-1}$. Thus we have: $h_U(v)$ is a proper suffix of w, w is a suffix of v, and $w \in \mathit{Pref}(U)$. Since w is a suffix of v that belongs to $\mathit{Pref}(U)$, but strictly longer than $h_U(v)$, there is a contradiction in the maximality of $|h_U(v)|$, so this case is impossible.

Second case: $h_U(va)$ is a suffix of $h_U(v)a$. Then, $h_U(va)$ is a suffix of $h_U(h_U(v)a)$. Since $h_U(v)a$ is a suffix of va, $h_U(h_U(v)a)$ is a suffix of $h_U(va)$. Both properties imply $h_U(va) = h_U(h_U(v)a)$ and the expected result follows. □

Now the dictionary matching automaton can be constructed according to Theorem 2.30, borrowed from [CH97b].

Theorem 2.30
Let X be a finite language. Then the automaton $M = (Q, A, q_0, \delta, F)$, where $Q = \{q_x \mid x \in \mathit{Pref}(X)\}$, $q_0 = q_\varepsilon$, $\delta(q_p, a) = q_{h_X(pa)}$, $p \in \mathit{Pref}(X)$, $a \in A$, $F = \{q_x \mid x \in \mathit{Pref}(X) \cap A^*X\}$, recognizes the language A^*X. This automaton is deterministic and complete.

Proof
Let $v \in A^*$. It follows from properties (2) and (3) of Theorem 2.29 that after reading v the automaton will be in the state $q_{h_X(v)}$. If $v \in A^*X$, it must hold $h_X(v) \in A^*X$ from (1) of Theorem 2.29; which shows that $q_{h_X(v)}$ is a final state, and finally that v is recognized by the automaton.

Conversely, if v is recognized by the automaton, we have $h_X(v) \in A^*X$ by definition of the automaton. This implies that $v \in A^*X$ from (1) of Theorem 2.29 again. □

Transition diagram of an example of dictionary matching automaton is shown in Figure 2.33.

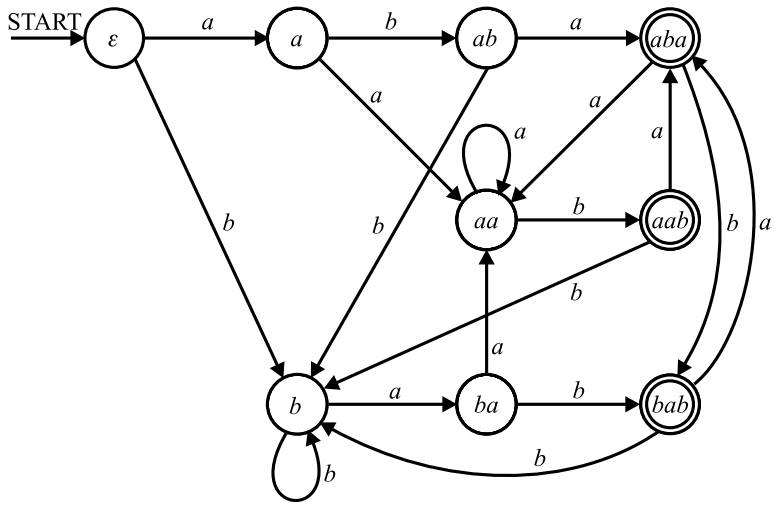


Figure 2.33: Transition diagram of dictionary matching automaton for language $\{aba, aab, bab\}$

2.4.1 Construction of a dictionary matching automaton

The first method of dictionary matching automata construction follows directly from Theorem 2.30. But, as will be shown in this section, dictionary matching automata can be build using standard algorithms. This method is described in Algorithm 2.31.

Algorithm 2.31
Construction of a dictionary matching automaton for a given finite language.

Input: Finite language X.
Output: Dictionary matching automaton accepting language A^*X.
Method:

1. Create a tree-like finite automaton accepting language X (using Algorithm 2.32),
2. Add a selfloop $\delta(q_0, a) = \delta(q_0, a) \cup \{q_0\}$ for each $a \in A$ to M.
3. Using the subset construction (see Algorithm 1.40) make a deterministic dictionary matching automaton accepting language A^*X.

□

Algorithm 2.32
Construction of a deterministic automaton accepting the set of strings.
Input: Finite language X.
Output: Deterministic finite automaton $M = (Q, A, \delta, q_0, F)$ accepting language X.
Method:

1. $Q = \{q_x \mid x \in \mathit{Pref}(X)\}$,
2. $q_0 = q_\varepsilon$,
3. $\delta(q_p, a) = \begin{cases} q_{pa} & \text{in case when } pa \in \mathit{Pref}(X), \\ \mathit{undefined} & \text{otherwise,} \end{cases}$
4. $F = \{q_p \mid p \in X\}$. □

Example 2.33
Let us create a dictionary matching automaton for language $\{aba, aab, bab\}$ to illustrate this algorithm. The outcome from the step (2) of Algorithm 2.31 is shown in Figure 2.34 and the result of the whole algorithm is shown in Figure 2.35. □

What is the result of this algorithm? As shown in Theorem 2.34, automata created according to Theorem 2.30 are equivalent to automata created according to Algorithm 2.31.

Theorem 2.34
Given finite language X, finite automaton $M_1 = (Q_1, A, \delta_1, \{q_\varepsilon\}, F_1)$ accepting language A^*X created by Algorithm 2.31 is equivalent to finite automaton $M_2 = (Q_2, A, \delta_2, q_\varepsilon, F_2)$ accepting the same language but created by Theorem 2.30.

Proof
The first step is to show that after reading a string w automaton M_1 will be in state $q = \delta_1^*(\{q_\varepsilon\}, w) = \{q_x \mid x \in \mathit{Suff}(w) \cap \mathit{Pref}(X)\}$. Let us remind the

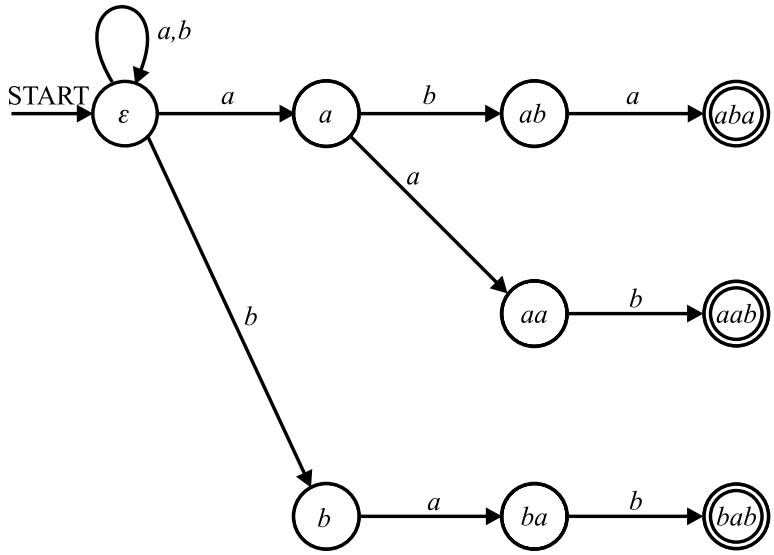


Figure 2.34: Transition diagram of nondeterministic dictionary matching automaton for the language $\{aba, aab, bab\}$

reader that deterministic automaton M_1 was created from nondeterministic automaton $M_1' = (Q_1', A, \delta_1', q_\varepsilon, F_1')$ by the subset construction. As follows from the subset construction algorithm, after reading string w automaton M_1 will be in state $q = \delta_1^*(\{q_\varepsilon\}, w) = {\delta_1'}^*(q_\varepsilon, w)$. Thus it is enough to show that ${\delta_1'}^*(q_\varepsilon, w) = \{q_x \mid x \in \mathit{Suff}(w) \cap \mathit{Pref}(X)\}$. Given string $v \in \mathit{Suff}(w) \cap \mathit{Pref}(X)$, w can be written as uv, $u \in A^*$. Thus, there is a sequence of moves $(q_\varepsilon, uv) \vdash_{M_1'}^{|u|} (q_\varepsilon, v) \vdash_{M_1'}^{|v|} (q_v, \varepsilon)$, so $q_v \in {\delta_1'}^*(q_\varepsilon, w)$. Conversely, consider $q_v \in {\delta_1'}^*(q_\varepsilon, w)$. Since M_1' can read arbitrarily long words only by using the selfloop for all $a \in A$ in the initial state, the sequence of moves must be as follows $(q_\varepsilon, w) \vdash_{M_1'}^* (q_\varepsilon, v) \vdash_{M_1'}^{|v|} (q_v, \varepsilon)$. Consequently $v \in \mathit{Suff}(w) \cap \mathit{Pref}(X)$.

Since the set of states $Q_1 \subseteq \mathcal{P}(\{q_x \mid x \in \mathit{Pref}(X)\})$, it is possible to define an isomorphism $f|_{Q_1} : \mathcal{P}(\{q_x \mid x \in \mathit{Pref}(X)\}) \to \{q_x \mid x \in \mathit{Pref}(X)\}$ as follows:

$$f(q) = p_w \in q, |w| \text{ is maximal.}$$

Now it is necessary to show that

1. $\forall q_1, q_2 \in Q_1,\ q_1 \neq q_2 \Rightarrow f(q_1) \neq f(q_2)$,
2. $\forall p \in Q_2\ \exists q \in Q_1,\ f(q) = p$,
3. $f(q_0^1) = q_0^2$
4. $f(\delta_1(q, a)) = \delta_2(f(q), a)$,
5. $f(F_1) = F_2$.

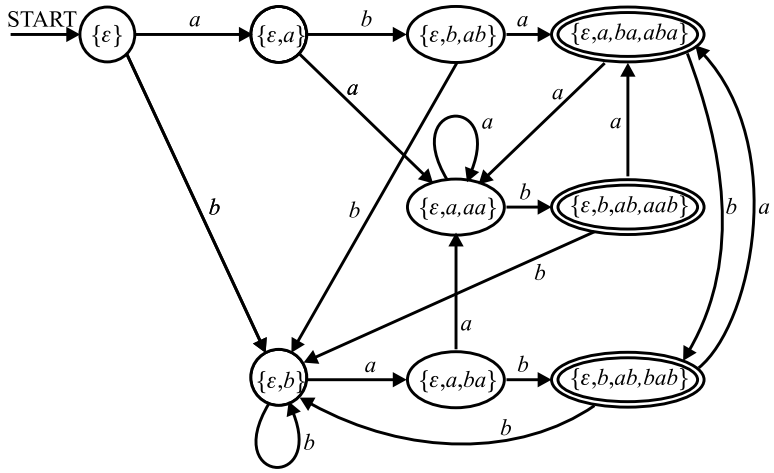


Figure 2.35: Transition diagram of deterministic dictionary matching automaton for language $\{aba, aab, bab\}$ created from nondeterministic one by the subset construction

Let us suppose $p = \delta_1^*(\{q_\varepsilon\}, u)$, $q = \delta_1^*(\{q_\varepsilon\}, v)$, $p \neq q$, and $f(p) = f(q)$. But from the definition of f, it must hold $p = \{q_x |\ x \in \mathit{Suff}(u) \cap \mathit{Pref}(X)\} = \{q_x \mid q_y = f(p), x \in \mathit{Suff}(y) \cap \mathit{Pref}(X)\}$ and $q = \{q_x \mid x \in \mathit{Suff}(v) \cap \mathit{Pref}(X)\} = \{q_x \mid q_y = f(q), x \in \mathit{Suff}(y) \cap \mathit{Pref}(X)\}$, which implies $p = q$. This is the contradiction, so it proves (1).

Property (2) holds because for all $u \in \mathit{Pref}(X)$, $q_u \in \delta_1^*(\{q_\varepsilon\}, u) = \{q_x \mid x \in \mathit{Suff}(u) \cap \mathit{Pref}(X)\}$. Thus $f(\delta_1^*(\{\varepsilon\}, u)) = q_u$.

Property (3) clearly holds.

It will be shown that $f(\delta_1^*(\{q_\varepsilon\}, u)) = \delta_2^*(q_\varepsilon, u)$ which implies the property (4). It is known that $f(\delta_1^*(\{q_\varepsilon\}, u)) = f(\{q_x \mid x \in \mathit{Suff}(u) \cap \mathit{Pref}(X))\}$ and $\delta_2^*(u_\varepsilon, u) = q_{h_X(u)}$. Thus, the preposition holds from the definitions of f and h_X.

Property (5) clearly holds from previous properties and thus both automata accept the same language. □

The main consequence of the previous Theorem is that during the transformation of a nondeterministic tree-like automaton with the self loop for all $a \in A$ in the initial state to the deterministic one, the number of states does not increase.

But it is possible to show more. It is easy to see that a deterministic dictionary matching automaton accepting language A^*X can be build from any acyclic automaton accepting language X by the last two steps of Algorithm 2.31. As can be seen in the next Theorem, the number of states of an automaton created in such way cannot be greater than the number of states of finite automaton accepting the same language and created by Algorithm

2.31 from scratch.

Theorem 2.35
Given acyclic automaton $M = (Q, A, \delta, q_0, F)$ accepting language X, finite automaton $M_1 = (Q_1, A, \delta_1, q_0^1, F_1)$ accepting language A^*X created by last two steps of Algorithm 2.31 contains at most the same number of states as finite automaton $M_2 = (Q_2, A, \delta_2, q_0^2, F_2)$ accepting the same language and created by Algorithm 2.31 from scratch.

Proof
Let us remind that deterministic automaton M_1 was created from nondeterministic automaton $M' = (Q, A, \delta', q_0, F)$ by the subset construction.

The set of active states of automaton M' after reading $u \in A$ is $\delta'^*(q_0, u)$, which is equal to $\delta'^*(q_0, h_X(u))$. Let us denote the active state of automaton M_1 after reading u by q. Subset construction ensures that $\delta'^*(q_0, u) \subseteq q$. So, in the worst case for all $q \in Q_1$ it holds $q = \delta'^*(q_0, u)$, $u \in \mathit{Pref}(X)$, which completes the proof. □

2.4.2 Approximate string matching

In order to prove the upper bound of the state complexity of deterministic finite automata for approximate string matching, it is necessary to limit the number of states of the dictionary matching automata accepting language A^*X with respect to the size of language X.

Theorem 2.36
Given acyclic finite automaton M accepting language X, the number of states of the deterministic dictionary matching automaton created from M is

$$\mathcal{O}(\sum_{w \in X} |w|).$$

Proof
Because of Theorem 2.35, the number of states of such way created deterministic dictionary matching automaton is at most the same as the number of states of a tree-like finite automaton accepting X, whose number of states is in the worst case equal to $1 + \sum_{w \in X} |w|$. □

2.4.2.1 Hamming distance At first, it is necessary to define the finite automaton for approximate string matching using Hamming distance. It is the "Hamming" automaton $M(A^*H_k(p))$ accepting language $A^*H_k(p)$, where $H_k(p) = \{u \mid u \in A^*, D_H(u, p) \leq k\}$ for the given pattern $p \in A^*$ and the number of allowed errors $k \geq 1$.

Since $H_k(p)$ is the finite language, it is possible to estimate the number of states of the deterministic automaton $M(A^*H_k(p))$ using Theorem 2.36. The only concern is to compute the size of the language $H_k(p)$.

Theorem 2.37
The number of strings generated by at most k replace operations from the pattern $p = p_1p_2 \dots p_m$ is $\mathcal{O}\left(|A|^k m^k\right)$. □

Proof
The set of strings created by exactly $i\,(0 \leq i \leq k)$ operations replace are made by replacing exactly i symbols of p by other symbols. There are $\binom{m}{i}$ possibilities for choosing i symbols from m. Each chosen symbol can be replaced by $|A| - 1$ symbols, so the number of generated strings is at most

$$\binom{m}{i}(|A|-1)^i = (|A|-1)^i\mathcal{O}\left(m^i\right) = \mathcal{O}\left(|A|^i m^i\right),$$

because $\binom{m}{i} = \mathcal{O}(m^i)$. The set of strings created by at most k operations replace is the union of the abovementioned sets of strings. Thus, the cardinality of this set is

$$\sum_{i=0}^{k} \mathcal{O}\left(|A|^i m^i\right) = \mathcal{O}\left(|A|^k m^k\right).$$

□

Since the number of strings generated by the replace operation is now known, it is possible to estimate the number of states of the deterministic Hamming automaton.

Theorem 2.38
The number of states of deterministic finite automaton $M(A^*H_k(p))$, $p = p_1p_2 \dots p_m$ is

$$\mathcal{O}\left(|A|^k m^{k+1}\right).$$

Proof
As shown in Theorem 2.36 the number of states of this automaton is at most the same as the size of language $H_k(p)$. As for all $u \in H_k(p)$ holds $|u| = m$, the size of language $H_k(p)$ is

$$\mathcal{O}(\sum_{u \in H_k(p)} |u|) = \mathcal{O}\left(m|A|^k m^k\right) = \mathcal{O}\left(|A|^k m^{k+1}\right).$$

□

2.4.2.2 Levenshtein distance The same approach as in Section 2.4.2.1 can be used to bound the number of states of a deterministic automaton for approximate string matching using Levenshtein distance. It is the "Levenshtein" automaton $M(A^*L_k(p))$ accepting language $A^*L_k(p)$, where $L_k(p) = \{u \mid u \in A^*, D_L(u,p) \leq k\}$ for given pattern $p \in A^*$ and the number of allowed errors $k \geq 1$.

So, the number of strings generated from pattern by insert and delete operations is to be found.

Theorem 2.39
The number of strings generated by at most k insert operations from the pattern $p = p_1 p_2 \dots p_m$ is $\mathcal{O}\left(|A|^k m^k\right)$.

Proof
Imagine that all above mentioned strings are generated from empty strings by sequential symbol addition. Then the number of strings generated by exactly i insert operations can be transformed to the number of tours that can be found in a chessboard like oriented graph from the position $(0,0)$ to position (m,i) multiplied by A^i (it is possible to insert an arbitrary symbol from the alphabet). Position $(0,0)$ represents empty string, position (m,i) represents pattern p with i inserted symbols, and each move of the tour represents addition of the one symbol to the generated string. Move $(x,y) \rightarrow (x+1,y)$ represents addition of the $(x+1)$-st symbol of the pattern, while move $(x,y) \rightarrow (x,y+1)$ represents addition of an inserted symbol to the pattern. An example of this graph is shown in Figure 2.36. The number in each node represents the number of tours leading to the node. The number

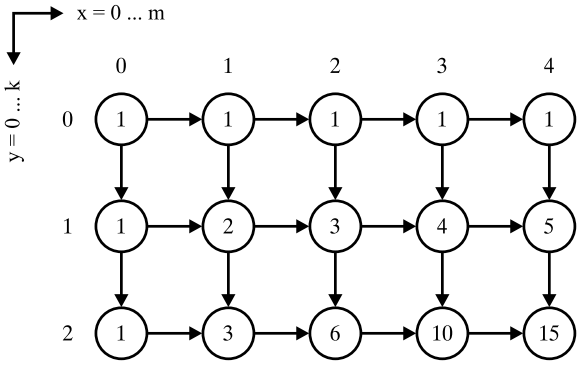


Figure 2.36: Chessboard like oriented graph representing strings generated by insert operation

of tours $c_{x,y}$ can be defined by recursive formula

$$\begin{aligned} c_{x,0} &= 1 && 0 \leq x, \\ c_{0,y} &= 1 && 0 \leq y, \\ c_{x,y} &= c_{x-1,y} + c_{x,y-1} && 1 \leq x \quad 1 \leq y. \end{aligned}$$

Because of the Pascal triangle method of binomial number computation, it is clear that

$$c_{x,y} = \binom{x+y}{min(x,y)}.$$

In order to continue, it is possible to use following equation:

$$\begin{aligned}\binom{x+z}{z} &= \frac{(x+1)\cdot(x+2)\cdot\ldots\cdot(x+z-1)\cdot(x+z)}{1\cdot 2\cdot\ldots\cdot(z-1)\cdot z} = \\ &= \frac{x+1}{1}\cdot\frac{x+2}{2}\cdot\ldots\cdot\frac{x+(z-1)}{z-1}\cdot\frac{x+z}{z} = \\ &= \left(\frac{x}{1}+1\right)\cdot\left(\frac{x}{2}+1\right)\cdot\ldots\cdot\left(\frac{x}{z-1}+1\right)\cdot\left(\frac{x}{z}+1\right).\end{aligned}$$

In case that $x \geq 2$ all fractions but the first one are smaller than (or equal to) x. Thus

$$\binom{x+z}{z} \leq (x+1) * x^{z-1} = x^z + x^{z-1} = \mathcal{O}\left(x^z\right).$$

As the number of allowed errors is smaller than the length of pattern, $i < m$,

$$c_{m,i} = \binom{m+i}{min(m,i)} = \binom{m+i}{i} = \mathcal{O}\left(m^i\right).$$

Thus, the number of strings generated by exactly i insert operations is $\mathcal{O}\left(|A|^i m^i\right)$. Since the set of strings generated by at most k insert operations is the union of the above mentioned sets of strings, the cardinality of this set is

$$\sum_{i=0}^{k} \mathcal{O}\left(|A|^i m^i\right) = \mathcal{O}\left(|A|^k m^k\right).$$

□

Theorem 2.40
The number of strings generated by at most k delete operations from pattern $p = p_1 p_2 \ldots p_m$ is $\mathcal{O}\left(m^k\right)$.

Proof
Sets of strings generated by exactly i delete operations consist of strings that are made by deleting exactly i symbols from p. There are $\binom{m}{i}$ possibilities for choosing i symbols from m, so the number of such strings is at most $\binom{m}{i} = \mathcal{O}\left(m^i\right)$. Since the set of strings generated by at most k delete operations is the union of the above mentioned sets, the number of strings within this set is

$$\sum_{i=0}^{k} \mathcal{O}\left(m^i\right) = \mathcal{O}\left(m^k\right).$$

□

Now it is known the number of strings generated by each edit operation, so it is possible to estimate the number of strings generated by these operations all at once.

Theorem 2.41
The number of strings generated by at most k replace, insert, and delete operations from pattern $p = p_1 p_2 \dots p_m$ is $\mathcal{O}\left(|A|^k m^k\right)$.

Proof
The number of strings generated by at most k replace, insert, delete operations can be computed as a sum of the number of strings generated by these operations for exactly i errors allowed for $0 \leq i \leq k$. Such strings are generated by a combination of above mentioned operations, so the number of generated strings is

$$\sum_{x=0}^{k} \sum_{y=0}^{k-x} \sum_{z=0}^{k-x-y} \underbrace{\mathcal{O}\left(|A|^x m^x\right)}_{\substack{replace\ x \\ symbols}} \underbrace{\mathcal{O}\left(|A|^y m^y\right)}_{\substack{insert\ y \\ symbols}} \underbrace{\mathcal{O}\left(m^z\right)}_{\substack{delete\ z \\ symbols}} =$$

$$= \sum_{x=0}^{k} \sum_{y=0}^{k-x} \sum_{z=0}^{k-x-y} \mathcal{O}\left(|A|^{x+y} m^{x+y+z}\right) =$$

$$= \mathcal{O}\left(|A|^k m^k\right) \qquad \square$$

The last step is to bound the number of states of the deterministic dictionary matching automaton.

Theorem 2.42
The number of states of deterministic finite automaton $M(A^* L_k(p))$, $p = p_1 p_2 \dots p_m$ is

$$\mathcal{O}\left(|A|^k m^{k+1}\right).$$

Proof
The proof is the same as for Theorem 2.38. Since for all $u \in L_k(p)$ holds $|u| \leq m + k$, the size of language $L_k(p)$ is

$$\mathcal{O}(\sum_{u \in L_k(p)} |u|) = \mathcal{O}\left((m+k)|A|^k m^k\right) = \left(|A|^k m^{k+1}\right). \qquad \square$$

2.4.2.3 Generalized Levenshtein distance It is clear that the concepts from Sections 2.4.2.1 and 2.4.2.2 will be used also for generalized Levenshtein distance.

The finite automaton for approximate string matching using generalized Levenshtein distance $M(A^* G_k(p))$ will be called the finite automaton accepting language $A^* G_k(p)$, where $G_k(p) = \{u \mid u \in A^*, D_G(u, p) \leq k\}$ for given pattern $p \in A^*$ and the number of allowed errors $k \geq 1$.

The only unexplored operation is the transpose operation.

Theorem 2.43
The number of strings generated by at most k transpose operations from pattern $p = p_1 p_2 \dots p_m$ is $\mathcal{O}\left(m^k\right)$.

Proof
The strings generated by exactly i transpose operations are made by transposing exactly i pairs of symbols from p. There is less than $\binom{m-1}{i}$ possibilities for choosing i pairs from m symbols for transpose operation, when each symbol can participate only in one pair. The number of such generated strings is at most $\binom{m-1}{i} = \mathcal{O}\left(m^i\right)$. Since the set of strings generated by at most k transpose operations is the union of the above mentioned sets of strings, its cardinality is

$$\sum_{i=0}^{k} \mathcal{O}\left(m^i\right) = \mathcal{O}\left(m^k\right).$$

□

Now, the number of strings generated by all operations defined by generalized Levenshtein distance is to be found.

Theorem 2.44
The number of strings generated by at most k replace, insert, delete, and transpose operations from the pattern $p = p_1 p_2 \dots p_m$ is $\mathcal{O}\left(|A|^k m^k\right)$.

Proof
The number of strings generated by at most k replace, insert, delete, and transpose operations can be computed as a sum of the numbers of strings generated by these operations for exactly i errors allowed for $0 \leq i \leq k$. Such strings are generated by a combination of above mentioned operations. So the number of generated strings is

$$\sum_{w=0}^{k} \sum_{x=0}^{k-w} \sum_{y=0}^{k-w-x} \sum_{z=0}^{k-w-x-y} \underbrace{\mathcal{O}\left(|A|^w m^w\right)}_{\substack{replace\ w \\ symbols}} \underbrace{\mathcal{O}\left(|A|^x m^x\right)}_{\substack{insert\ x \\ symbols}} \underbrace{\mathcal{O}\left(m^y\right)}_{\substack{delete\ y \\ symbols}} \underbrace{\mathcal{O}\left(m^z\right)}_{\substack{transpose\ z \\ pairs}} =$$

$$= \sum_{w=0}^{k} \sum_{x=0}^{k-w} \sum_{y=0}^{k-w-x} \sum_{z=0}^{k-w-x-y} \mathcal{O}\left(|A|^{w+x} m^{w+x+y+z}\right) =$$

$$= \mathcal{O}\left(|A|^k m^k\right)$$

□

Finally, the number of states of the deterministic dictionary matching automaton will be found.

Theorem 2.45
The number of states of deterministic finite automaton $M(A^*G_k(p))$, $p = p_1p_2\ldots p_m$ is

$$\mathcal{O}\left(|A|^k m^{k+1}\right).$$

Proof
The proof is the same as for Theorem 2.38. Since for all $u \in G_k(p)$ holds $|u| \leq m + k$ the size of language $G_k(p)$ is

$$\mathcal{O}(\sum_{u \in G_k(p)} |u|) = \mathcal{O}\left((m+k)|A|^k m^k\right) = \mathcal{O}\left(|A|^k m^{k+1}\right). \qquad \square$$

2.4.2.4 Γ distance The finite automaton for approximate string matching using Γ distance $M(A^*\Gamma_k(p))$ will be called the finite automaton accepting language $A^*\Gamma_k(p)$, where A is an ordered alphabet, $\Gamma_k(p) = \{u \mid u \in A^*, D_\Gamma(u,p) \leq k\}$ for the given pattern $p \in A^*$ and the number of allowed errors $k \geq 1$.

Since it will be used the same approach as in previous sections, it is necessary to compute cardinality of the set $\Gamma_k(p)$. In order to do that, it is necessary to prove two auxiliary lemmas.

Theorem 2.46
For all $i \geq 1$, $j \geq 1$, $(j-1)^{i+1} + (j-1)^i + j^i \leq j^{i+1}$.

Proof
It will be shown by induction on i. It holds for $i = 1$:

$$(j-1)^2 + (j-1) + j = j^2 - 2j + 1 + j - 1 + j = j^2 \leq j^2.$$

Consider the assumption is fulfilled for $i \geq 1$. Then for $i + 1$

$$\begin{aligned} & (j-1)^{i+2} + (j-1)^{i+1} + j^{i+1} = \\ = \; & (j-1)(j-1)^{i+1} + (j-1)(j-1)^i + j\,j^i \leq \end{aligned}$$

as $j \geq 1$

$$\begin{aligned} \leq \; & j(j-1)^{i+1} + j(j-1)^i + j\,j^i = \\ = \; & j\left((j-1)^{i+1} + (j-1)^i + j^i\right) \leq \end{aligned}$$

from induction assumption

$$\leq \; j\,j^{i+1} = j^{i+2}$$

which completes the proof. $\square$

Theorem 2.47
For all $i \geq 3$, $j \geq 2$, $\sum_{x=0}^{i}(j-1)^x + \sum_{x=0}^{i-1}(j-1)^x \leq j^i$.

Proof
It will be shown by induction on i. It is satisfied for $i = 3$:

$$\begin{aligned} & \sum_{x=0}^{3}(j-1)^x + \sum_{x=0}^{2}(j-1)^x \\ = \;& 2(j-1)^0 + 2(j-1)^1 + 2(j-1)^2 + (j-1)^3 = \\ = \;& 2 + 2j - 2 + 2j^2 - 4j + 2 + j^3 - 2j^2 + j - j^2 + 2j - 1 = \\ = \;& j^3 - j^2 + j + 1 \leq \end{aligned}$$

since $j \geq 2$

$$\leq \; j^3$$

Consider the assumption is satisfied for $i \geq 3$. Than for $i + 1$

$$\begin{aligned} & \sum_{x=0}^{i+1}(j-1)^x + \sum_{x=0}^{i}(j-1)^x = \\ = \;& (j-1)^{i+1} + (j-1)^i + \sum_{x=0}^{i}(j-1)^x + \sum_{x=0}^{i-1}(j-1)^x \leq \end{aligned}$$

from induction assumption

$$\leq \; (j-1)^{i+1} + (j-1)^i + j^i \leq$$

from Lemma 2.46

$$\leq \; j^{i+1} \tag{3}$$

which completes the proof. □

Theorem 2.48
The number of strings generated by from the pattern $p = p_1 p_2 \ldots p_m$ with at most k allowed errors in Γ distance is $\mathcal{O}\left(m^k\right)$.

Proof
The number of strings generated from the pattern p of the length m for exactly k errors can be computed as the number of different paths in the transition diagram of an automaton $M(\Gamma_k(p))$ from the initial state $q_{0,0}$ to the final state $q_{k,m}$, where the transition diagram of the automaton $M(\Gamma_k(p))$ is the same as the transition diagram of the automaton $M(A^*\Gamma_k(p))$ (shown in Figure **REF**) without the self loop for the whole alphabet in the initial

state $q_{0,0} \in \delta(q_{0,0}, a)$, $\forall a \in A$. The number of these paths can be computed by following recurrent formula:

$$c_{i,j} = \begin{cases} 1 & i = 0, j = 0 \\ 0 & i > 0, j = 0 \\ c_{i,j-1} + 2\sum_{x=0}^{i-1} c_{x,j-1} & \text{otherwise} \end{cases} .$$

Let us show in several steps, that $c_{i,j} \leq 2j^i$.

- $i = 0, j > 0$: $c_{0,j} = c_{0,j-1} = c_{0,j-2} = \ldots = 1 \leq 2\, j^0$
- $i = 1, j > 0$: By induction on j. It is satisfied for $j = 1$ because $c_{1,1} = 2 \leq 2\,1^1$. Consider the assumption holds for $j > 0$. Than for $j + 1$

  $$c_{1,j+1} = c_{1,j} + 2 \cdot c_{0,j} \leq$$

  from induction assumption and the fact that $c_{0,j} = 1$

  $$\leq \quad 2j + 2 = 2(j + 1) \leq 2(j + 1)^1$$

- $i = 2, j > 0$: By induction on j. It is satisfied for $j = 1$ because $c_{2,1} = 2 \leq 2\,1^2$. Consider the assumption holds for $j \geq 1$. Than for $j + 1$

  $$c_{2,j+1} = c_{2,j} + 2 \cdot c_{1,j} + 2 \cdot c_{0,j} \leq$$

  from induction assumption

  $$\begin{aligned} &\leq \quad 2j^2 + 2 \cdot 2j + 2 \cdot 1 = 2(j^2 + 2j + 1) \leq \\ &\leq \quad 2(j + 1)^1 = 2(j + 1)^2 \end{aligned}$$

- $i \geq 3, j = 1$: $c_{i,1} = c_{i,0} + 2 \cdot c_{i-1,0} + 2 \cdot c_{i-2,0} + \ldots + 2 \cdot c_{0,0} = 2$.
- $i \geq 3, j \geq 2$: By induction on j. It was shown in previous step that the assumption holds for $j - 1 \geq 1$. Then for j

  $$c_{i,j} = c_{i,j-1} + 2\sum_{x=0}^{i-1} c_{x,j-1} \leq \tag{4}$$

  from induction assumption

  $$\leq \quad 2(j-1)^i + 2\sum_{x=0}^{i-1} 2(j-1)^x \leq \sum_{x=0}^{i} 2(j-1)^x + \sum_{x=0}^{i-1} 2(j-1)^x \leq$$

  from Lemma 2.47

  $$\leq \quad 2j^i$$

Thus the number of string generated from pattern of the length m with exactly i allowed errors in Γ distance is $\mathcal{O}\left(m^i\right)$.

Since the set of strings generated from pattern of the length m with at most k allowed errors is the union of the above mentioned sets of strings, its cardinality is

$$\sum_{i=0}^{k} \mathcal{O}\left(m^i\right) = \mathcal{O}\left(m^k\right).$$

□

Finally, the number of states of the deterministic dictionary matching automaton will be found.

Theorem 2.49
The number of states of the deterministic finite automaton $M(A^*\Gamma_k(p))$, $p = p_1p_2 \ldots p_m$ is

$$\mathcal{O}\left(m^{k+1}\right).$$

Proof
The proof is the same as for Lemma 2.38. Since for all $u \in \Gamma_k(p)$, it holds $|u| = m$ the size of the language $\Gamma_k(p)$ is

$$\mathcal{O}(\sum_{u \in \Gamma_k(p)} |u|) = \mathcal{O}\left(m\ m^k\right) = \mathcal{O}\left(m^{k+1}\right).$$

□

2.5 (Δ, Γ) distance

The finite automaton for approximate string matching using $(\Delta\Gamma)$ distance $M(A^*(\Delta_l\Gamma_k)(p))$ will be called the finite automaton accepting language $A^*(\Delta_l\Gamma_k)(p)$, where A is an ordered alphabet, $(\Delta_l\Gamma_k)(p) = \{u \mid u \in A^*, D_\Delta(u,p) \leq l, D_\Gamma(u,p) \leq k\}$ for the given pattern $p \in A^*$ and the number of allowed errors $k, l \in \mathbb{N}$.

It is obvious that it will be used the same approach as in previous sections. Thus it is necessary to compute cardinality of the set $(\Delta_l\Gamma_k)(p)$.

Let us start by special case when $l = 1$. In order to do that, it is necessary to prove one auxiliary lemma.

Theorem 2.50
For all $i \geq 2$, $j \geq 1$, $(j-1)^i + 2(j-1)^{i-1} \leq j^i$.

Proof
It will be shown by induction on i. It holds for $i = 2$:

$$(j-1)^2 + 2(j-1) = j^2 - 2j + 1 + 2j - 2 = j^2 - 1 \leq j^2.$$

Consider the assumption is fulfilled for $i \geq 2$. Than for $i + 1$

$$(j-1)^{i+1} + 2(j-1)^i = (j-1)\left((j-1)^i + 2(j-1)^{i-1}\right) \leq$$

from induction assumption

$$\leq \quad (j-1)j^i = j^{i+1} - j^i \leq j^{i+1}$$

which completes the proof. □

Theorem 2.51
The number of strings generated from pattern p of the length m with at most k allowed errors in Γ distance and at most 1 allowed error in Δ distance is $\mathcal{O}\left(m^k\right)$.

Proof
The number of strings generated from the pattern p of the length m for exactly k errors can be computed as the number of different paths in the transition diagram of an automaton $M((\Delta_l\Gamma_k)(p))$ from the initial state $q_{0,0}$ to the final state $q_{k,m}$, where the transition diagram of the automaton $M((\Delta_l\Gamma_k)(p))$ is the same as the transition diagram of the automaton $M(A^*(\Delta_l\Gamma_k)(p))$ (shown in Figure **REF**) without the self loop for the whole alphabet in the initial state $q_{0,0} \in \delta(q_{0,0}, a)$, $\forall a \in A$. The number of these paths can be computed by following recurrent formula:

$$c_{i,j} = \begin{cases} 1 & i = 0, j = 0 \\ c_{i,j-1} + 2c_{i-1,j-1} & 0 \leq i \leq j, 0 < j \\ 0 & \text{otherwise} \end{cases} .$$

Let us show in several steps that $c_{i,j} \leq 2j^i$.

- $i = 0$, $j > 0$: $c_{0,j} = c_{0,j-1} = \ldots = c_{0,0} = 1 \leq 2j^0$
- $i = 1$, $j > 0$: By induction on j. It is satisfied for $j = 1$ because $c_{1,1} = c_{1,0} + 2c_{0,0} = 2 \leq 2$. Consider the assumption holds for $j > 0$. Than for $j + 1$

  $$c_{1,j+1} = c_{1,j} + 2c_{0,j} \leq$$

  from induction assumption and the fact that $c_{0,j} = 1$

  $$2j + 2 = 2(j + 1) \leq 2(j + 1)^1.$$

- $i \geq 2$, $j \geq 1$; By induction on j. It is satisfied for $j = 1$ because $c_{i,1} = c_{i,0} + 2c_{i-1,0} = 0$. Consider the condition holds for $j - 1 \geq 1$. Than for j

  $$c_{i,j} = c_{i,j-1} + 2c_{i-1,j-1} \leq$$

  from induction assumption

  $$\leq 2(j-1)^i + 2 \cdot 2(j-1)^{j-1}$$

  from Lemma 2.50

  $$\leq 2j^i.$$

Thus the number of string generated from pattern of the length m with exactly i allowed errors in Γ distance and 1 error in Δ distance is $\mathcal{O}\left(m^i\right)$.

Since the set of strings generated from pattern of the length m with at most k allowed errors in Γ distance and 1 error in Δ distance is the union of the above mentioned sets of strings, its cardinality is

$$\sum_{i=0}^{k} \mathcal{O}\left(m^i\right) = \mathcal{O}\left(m^k\right).$$

□

The other special case is that $l \geq k$. It is quite easy to see, that this is the same case as when just Γ distance is used. The number of strings generated in this case, which was given by Lemma 2.48, is $\mathcal{O}\left(m^k\right)$.

Since the asymptotic number of strings generated by combined $(\Delta\Gamma)$ distance is the same in both cases ($l = 1$ and $l \geq k$), the number of allowed errors in Δ distance does not affect the asymptotic number of generated strings.

Now it is possible to estimate the number of states of the deterministic dictionary matching automaton.

Theorem 2.52
The number of states of the deterministic finite automaton $M(A^*(\Delta_l\Gamma_k)(p))$, $p = p_1p_2\ldots p_m$ is

$$\mathcal{O}\left(m^{k+1}\right).$$

Proof
The proof is the same as for Lemma 2.38. Since for all $u \in (\Delta_l\Gamma_k)(p)$, it holds $|u| = m$ the size of the language $(\Delta_l\Gamma_k)(p)$ is

$$\mathcal{O}(\sum_{u\in(\Delta_k\Gamma_k)(p)} |u|) = \mathcal{O}\left(m\ m^k\right) = \mathcal{O}\left(m^{k+1}\right).$$

□

2.6 Δ distance

The finite automaton for approximate string matching using Δ distance $M(A^*\Delta_k(p))$ will be called the finite automaton accepting language $A^*\Delta_k(p)$, where A is an ordered alphabet, $\Delta_k(p) = \{u \mid u \in A^*, D_\Delta(u,p) \leq k\}$ for the given pattern $p \in A^*$ and the number of allowed errors $k \in \mathbb{N}$.

Since it will be used the same approach as in previous sections, it is necessary to compute cardinality of the set $\Delta_k(p)$.

Theorem 2.53
The number of strings generated by from the pattern $p = p_1p_2\ldots p_m$ with at most k allowed errors in Δ distance is $\mathcal{O}\left((2k+1)^m\right)$.

Proof
Since Δ distance is computed as a maximum of distances of individual symbols at corresponding positions, each symbol can be replaced by at most $2k$ different symbols (k symbols that are smaller and k symbols that are bigger in the alphabet ordering). Therefore, at each position can be at most $2k+1$ different symbols. As, the length of the pattern is m, Δ distance generates at most $(2k+1)^m$ different strings. □

Finally, let us compute the number of states of the deterministic dictionary matching automaton.

Theorem 2.54
The number of states of the deterministic finite automaton $M(A^*\Delta_k(p))$, $p = p_1p_2 \dots p_m$ is

$$\mathcal{O}\left(m(2k+1)^m\right).$$

Proof
The proof is the same as for Lemma 2.38. Since for all $u \in \Delta_k(p)$, it holds $|u| = m$ the size of the language $\Delta_k(p)$ is

$$\mathcal{O}(\sum_{u \in \Delta_k(p)} |u|) = \mathcal{O}\left(m(2k+1)^m\right) = \mathcal{O}\left(m(2k+1)^m\right).$$

□

3 Finite automata accepting parts of a string

In this Chapter we explain how to construct finite automata accepting all prefixes, suffixes, factors, and subsequences of a given string. At the end we show the construction of factor oracle automaton accepting all factors of a given string and moreover some of its subsequences.

3.1 Prefix automaton

Having string $x = a_1a_2 \dots a_n$ we can express set *Pref*(x) (see Def. 1.1) using the following two forms of regular expressions:

$$\begin{aligned} R_{Pref} &= \varepsilon + a_1 + a_1a_2 + \ldots + a_1a_2 \dots a_n \\ &= \varepsilon + a_1(\varepsilon + a_2(\varepsilon + \ldots + a_{n-1}(\varepsilon + a_n) \dots)). \end{aligned}$$

Using the first form of regular expression R_{Pref}, we can construct the finite automaton accepting set *Pref*(x) using Algorithm 3.1.

Algorithm 3.1
Construction of the prefix automaton I (union of prefixes).
Input: String $x = a_1a_2 \dots a_n$.
Output: Finite automaton M accepting language *Pref*(x).
Method: We use description of language *Pref*(x) by regular expression:

$$R_{Pref} = \varepsilon + a_1 + a_1a_2 + \ldots + a_1a_2 \dots a_n.$$

1. Construct n finite automata M_i accepting strings $a_1a_2 \dots a_i$ for all $i = 0, 1, \dots, n$.
2. Construct automaton M accepting union of languages $L(M_i)$, $i = 0, 1, \dots, n$.

   $L(M) = L(M_0) \cup L(M_1) \cup L(M_2) \cup \ldots \cup L(M_n)$. □

Transition diagram of the prefix automaton constructed by Algorithm 3.1 is depicted in Fig. 3.1.

If we use the second form of regular expression:

$$R_{Pref} = \varepsilon + a_1(\varepsilon + a_2(\varepsilon + \ldots + a_{n-1}(\varepsilon + a_n) \dots)),$$

we can construct the finite automaton using Algorithm 3.2.

Algorithm 3.2
Construction of prefix automaton II (set of neighbours).
Input: String $x = a_1a_2 \dots a_n$.
Output: Finite automaton M accepting language *Pref*(x).
Method: We use description of language *Pref*(x) by regular expression:

$$R_{Pref} = \varepsilon + a_1(\varepsilon + a_2(\varepsilon + \ldots + a_{n-1}(\varepsilon + a_n) \dots)).$$

1. We will use the method of neighbours.
   (a) The set of initial symbols: $IS = \{a_1\}$.

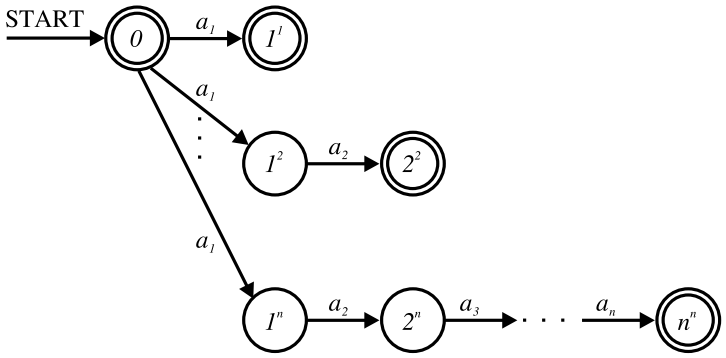


Figure 3.1: Transition diagram of the finite automaton accepting language *Pref*$(a_1a_2 \dots a_n)$ constructed for regular expression $R_{Pref} = \varepsilon + a_1 + a_1a_2 + \dots + a_1a_2 \dots a_n$ from Algorithm 3.1

(b) The set of neighbours: $NS = \{a_1a_2, a_2a_3, \dots, a_{n-1}a_n\}$.
(c) The set of final symbols: $FS = \{a_1, a_2, \dots, a_n\}$.

2. Construct automaton
$M = (\{q_0, q_1, q_2, \dots, q_n\}, A, \delta, q_0, F)$
where $\delta(q_0, a_1) = q_1$ because $IS = \{a_1\}$,
$\delta(q_i, a_{i+1}) = q_{i+1}$ for all $i = 1, 2, \dots, n-1$,
because a) each state $q_i, i = 1, 2, \dots, n-1$, coresponds to the prefix $a_1a_2 \dots a_i$,
b) $a_ia_{i+1} \in NS$,
$F = \{q_0, q_1, \dots, q_n\}$ because the set of final symbols is
$FS = \{a_1, a_2, \dots, a_n\}$ and $\varepsilon \in h(R_{Pref})$. □

Transition diagram of the resulting prefix is automaton M depicted in Fig. 3.2.

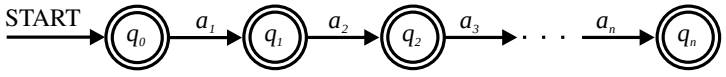


Figure 3.2: Transition diagram of the finite automaton accepting language *Pref*$(a_1a_2 \dots a_n)$ constructed for regular expression $R_{Pref} = \varepsilon + a_1(\varepsilon + a_2(\varepsilon + \dots + a_{n-1}(\varepsilon + a_n)\dots))$ from Algorithm 3.2

Example 3.3
Let us have string $x = abab$. Construct automata accepting *Pref*(x) using both methods of their construction. Using Algorithm 3.1, we obtain prefix automaton M_1 having the transition diagram depicted in Fig. 3.3. Algorithm 3.2 yields prefix automaton M_2 having transition diagram depicted in Fig. 3.4. Both automata M_1 and M_2 are accepting language

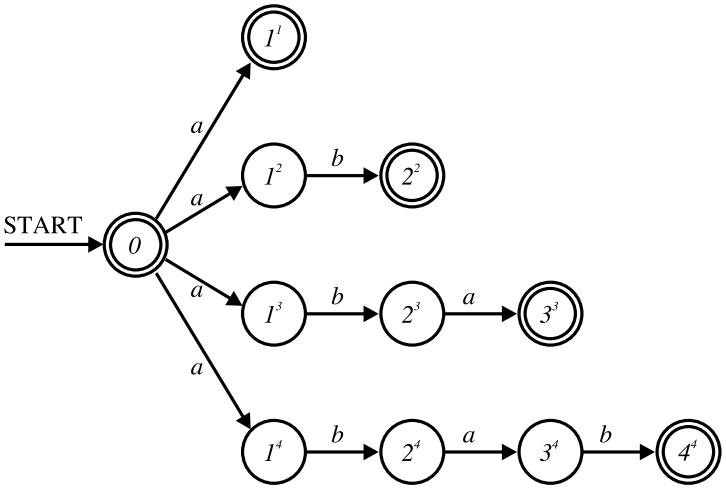


Figure 3.3: Transition diagram of prefix automaton M_1 accepting *Pref*(*abab*) from Example 3.3

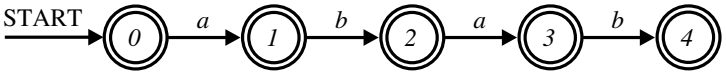


Figure 3.4: Transition diagram of prefix automaton M_2 accepting *Pref*(*abab*) from Example 3.3

Pref(*abab*) and therefore they should be equivalent. Let us show it:
As prefix automaton M_1 is nondeterministic, let us construct its deterministic equivalent M_1'. Its transition diagram is depicted in Fig. 3.5 and it is obvious that both automata M_1' and M_2 are equivalent. □

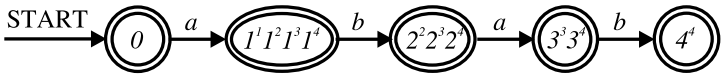


Figure 3.5: Transition diagram of deterministic automaton M_1' from Example 3.3

The second variant of the construction of the prefix automaton is more straightforward than the first one. Therefore we will simplify it for the practical use in the following algorithm. As the states in this automaton correspond to the length of respective prefixes, we will use integer numbers as labels of states.

Algorithm 3.4
The construction of a finite automaton accepting set *Pref*(x).
Input: String $x = a_1 a_2 \dots a_n$.

Output: Deterministic finite automaton $M = (Q, A, \delta, q_0, F)$ accepting set *Pref*(x).
Method:
$Q = \{0, 1, 2, \ldots, n\}$,
A is the set of all different symbols in x,
$\delta(i-1, a_i) = i, i = 1, 2, \ldots, n$,
$q_0 = 0$,
$F = \{0, 1, 2, \ldots, n\}$. □

Example 3.5
Let us construct the deterministic finite automaton accepting *Pref*$(abba) = \{\varepsilon, a, ab, abb, abba\}$ using Algorithm 3.4. The resulting automaton $M = (\{0, 1, 2, 3, 4\}, \{a, b\}, \delta, 0, \{0, 1, 2, 3, 4\})$. Its transition diagram is depicted in Fig 3.6. □

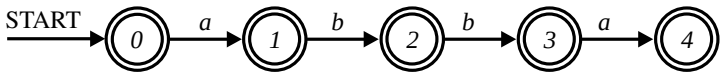


Figure 3.6: Transition diagram of finite automaton M accepting set *Pref*$(abba)$ from Example 3.5

3.2 Suffix automaton

Having string $x = a_1 a_2 \ldots a_n$, we can express set *Suff*(x) (see Def. 1.2) using the following two forms of regular expressions:

$$\begin{aligned} R_{Suff}(x) &= a_1 a_2 \ldots a_n + a_2 a_3 \ldots a_n + \ldots + a_n + \varepsilon \\ &= (\ldots((a_1 + \varepsilon)a_2 + \varepsilon)a_3 + \ldots + \varepsilon)a_n + \varepsilon. \end{aligned}$$

Using the first form of regular expression R_{Suff} we can construct the finite automaton accepting set *Suff*(x) using Algorithm 3.6. Let us call it the suffix automaton for string x.

Algorithm 3.6
Construction of the suffix automaton I (union of suffixes).
Input: String $x = a_1 a_2 \ldots a_n$.
Output: Finite automaton M accepting language *Suff*(x).
Method: We use description of language *Suff*(x) by regular expression:
$R_{Suff} = a_1 a_2 \ldots a_n + a_2 \ldots a_n + \ldots + a_n + \varepsilon$.

1. Construct n finite automata M_i accepting strings $a_i a_{i+1} \ldots a_n$ for $i = 1, 2, \ldots, n$. Construct automaton M_0 accepting empty string.
2. Construct automaton M_N accepting union of languages $L(M_i)$, $i = 0, 1, 2, \ldots, n$, i.e. $L(M_N) = L(M_0) \cup L(M_1) \cup L(M_2) \cup \ldots \cup L(M_n)$.

3. Construct deterministic automaton M equivalent to automaton M_N. □

Transition diagram of the suffix automaton constructed by Algorithm 3.6 is depicted in Fig. 3.7.

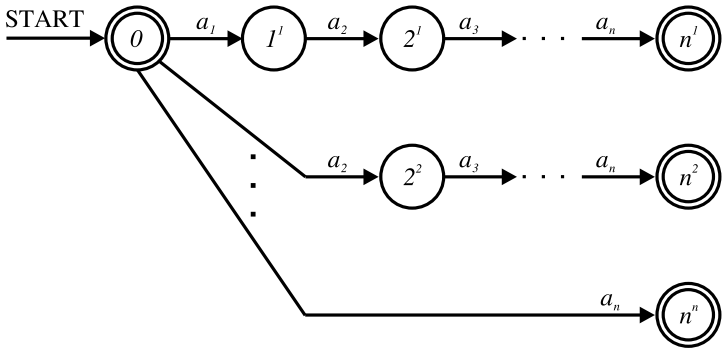


Figure 3.7: Transition diagram of finite automaton accepting language $Suff(a_1a_2 \dots a_n)$ constructed for regular expression $R_{Suff} = a_1a_2 \dots a_n + a_2 \dots a_n + \dots + a_n + \varepsilon$

If we use the second form of the regular expression:
$R_{Suff}(x) = (\dots(a_1 + \varepsilon)a_2 + \varepsilon)a_3 + \dots + \varepsilon)a_n + \varepsilon$,
we can construct the finite automaton using Algorithm 3.7.

Algorithm 3.7
Construction of the suffix automaton II (use of ε–transitions).
Input: String $x = a_1a_2 \dots a_n$.
Output: Finite automaton M accepting language $Suff(x)$.
Method: We use description of language $Suff(x)$ by regular expression
$R_{Suff}(x) = (\dots((a_1 + \varepsilon)a_2 + \varepsilon)a_3 + \dots + \varepsilon)a_n + \varepsilon$.

1. Construct finite automaton M_1 accepting string $x = a_1a_2 \dots a_n$.
   $M_1 = (\{q_0, q_1, \dots, q_n\}, A, \delta, q_0, \{q_n\})$,
   where $\delta(q_i, a_{i+1}) = q_{i+1}$ for all $i = 0, 1, \dots, n-1$.
2. Construct finite automaton $M_2 = (\{q_0, q_1, \dots, q_n\}, A, \delta', q_0, \{q_n\})$ from the automaton M_1 by inserting ε–transitions:
   $\delta(q_0, \varepsilon) = \{q_1, q_2, \dots, q_{n-1}, q_n\}$.
3. Replace all ε–transitions in M_2 by non–ε–transitions. The resulting automaton is M_3.
4. Construct deterministic finite automaton M equivalent to automaton M_3. □

Suffix automaton M_2 constructed by Algorithm 3.7 has, after step 2., transition diagram depicted in Fig. 3.8. Suffix automaton M_3 has, after step

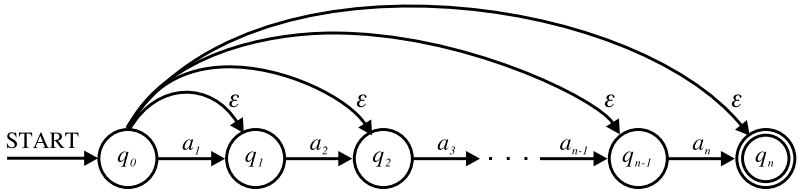


Figure 3.8: Transition diagram of suffix automaton M_2 with ε–transitions constructed in step 2. of Algorithm 3.7

3. of Algorithm 3.7, its transition diagram depicted in Fig. 3.9.

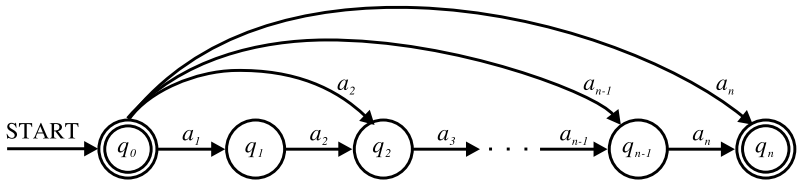


Figure 3.9: Transition diagram of suffix automaton M_3 after the removal of ε–transitions in the step 3. of Algorithm 3.7

We can use an alternative method for the construction of the suffix automaton described by the second form of regular expression:
$R_{Suff}(x) = (\ldots(a_1 + \varepsilon)a_2 + \varepsilon)a_3 + \ldots + \varepsilon)a_n + \varepsilon$.
Algorithm 3.8 uses this method.

Algorithm 3.8
Construction of the suffix automaton III (using more initial states).
Input: String $x = a_1 a_2 \ldots a_n$.
Output: Finite automaton M accepting language $Suff(x)$.
Method: We use description of language $Suff(x)$ by regular expression
$R_{Suff}(x) = (\ldots((a_1 + \varepsilon)a_2 + \varepsilon)a_3 + \ldots + \varepsilon)a_n + \varepsilon$.

1. Construct finite automaton M_1 accepting string $x = a_1 a_2 \ldots a_n$.
   $M_1 = (\{q_0, q_1, q_2, \ldots, q_n\}, A, \delta, q_0, \{q_n\})$,
   where $\delta(q_i, a_{i+1}) = q_{i+1}$ for all $i = 0, 1, \ldots, n-1$.
2. Construct finite automaton $M_2 = (\{q_0, q_1, q_2, \ldots, q_n\}, A, \delta, I, \{q_n\})$ from automaton M_1 having this set of initial states:
   $I = \{q_0, q_1, \ldots, q_{n-1}, q_n\}$.
3. Construct deterministic automaton M equivalent to automaton M_2. We use following steps for the construction:
   (a) Using Algorithm 1.39 construct automaton M_3 equivalent to automaton M_2 having just one initial state and ε-transitions from the initial state to all other states.

(b) Using Algorithm 1.38 construct automaton M_4 without ε-transitions equivalent to automaton M_3.

(c) Using Algorithm 1.40 construct deterministic automaton M equivalent to automaton M_4. □

Transition diagram of suffix automaton M_2 constructed in step 2. of Algorithm 3.8 is depicted in Fig. 3.10.

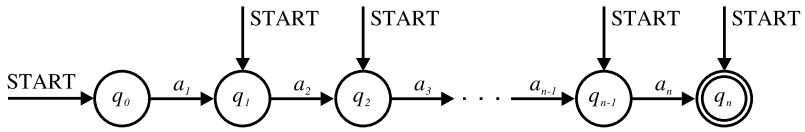


Figure 3.10: Transition diagram of suffix automaton M_2 accepting language $\mathit{Suff}(a_1a_2\ldots a_n)$ constructed in step 2. of Algorithm 3.8

Example 3.9
Let us have string $x = abab$. Construct automata accepting $\mathit{Suff}(x)$ using all three methods of their construction. Using Algorithm 3.6 we obtain (after the step 2.) suffix automaton M_1 having the transition diagram depicted in Fig. 3.11. Algorithm 3.7 yields, after step 2., suffix automaton M_2 with

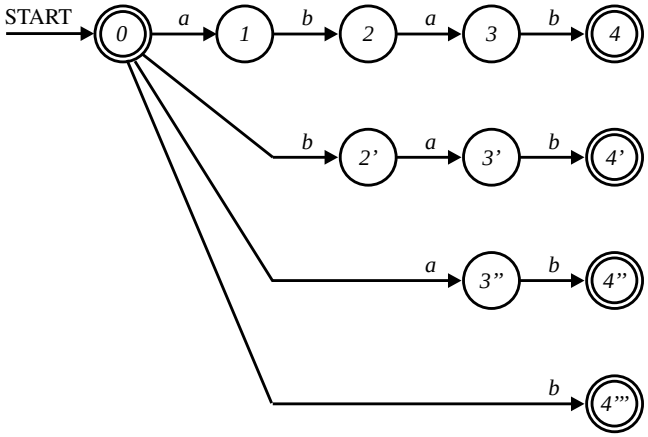


Figure 3.11: Transition diagram of suffix automaton M_1 accepting $\mathit{Suff}(abab)$ from Example 3.9

the transition diagram depicted in Fig. 3.12. Algorithm 3.8 yields, after the step 2., suffix automaton M_3 with the transition diagram depicted in Fig. 3.13. All automata M_1, M_2 and M_3 accepts language $\mathit{Suff}(abab)$ and therefore they should be equivalent. Let us show it.

As suffix automaton M_1 is nondeterministic, let us construct equivalent deterministic automaton M_1'. Transition table and transition diagram of

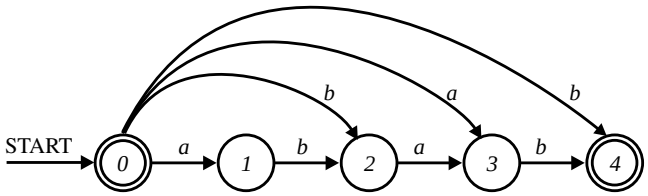


Figure 3.12: Transition diagram of suffix automaton M_2 accepting *Suff*(*abab*) from Example 3.9

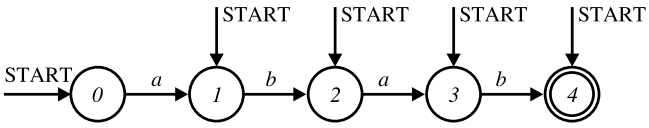


Figure 3.13: Transition diagram of nondeterministic suffix automaton M_3 accepting *Suff*(*abab*) from Example 3.9

suffix automaton M_1' are depicted in Fig. 3.14. Automaton M_1' can be min-

	a	b
0	$1'3''$	$2'4''$
$13''$		$24''$
$2'4''$	$3'$	
$24''$	3	
3		4
$3'$		$4'$

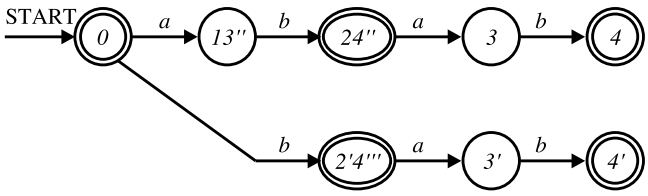


Figure 3.14: Transition table and transition diagram of deterministic suffix automaton M_1' from Example 3.9

imized as the states in pairs $\{(2, 4''), (2', 4''')\}, \{3, 3'\}$, and $\{4, 4'\}$ are equivalent. Transition diagram of minimized suffix automaton M_1' is depicted in Fig. 3.15.

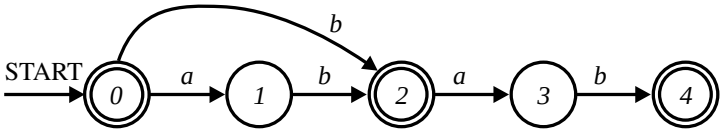


Figure 3.15: Transition diagram of deterministic suffix automaton M_1' after minimization from Example 3.9

Suffix automaton M_2 is also nondeterministic and after the determinization we obtain automaton M_2' having transition diagram depicted in Fig. 3.16.

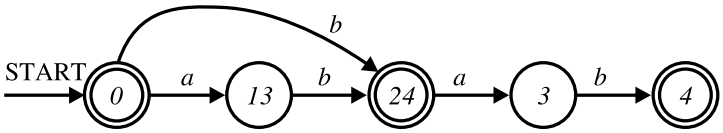


Figure 3.16: Transition diagram of deterministic suffix automaton M_2' from Example 3.9

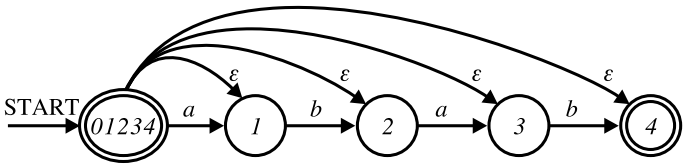
Suffix automaton M_3 has five initial states. Construction of equivalent automaton M_3 is shown step by step in Fig. 3.17.

Suffix automata M_1', M_2', and M_3' (see Figs 3.15, 3.16, 3.17) are obviously equivalent. □

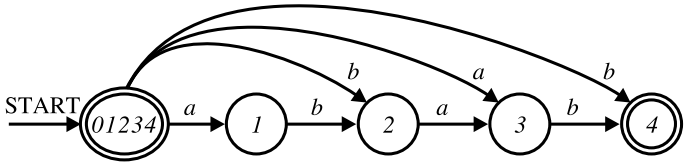
We can use the experience from the possible constructions of the suffix automaton in the practical algorithm.

Algorithm 3.10
Construction of a finite automaton accepting set *Suff*(x).
Input: String $x = a_1 a_2 \dots a_n$.
Output: Deterministic finite automaton $M = (Q, A, \delta, q_0, F)$ accepting set *Suff*(x).
Method:

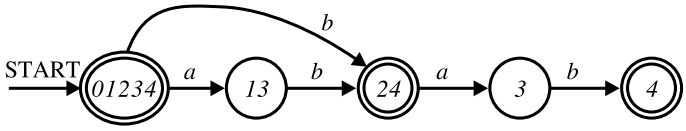
1. Construct finite automaton $M_1 = (Q_1, A, \delta_1, q_0, F_1)$ accepting string x and empty string:
   $Q_1 = \{0, 1, 2, \dots, n\}$,
   A is the set of all different symbols in x,
   $\delta_1(i-1, a_i) = i, i = 1, 2, \dots, n$,
   $q_0 = 0$,
   $F_1 = \{0, n\}$.
2. Insert additional transitions into automaton M_1 leading from initial state 0 to states $2, 3, \dots, n$:
   $\delta(0, a) = i$ if $\delta(i-1, a) = i$ for all $a \in A$, $i = 2, 3, \dots, n$.
   The resulting automaton is M_2.



a) Transition diagram of suffix automaton with one initial state and ε-transitions



b) Transition diagram of suffix automaton after removal of ε-transitions



c) Transition diagram of deterministic suffix automaton M_3'

Figure 3.17: Three steps of construction of deterministic suffix automaton M_3' from Example 3.9

3. Construct deterministic finite automaton M equivalent to automaton M_2. □

Definition 3.11 (Terminal state of the suffix automaton)
The final state of the suffix automaton having no outgoing transition is called *terminal* state. □

Definition 3.12 (Backbone of the suffix automaton)
The *backbone* of suffix automaton M for string x is the longest continuous sequence of states and transitions leading from the initial state to terminal state of M. □

Example 3.13
Let us construct the deterministic finite automaton accepting *Suff*$(abba) = \{abba, bba, ba, a, \varepsilon\}$ using Algorithm 3.10. Automaton $M_1 = (\{0,1,2,3,4\}, \{a,b\}, \delta_1, 0, \{0,4\})$ accepting strings $\{\varepsilon, abba\}$ has the transition diagram depicted in Fig. 3.18. Finite automaton $M_2 = (\{0,1,2,3,4\}, \{a,b\}, \delta_2, 0, \{0,4\})$ with additional transitions has the transition diagram depicted in Fig. 3.19. The final result of this construction is deterministic finite automaton $M = (\{0,14,2,23,3,4\}, \{a,b\}, \delta, \{0,4\})$ which transition table is shown in Table 3.1. d–subsets of automaton M are: $0, 14, 2, 23, 3, 4$. Tran-

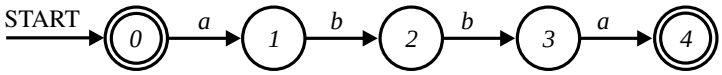


Figure 3.18: Transition diagram of finite automaton M_1 accepting string *abba* from Example 3.13

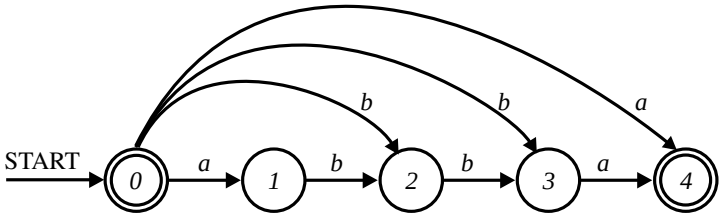


Figure 3.19: Transition diagram of nondeterministic finite automaton M_2 with additional transitions from Example 3.13

sition diagram of automaton M is depicted in Fig. 3.20. □

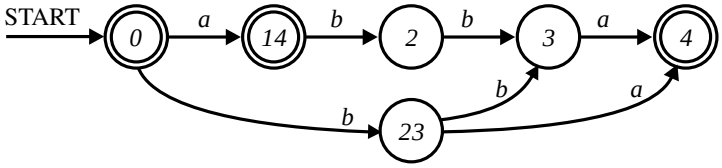


Figure 3.20: Transition diagram of deterministic finite automaton M accepting set $\mathit{Suff}(abba) = \{abba, bba, ba, a, \varepsilon\}$ from Example 3.13

3.3 Factor automaton

Factor automaton is in some sources called Directed Acyclic Word Graph (DAWG).

Having string $x = a_1 a_2 \dots a_n$, we can express set $Fact(x)$ (see Def. 1.3) using regular expressions:

1. $$\begin{aligned} R_{Fac1}(x) = \quad & (\dots((a_1+\varepsilon)a_2+\varepsilon)a_3+\dots+\varepsilon)a_n+\varepsilon \\ & +(\dots((a_1+\varepsilon)a_2+\varepsilon)a_3+\dots+\varepsilon)a_n \\ & +(\dots((a_1+\varepsilon)a_2+\varepsilon)a_3+\dots+\varepsilon)a_{n-1} \\ & +\dots \\ & + \qquad a_1, \end{aligned}$$
2. $$\begin{aligned} R_{Fac2}(x) = \varepsilon & + a_1(\varepsilon+a_2(\varepsilon+\dots+a_{n-1}(\varepsilon+a_n)\dots)) \\ & + a_1(\varepsilon+a_2(\varepsilon+\dots+a_{n-1}(\varepsilon+a_n)\dots)) \\ & + \qquad a_2(\varepsilon+\dots+a_{n-1}(\varepsilon+a_n)\dots) \\ & +\dots \\ & + \qquad\qquad\qquad\qquad a_n. \end{aligned}$$

	a	b
0	14	23
14		2
2		3
23	4	3
3	4	
4		

Table 3.1: Transition table of deterministic finite automaton M from Example 3.13

The first variant of the regular expression corresponds to that set $Fact(x)$ is exactly the set of all suffixes of all prefixes of x. The second variant corresponds to the fact that set $Fact(x)$ is exactly the set of all prefixes of all suffixes of x. It follows from these possibilities of understanding of both regular expressions that the combination of methods of constructing the prefix and suffix automata can be used for the construction of factor automata. For the first variant of the regular expression we can use Algorithms 3.6, 3.7, 3.8 and 3.10 as a base for the construction of suffix automata and to modify them in order to accept all prefixes of all suffixes accepted by suffix automata. Algorithm 3.14 makes this modification by setting all states final.

Algorithm 3.14
Construction of the factor automaton.
Input: String $x = a_1a_2 \dots a_n$.
Output: Finite automaton M accepting language $Fact(x)$.
Method:

1. Construct suffix automaton M_1 for string $x = a_1a_2 \dots a_n$ using any of Algorithms 3.6, 3.7, 3.8 or 3.10.
2. Construct automaton M_2 by setting all states of automaton M_1 final states.
3. Perform minimization of automaton M_2. The resulting automaton is automaton M. □

The resulting deterministic automaton need not be minimal and therefore the minimization takes place as the final operation.

Example 3.15
Let us have string $x = abbbc$. We construct the factor automaton accepting set $Fact(x)$ using all four possible ways of its construction. The first method is based on Algorithm 3.6. Factor automaton M_1 has after step 2. of Algorithm 3.6 transition diagram depicted in Fig. 3.21.

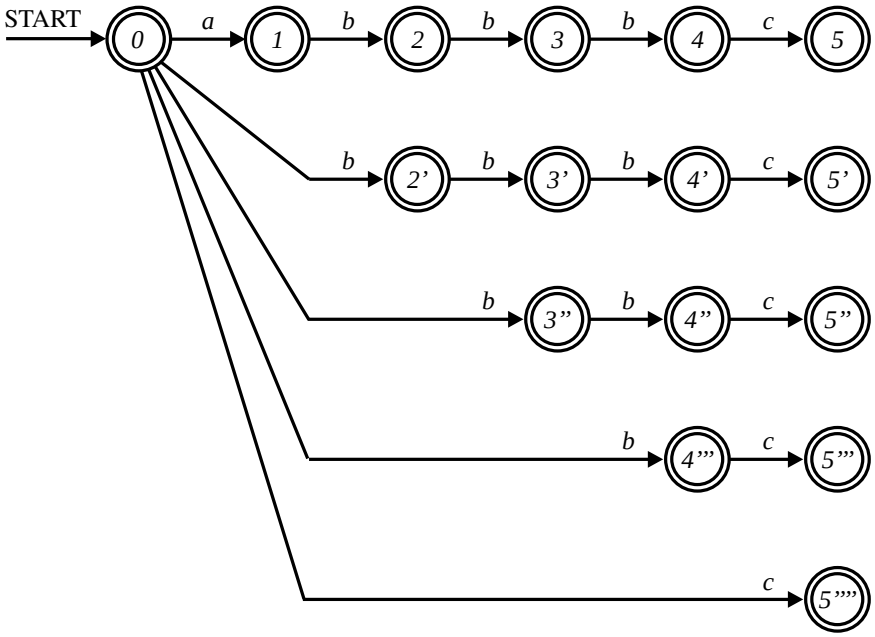


Figure 3.21: Transition diagram of factor automaton M_1 accepting set $Fact(abbbc)$ from Example 3.15

As factor automaton M_1 is nondeterministic, we do its determinisation. Transition table and transition diagram of deterministic factor automaton M_1' are depicted in Fig. 3.22. This automaton is not minimal because sets of states $\{4, 4'\}, \{5, 5', 5'', 5''', 5''''\}$ are equivalent. The transition diagram of minimal factor automaton M_1'' is depicted in Fig. 3.23. □

The second method of the construction of the factor automaton is based on Algorithm 3.7. Factor automaton M_2 after step 2 of Algorithm 3.7 has the transition diagram depicted in Fig. 3.24. Factor automaton M_2 is nondeterministic and therefore we do its determinisation. The resulting deterministic factor automaton M_2' is minimal and its transition table and transition diagram are depicted in Fig. 3.25.

	a	b	c
0	1	$2'3''4'''$	$5''''$
1		2	
2		3	
3		4	
4			5
5			
$2'3''4'''$		$3'4''$	$5'''$
$3'4''$		$4'$	$5''$
$4'$			$5'$
$5'$			
$5''$			
$5'''$			
$5''''$			

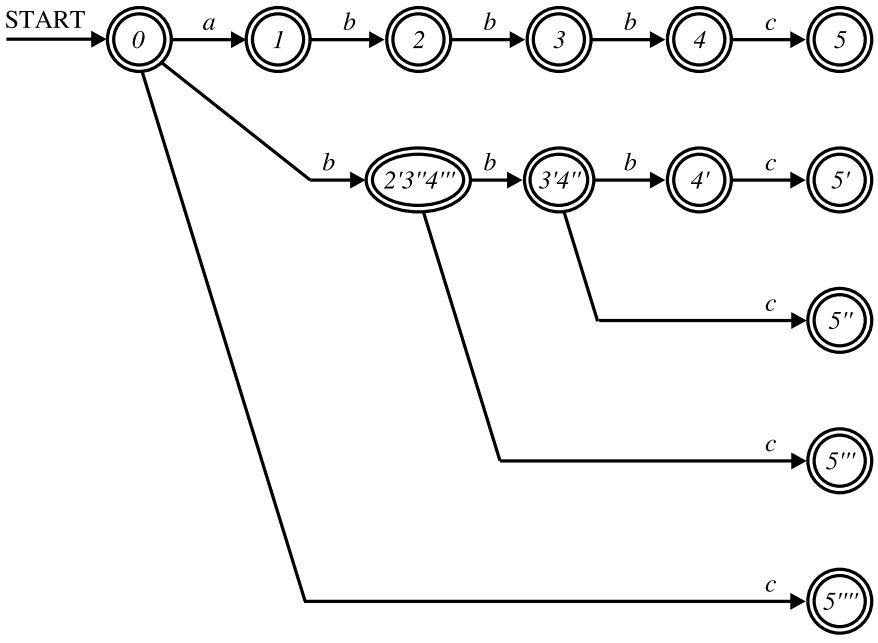


Figure 3.22: Transition table and transition diagram of deterministic factor automaton M_1' accepting set $Fact(abbbc)$ from Example 3.15

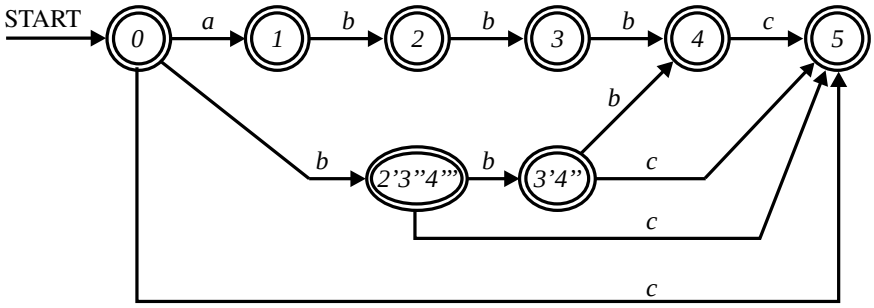


Figure 3.23: Transition diagram of minimal factor automaton M_1'' accepting set $Fact(abbbc)$ from Example 3.15

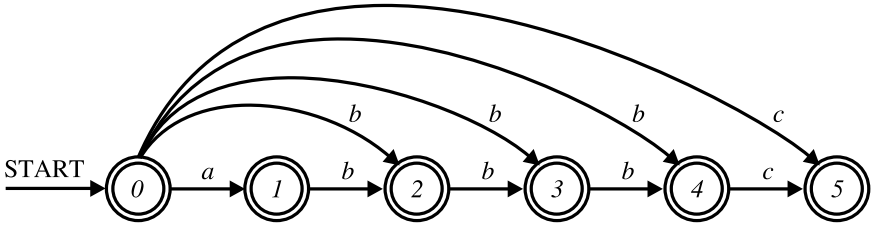


Figure 3.24: Transition diagram of nondeterministic factor automaton M_2 accepting set $Fact(abbbc)$ from Example 3.15

	a	b	c
0	1	234	5
1		2	
2		3	
3		4	
4			5
234		34	5
34		4	5
5			

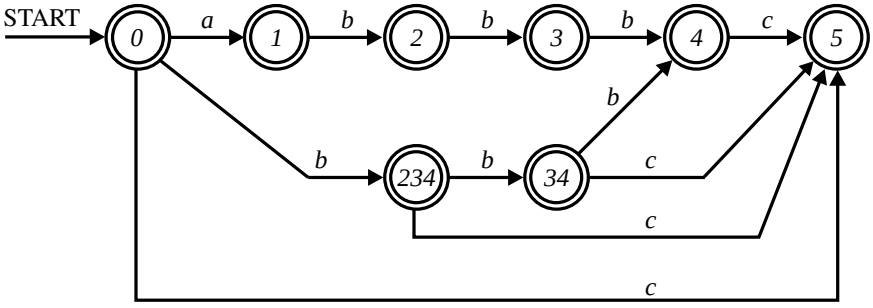


Figure 3.25: Transition table and transition diagram of deterministic factor automaton M_2' accepting set $Fact(abbbc)$ from Example 3.15

The third method of the construction of the factor automaton is based on Algorithm 3.8. Factor automaton M_3 has after step 2. of Algorithm 3.8 transition diagram depicted in Fig. 3.26. Automaton M_3 has more than

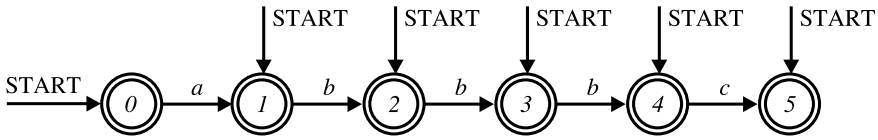


Figure 3.26: Transition diagram of factor automaton M_3 accepting set $Fact(abbbc)$ from Example 3.15

one initial state, therefore we transform it to automaton M_3' having just one initial state. Its transition table and transition diagram is depicted in Fig. 3.27. □

	a	b	c
012345	1	234	5
1		2	
2		3	
3		4	
4			5
234		34	5
34		4	
5			

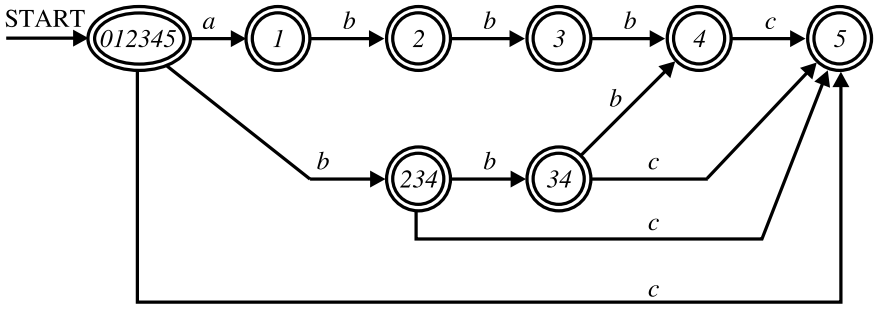


Figure 3.27: Transition table and transition diagram of factor automaton M_3' with just one initial state accepting set $Fact(abbbc)$ from Example 3.15

Note: We keep notions of terminal state and the backbone (see Defs. 3.11 and 3.12) also for factor automaton.

3.4 Parts of suffix and factor automata

We will need to use in some applications only some parts of suffix and factor automata instead of their complete forms. We identified three cases of useful

parts of both automata:

1. Backbone of suffix or factor automaton.
2. Front end of suffix or factor automaton.
3. Multiple front end of suffix or factor automaton.

3.4.1 Backbone of suffix and factor automata

The backbone of suffix or automaton for string x is such part of the automaton where all states and transitions which does not correspond to prefixes of x are removed. A general method of extraction of the backbone of suffix or factor automaton is the operation intersection.

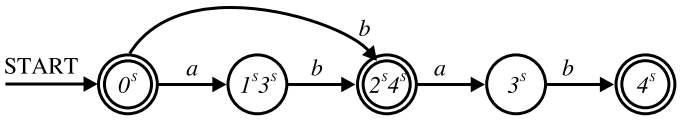
Algorithm 3.16
Construction of the backbone of suffix (factor) automaton.
Input: String $x = a_1a_2 \dots a_n$, deterministic suffix (factor) automaton $M_1 = (Q_1, A, \delta_1, q_{01}, F_1)$ for x, deterministic prefix automaton $M_2 = (Q_2, A, \delta_2, q_{02}, F_2)$ for x.
Output: Backbone $M = (Q, A, \delta, q_0, F)$ of suffix (factor) automaton $M_1 = (Q_1, A, \delta_1, q_{01}, F_1)$.
Method: Construct automaton M accepting intersection $\mathit{Suff}(x) \cap \mathit{Pref}(x)$ ($\mathit{Fact}(x) \cap \mathit{Pref}(x)$) using Algorithm 1.44. □

Example 3.17
Let us construct backbone of the suffix automaton for string $x = abab$. Transition diagrams of input automata M_1 and M_2 and output automaton M are depicted in Fig. 3.28. □

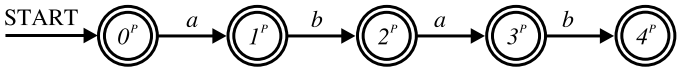
The resulting backbone is similar to input suffix automaton M_1. The only change is that transition from state 0^S to state 2^S4^S for input symbol b is removed. Moreover, we can see that the resulting backbone in Example 3.17 is equivalent to the input prefix automaton M_2. The important point of this construction is that d-subsets of suffix automaton M_1 are preserved in the resulting automaton M. This fact will be useful in some applications described in next Chapters.
The algorithm for extraction of the backbone of the suffix or factor automaton is very simple and straightforward. Nevertheless it can be used for extraction of backbones of suffix and factor automata for set of strings and for approximate suffix and factor automata as well.

3.4.2 Front end of suffix or factor automata

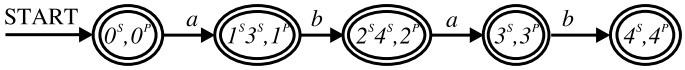
A front end of a suffix or factor automaton for string x is a finite automaton accepting prefixes of factors of string x having limited length which is strictly less than the length of string x.



a) Suffix automaton M_1 accepting *Suff*(*abab*)



b) Prefix automaton M_2 accepting *Pref*(*abab*)



c) Backbone M of suffix automaton M_1

Figure 3.28: Construction of the backbone of suffix automaton M_1 from Example 3.17

For the construction of a front end part of factor automaton we adapt Algorithm 1.40 for transformation of nondeterministic finite automaton to a deterministic finite automaton. The adaptation consists of two points:

1. To append information on the minimal distance of a state of an automaton from its initial state. The minimal distance is, in this case, the minimal number of transitions, which are necessary to reach the state in question from the initial state.
2. To stop construction of deterministic automaton as soon as all states having desired limited distance from the initial state are constructed.

Definition 3.18
Let $M = (Q, A, \delta, q_0, F)$ be an acyclic finite automaton accepting language $L(M)$. *Front end* of automaton M for given limit h is a minimal deterministic finite automaton M_h accepting at least all prefixes of strings from language $L(M)$ having length less or equal to h. This language will be denoted by L_h. □

Algorithm 3.19
Transformation of an acyclic nondeterministic finite automaton to a deterministic finite automaton with states having distance from the initial state less or equal to given limit.

Input: Nondeterministic acyclic finite automaton $M = (Q, A, \delta, q_0, F)$, limit h of maximal distance.
Output: Deterministic finite automaton $M_h = (Q_h, A, \delta_h, q_{0h}, F_h)$ such that $L_h(M) = L_h(M_h)$.
Method:

1. Set $Q_h = \{\{(q_0, 0)\}\}$ will be defined, state $\{q_0, 0\}$ will be treated as unmarked.
2. If each state in Q_h is marked then continue with step 5.
3. If there is no unmarked state (q, l) in Q_h, where l is less than h then continue with step 5.
4. An unmarked state (q, l) will be choosen from Q_h and the following operations will be executed:
   (a) $\delta_h((q, l), a) = (q', l + 1)$ for all $a \in A$, where $q' = \cup\delta(p, a)$ for all $a \in A, p \in q$,
   (b) if $(q', l') \in Q_h$ then $Q_h = Q_h \cup (q', min(l+1, l'))$ and $\delta_h((q, l), a) = (q', min(l + 1, l')$,
   (c) state (q, l) will be marked,
   (d) continue with step 2.
5. $q_{0h} = \{q_0, 0\}$.
6. $F_h = \{(q, l) : (q, l) \in Q_h, q \cap F \neq \emptyset\}$. □

Example 3.20
Let us have string $x = abab$. Construct front end of the factor automaton for string x of length $h = 2$. Transition diagram of nondeterministic factor automaton M for string x is depicted in Fig. 3.29. Transition diagram of

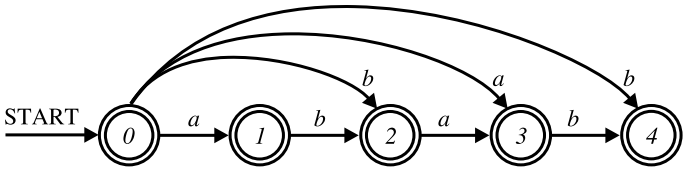


Figure 3.29: Transition diagram of nondeterministic factor automaton M for string $x = abab$ from Example 3.20

the front end of deterministic factor automaton M_h for $h = 2$ is depicted in Fig. 3.30. □

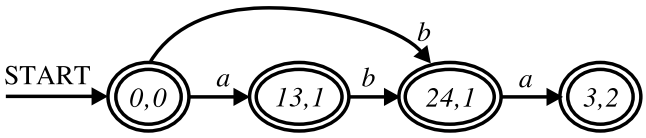


Figure 3.30: Transition diagram of the front end of deterministic factor automaton M_2 from Example 3.20

3.4.3 Multiple front end of suffix and factor automata

Let us recall definition of multiple state of deterministic finite automaton (see Def. 1.42). Multiple front end of a suffix or factor automaton is the part of them containing multiple states only. For the construction of such part of suffix or factor automaton we again adapt Algorithm 1.40 for transformation of nondeterministic finite automaton to a deterministic finite automaton. The adaption is very simple:
If some state constructed during determinisation is simple state then we omit it.

Algorithm 3.21
Construction of multiple states part of suffix or factor automaton.
Input: Nondeterministic suffix or factor automaton $M = (Q, A, \delta, q_0, F)$.
Output: Part $M' = (Q', A, \delta', q_0', F')$ of the deterministic factor automaton for M containing only multiple states (with exception of the initial state).
Method:

1. Set $Q' = \{\{q_0\}\}$ will be defined, state $\{q_0\}$ will be treated as unmarked.
2. If all states in Q' are marked then continue with step 4.
3. An unmarked state q will be choosen from Q' and the following operations will be executed:
   (a) $\delta'(q, a) = \cup\, \delta(p, a)$ for all $p \in q$ and for all $a \in A$,
   (b) if $\delta'(q, a)$ is a multiple state then $Q' = Q' \cup \delta'(q, a)$,
   (c) the state $q \in Q'$ will be marked,
   (d) continue with step 2.
4. $q_0' = \{q_0\}$.
5. $F' = \{q : q \in Q', q \cap F \neq \emptyset\}$. □

Example 3.22
Let us have string $x = abab$ as in Example 3.20. Construct multiple front end of the factor automaton for string x. Transition diagram of nondeterministic factor automaton M for string x is depicted in Fig. 3.29. Transition diagram of multiple front end M' of factor automaton M is depicted in Fig. 3.31. □

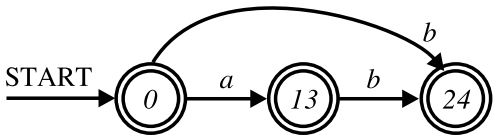


Figure 3.31: Transition diagram of the multiple states part of deterministic factor automaton M' from Example 3.22

3.5 Subsequence automata

The set of subsequences of string $x = a_1a_2 \dots a_n$ (see Def. 1.4) can be described by the following regular expression:

$R_{Sub}(x) = (a_1 + \varepsilon)(a_2 + \varepsilon) \dots (a_n + \varepsilon)$.

Therefore the next algorithm is based on the insertion of ε–transition.

Algorithm 3.23
The construction of a subsequence automaton accepting set $Sub(x)$.
Input: String $x = a_1a_2 \dots a_n$.
Output: Deterministic subsequence automaton $M = (Q, A, \delta, q_0, F)$ accepting set $Sub(x)$.
Method:

1. Construct finite automaton $M_1 = (Q_1, A, \delta_1, q_0, F_1)$ accepting all prefixes of string x:
   $Q_1 = \{0, 1, 2, \dots, n\}$,
   A is the set of all different symbols in x,
   $\delta_1(i-1, a_i) = i,\ i = 1, 2, \dots, n$,
   $q_0 = 0$, $F_1 = \{0, 1, 2, \dots, n\}$.
2. Insert ε–transitions into automaton M_1 leading from each state to its next state. Resulting automaton $M_2 = (Q, A, \delta_2, q_0, F_1)$, where $\delta_2 = \delta_1 \cup \delta'$, where $\delta'(i-1, \varepsilon) = i,\ i = 1, 2, \dots, n$.
3. Replace all ε–transitions by non–ε–transitions. The resulting automaton is M_3.
4. Construct deterministic finite automaton M equivalent to automaton M_3. All its states will be final states. □

Example 3.24
Let us construct the deterministic finite automaton accepting set $Sub(abba) = \{\varepsilon, a, b, ab, ba, aa, bb, aba, bba, abb, abba\}$. Let us mention, that strings aa, aba are subsequences of $x = abba$ but not its factors. Using Algorithm 3.23 we construct the subsequence automaton. Automaton $M_1 = (\{0, 1, 2, 3, 4\}, \{a, b\}, \delta_1, 0, \{0, 1, 2, 3, 4\})$ accepting all prefixes of string $abba$ has transition diagram depicted in Fig. 3.35.

Finite automaton M_2 with inserted ε–transitions has transition diagram depicted in Fig 3.32. Nondeterministic finite automaton M_3 after the elim-

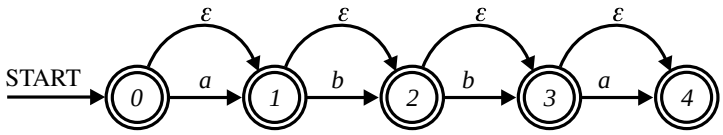


Figure 3.32: Transition diagram of automaton M_2 with ε–transitions accepting all subsequences of string *abba* from Example 3.24

ination of ε–transitions has transition diagram depicted in Fig. 3.33. The

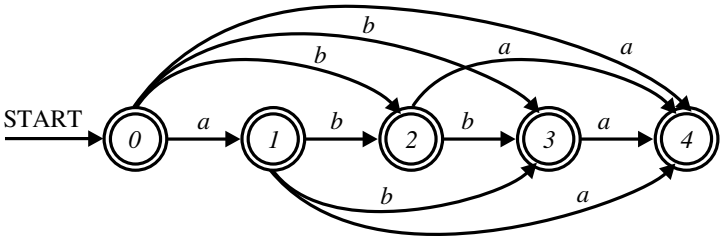


Figure 3.33: Transition diagram of nondeterministic finite automaton M_3 accepting set $Sub(abba)$ after elimination of the ε–transitions from Example 3.24

final result of this construction is deterministic finite automaton (subsequence automaton) M. Its transition table is shown in Table 3.2. Transition

	a	b
0	14	23
14	4	23
23	4	3
3	4	
4		

Table 3.2: Transition table of automaton M from Example 3.24

diagram of automaton M is depicted in Fig. 3.34. □

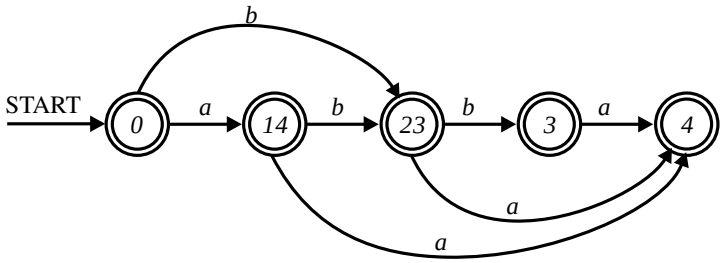


Figure 3.34: Transition diagram of deterministic subsequence automaton M accepting set $Sub(abba)$ from Example 3.24

3.6 Factor oracle automata

The factor oracle automaton for given string x accepts all factors of x and possibly some subsequences of x.

Factor oracle automaton is similar to the factor automaton, but it has always $n+1$ states, where $n = |x|$. It is possible to construct factor oracle automaton from factor automaton. This construction is based on the notion of *corresponding states* in a factor automaton.

Definition 3.25
Let M be the factor automaton for string x and q_1, q_2 be different states of M. Let there exist two sequences of transitions in M:
$(q_0, x_1) \vdash^* (q_1, \varepsilon)$, and
$(q_0, x_2) \vdash^* (q_2, \varepsilon)$.
If x_1 is a suffix of x_2 and x_2 is a prefix of x then q_1 and q_2 are *corresponding states*. □

The factor oracle automaton can be constructed by merging the corresponding states.

Example 3.26
Let us construct the deterministic finite automaton accepting set $Fact(abba) = \{\varepsilon, a, b, ab, bb, ba, abb, bba, abba\}$ using Algorithm 3.7. Automaton $M_1 = (\{0,1,2,3,4\}, \{a,b\}, \delta_1, 0, \{0,1,2,3,4\})$ accepting all prefixes of the string $abba$ has the transition diagram depicted in Fig. 3.35. Finite au-

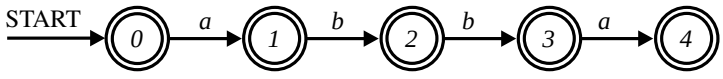


Figure 3.35: Transition diagram of finite automaton M_1 accepting set of all prefixes of the string $abba$ from Example 3.26

tomaton M_2 with inserted ε–transitions has the transition diagram depicted

in Fig. 3.36. Nondeterministic finite automaton M_3 after the elimination of ε–transitions has the transition diagram depicted in Fig. 3.37.

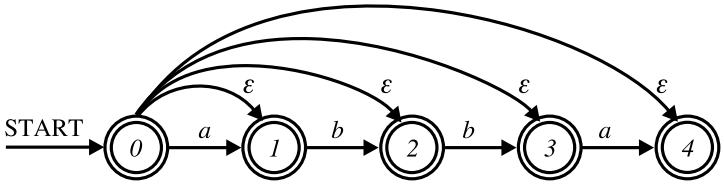


Figure 3.36: The transition diagram of automaton M_2 with ε–transitions accepting all factors of string *abba* from Example 3.26

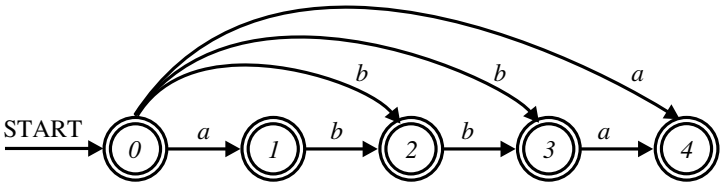


Figure 3.37: Transition diagram of nondeterministic factor automaton M_3 after the elimination of ε–transitions from Example 3.26

The final result of this construction is deterministic factor automaton M. Its transition table is shown in Table 3.3. The transition diagram of

	a	b
0	14	23
14		2
2		3
23	4	3
3	4	
4		

Table 3.3: The transition table of the automaton M from Example 3.26

automaton M is depicted in Fig. 3.38. The corresponding states in this automaton are: 2 and 23. If we make this two states equivalent, then we obtain factor oracle automaton $Oracle(abba)$ with the transition diagram depicted in Fig. 3.39. The language accepted by the automaton is:

$$\begin{aligned} L(Oracle(abba)) &= \{\varepsilon, a, b, ab, bb, ba, abb, bba, abba, aba\} \\ &= Fact(abba) \cup \{aba\}. \end{aligned}$$

String *aba* is not factor of *abba* but it is its subsequence. □

The approach used in Example 3.26 has this drawback: the intermediate result is a factor automaton and its number of states is limited by $2n - 2$

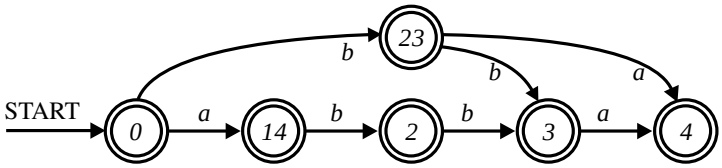


Figure 3.38: Transition diagram of factor automaton M accepting set $Fact(abba)$ from Example 3.26

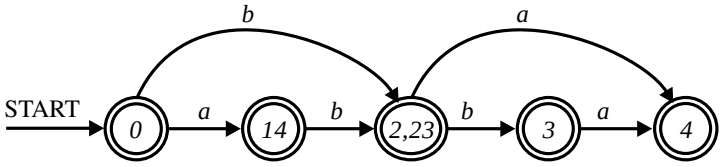


Figure 3.39: Transition diagram of the $Oracle(abba)$ from Example 3.27

while the number of states of factor oracle automaton is always equal to $n+1$, where n is the length of string x. Fortunately, a factor oracle automaton can be constructed directly during the determinization of nondeterministic factor automaton. In this case, it is necessary to fix the identification of corresponding states.
Interpretation A of Definition 3.25: Using our style of the numbering of states, the identification of corresponding states can be done as follows:

Two states
$p = \{i_1, i_2, \ldots, i_{n1}\}, q = \{j_1, j_2, \ldots, j_{n2}\}$
are corresponding states provided that set of states of nondeterministic factor automaton is ordered and the lowest states are equal which means that $i_1 = j_1$.

Example 3.27
During the determinisation of nondeterministic factor automaton M_3 (see Fig. 3.37) we will identify states 2 and 23 as corresponding states and the factor oracle automaton having transition diagram depicted in Fig. 3.39 can be constructed directly. □

A problem can appear during the construction of factor oracle automaton by merging of corresponding states of the respective factor automaton. The problem is that the resulting factor oracle automaton can be nondeterministic. Let us show this problem in next Example.

Example 3.28
Let us have text $T = abbcabcd$. The construction of factor oracle automaton using merging of states of the respective factor automaton is shown step by step in Fig. 3.40.

We can see, in Fig. 3.40 d), that the factor oracle automaton resulting

by merging of corresponding states $\{236, 26\}$ and $\{47, 4\}$ of the factor automaton having transition diagram depicted in Fig. 3.40 c) is nondeterministic. The nondeterminism is caused by transition $\delta(236, c) = \{47, 7\}$. The transition table of the nondeterministic factor oracle automaton depicted in Fig. 3.40 d) is Table 3.4. The result of the determinization of this factor

	a	b	c	d
0	15	236	47	8
15		236		
236		3	47, 7	
3			47	
47	5			8
5		6		
6			7	
7				8
8				

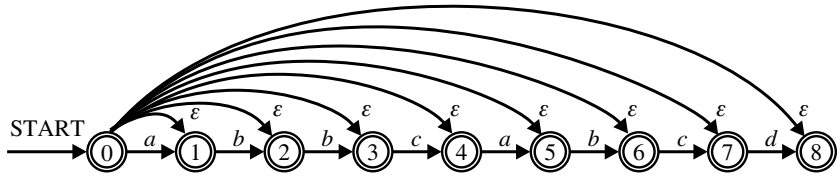
Table 3.4: Transition table of the nondeterministic factor oracle automaton having transition diagram depicted in Fig. 3.40 d) (see Example 3.28)

oracle automaton is the deterministic factor oracle automaton having the transition diagram depicted in Fig. 3.40 e) and Table 3.5 as the transition table. We can see in this table, that states $\{47\}$ and $\{47, 7\}$ are equivalent. This fact is expresed in Fig. 3.40 e). □

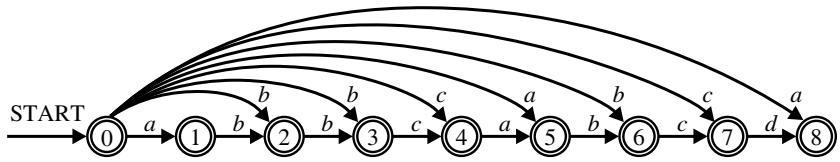
	a	b	c	d
0	15	236	47	8
15		236		
236		3	47, 7	
3			47	
47	5			8
47, 7	5			8
5		6		
6			7	
7				8
8				

Table 3.5: Transition table of the deterministic factor oracle automaton from Example 3.28; let us note, that states $\{47\}$ and $\{47, 7\}$ are equivalent

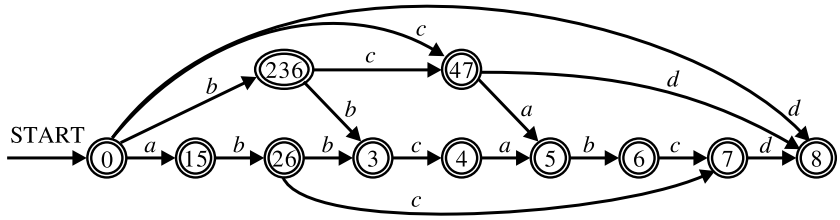
Let us discuss the problem of the nondeterminism of factor oracle automaton after merging of states.



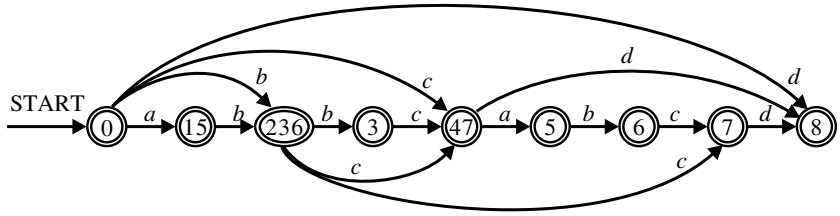
a) transition diagram of the nondeterministic factor automaton with ε-transitions



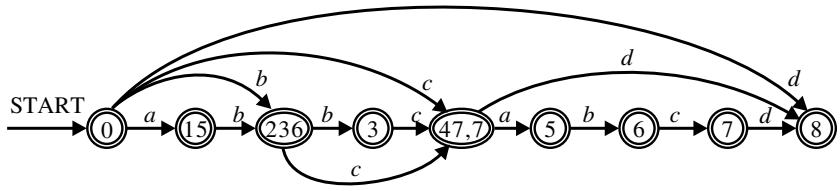
b) transition diagram of the nondeterministic factor automaton after removal of ε-transitions



c) transition diagram of the deterministic factor automaton



d) transition diagram of the nondeterministic factor oracle automaton



e) transition diagram of the deterministic factor oracle automaton

Figure 3.40: Construction of factor oracle automaton from Example 3.28

Let $\delta(q, a) = \{q_1, q_2\}$, for some $a \in A$, be the "nondeterministic" part of the transition function. State q_2 has greater depth than state q_1. The factor oracle automaton is homogenous automaton and due to its construction the d-subset (see Def. 1.41) of q_2 is a subset of d-subset of q_1. If it holds that $d\text{-subset}(q_1) = d\text{-subset}(q_1) \cup d\text{-subset}(q_2)$. It follows from this, that

$\delta(\{q_1\}, a) = \delta(\{q_1, q_2\}, a)$ for all $a \in A$.

The practical consequence of this reasoning is, that in the case of nondeterminism shown above, it is enough to remove "longer" transition. More precisely: $\delta(q, a) = \{q_1\}$.

There is possible for some texts to construct factor oracle automaton having less transitions than the factor oracle automaton constructed by merging corresponding states according to Definition 3.25. Let us show the possibility using an example.

Example 3.29
Let text the be $T = abcacdace$. The construction of factor oracle automaton using merging of states according to Definition 3.25 is shown step by step in Fig. 3.41. Moreover another principle of merging states of respective factor automaton (see Fig. 3.41c) can be used. This is merging of states (3,58,358). □

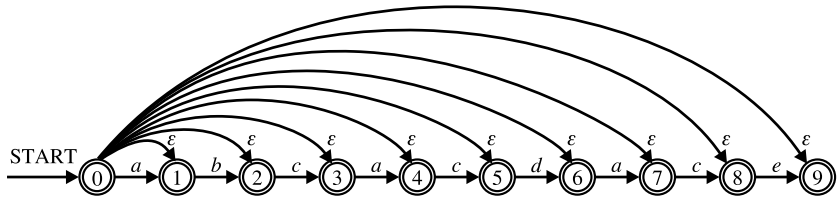
The principle used in Example 3.29 is based on the following interpretation B of Definition 3.25:
States having d-subsets:
$p = \{i_1, i_2, \ldots, i_{n1}\}, q = \{j_1, j_2, \ldots, j_{n2}\}$ are corresponding states for $|p| > |q|$ when $q \subset p$.

From this follows that states 358 and 58 are corresponding states and states 3 and 358 are corresponding states according to the interpretation A of Definition 3.25.

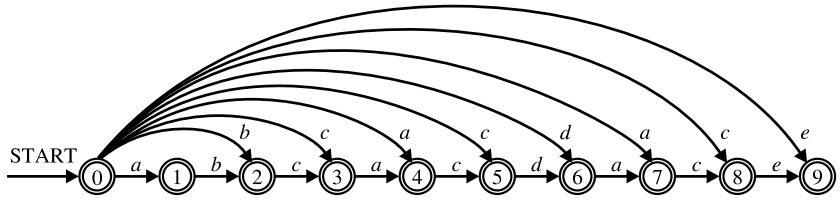
Factor oracle accepts language $L(Oracle(x))$ for string x. Language L contains set $Fact(x)$ and moreover some subsequences. Let us do characterisation of the language accepted by factor oracles.

The main idea behind the method of this characterisation is an operation called contraction. The contraction consists in removal of some string from x starting with a repeating factor a continuing by gap to some of its repetition. Let us show this principle using an example.

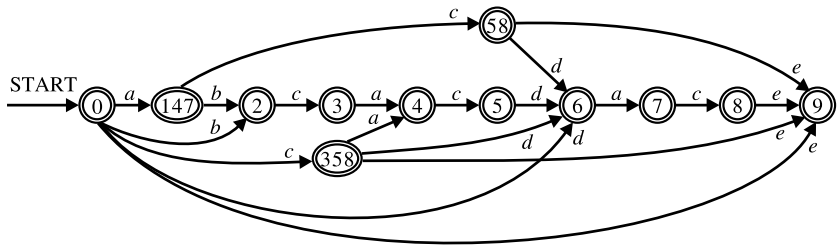
Example 3.30
Let text be $T = gaccattctc$. We start by construction of factor automaton for text T. Transition diagram of nondeterministic factor automaton M_N is depicted in Fig. 3.42a. Transition table of deterministic factor



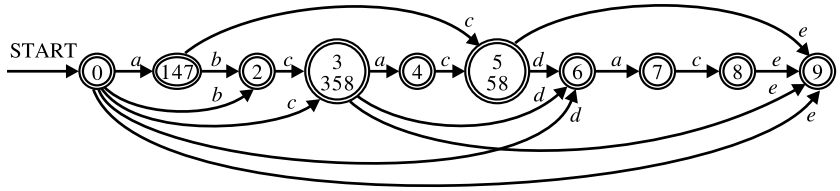
a) transition diagram of the nondeterministic factor automaton with ε-transitions



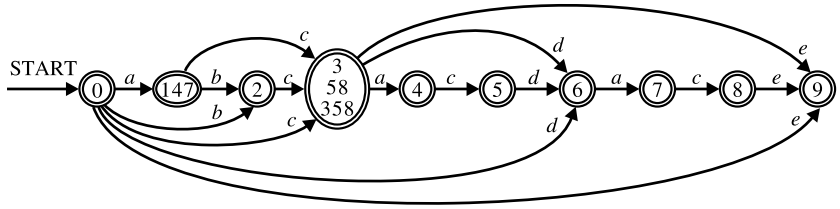
b) transition diagram of the nondeterministic factor automaton after removal of ε-transitions



c) transition diagram of the deterministic factor automaton



d) transition diagram of the factor oracle automaton after merging pair of states (3,358) and (5,58) has 8 external transition



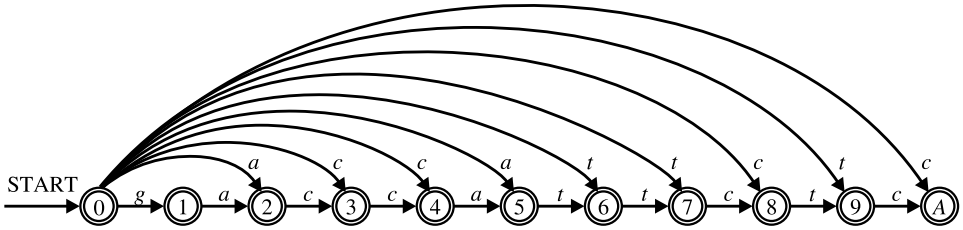
e) transition diagram of deterministic factor oracle automaton after merging states (3,58,358) has 7 external transitions

Figure 3.41: Construction of factor oracle automaton and its "optimised" variant from Example 3.29

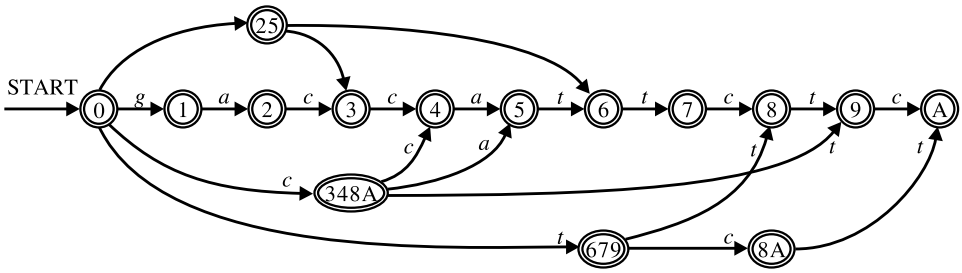
automaton M_D is shown in Table 3.6. Transition diagram of deterministic factor automaton M_D is depicted in Fig. 3.42b. The corresponding states are these pairs of states:

$(2, 25), (3, 348A), (6, 679), (8, 8A)$.

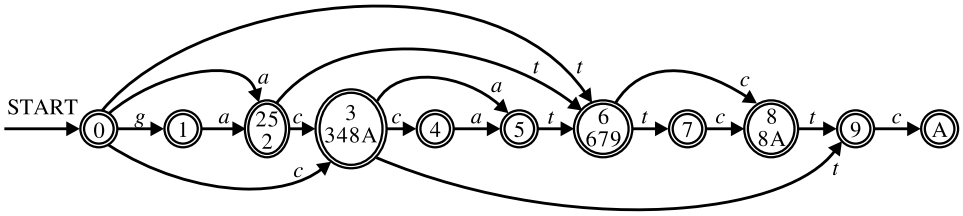
On the base of this correspondence we can construct factor oracle automaton M_O having transition diagram depicted in Fig. 3.42c.



a) transition diagram of nondeterministic factor automaton M_N



b) transition diagram of deterministic factor automaton M_D



c) transition diagram of factor oracle automaton M_O

Figure 3.42: Transition diagrams of automata M_N, M_D and M_O for text $T = gaccattctc$ from Example 3.30

Using factor automaton M_D we can construct repetition table R shown in Table 3.7. □

Using the repetition table we can construct a set of contractions. Set *Contr* of contractions is a set of pairs (i, j) where

i is the starting position of contraction,

j is the first position behind the contraction.

The contractions are closely related to the repetition of factors in text T. The contraction consists in removing the first occurrence of some repeating

	a	c	g	t
0	25	$348A$	1	679
1	2			
25		3		6
2		3		
3		4		
$348A$	5	4		9
4	5			
5				6
6				7
679		$8A$		7
7		8		
8				9
$8A$				9
9		A		
A				

Table 3.6: Transition table of deterministic factor automaton M_D for text $T = gaccattctc$ from Example 3.30

d-subset	Factor	Repetitions
25	a	$(2, F), (5, G)$
$348A$	c	$(3, F), (4, S), (8, G), (A, G)$
679	t	$(6, F), (7, S), (9, G)$
$8A$	tc	$(8, F), (A, S)$

Table 3.7: Repetition table R for text $T = gaccattctc$ from Example 3.30

factor and the gap starting behind it and ending just before some next occurrence of it. The contraction can be combined provided that the repetition of some factor is a repetition with gap. It cannot be combined in case of its overlapping and if one contraction contains another as a substring.

Using contractions described above, we obtain set of pairs $Contr(T)$ for text T.

Example 3.31
Let text be $T = gaccattctc$ as in Example 3.30. The set of contractions based on repetition table R is:

$Contr(T) = \{(2, 5), (3, 4), (3, 8), (3, A), (6, 7), (6, 9), (7, 9)\}$.
Let us list set $SC(T)$ of all strings created from T using this contractions and containing also string T.

$$
\begin{array}{lll}
T & = gaccattctc = gaccattctc & (*) \\
C\{(7,9)\} & = gaccatt\not{t}\not{c}tc = gaccattc & \\
C\{(3,4)\} & = ga\not{c}attctc = gacattctc & (*) \\
C\{(3,4),(7,9)\} & = ga\not{c}att\not{t}\not{c}tc = gacattc & \\
C\{(3,10)\} & = ga\not{c}\not{c}\not{a}\not{t}\not{t}\not{c}\not{t}c = gac & \\
C\{(6,7)\} & = gacca\not{t}tctc = gaccatctc & (*) \\
C\{(6,9)\} & = gaccat\not{t}\not{c}tc = gaccatc & \\
C\{(3,4),(6,7)\} & = ga\not{c}ca\not{t}tctc = gacatctc & (*) \\
C\{(3,4),(6,9)\} & = ga\not{c}ca\not{t}\not{c}tc = gacatc & \\
C\{(2,5)\} & = g\not{a}\not{c}\not{c}attctc = gattctc & (*) \\
C\{(2,5),(7,9)\} & = g\not{a}\not{c}\not{c}att\not{t}\not{c}tc = gattc & \\
C\{(2,5),(6,7)\} & = g\not{a}\not{c}\not{c}a\not{t}tctc = gatctc & (*) \\
C\{(2,5),(6,9)\} & = g\not{a}\not{c}\not{c}a\not{t}\not{c}tc = gatc & \\
C\{(3,8)\}) & = ga\not{c}\not{c}\not{a}\not{t}\not{t}ctc = gactc & (*)
\end{array}
$$

□

The set of strings $SC(T)$ created by contractions can contain some string which are substring of other elements of set SC. Such strings can be from SC removed because they are redundant.

Example 3.32
The set $SCO(T)$ for text $T = gaccattctc$ from Example 3.30 contains after optimisation these strings (marked in Example 3.31 by *):
$SCO(T) = \{gaccattctc, gacattctc, gaccatctc, gacatctc, gattctc, gatctc, gactc\}$.
Language accepted by factor oracle M_0 for T is:
$L(M_0) = Fact(SCO(T))$. □

3.7 The complexity of automata for parts of strings

The maximum state and transition complexities of prefix, suffix, factor, subsequence, and factor oracle automata are summarized in Table 3.8. The length of the string is always equal to n. $|A|$ is the size of alphabet A. The factor automaton having maximal state and transition complexities is depicted in Fig. 3.23. This complexity is reached for strings $ab^{n-2}c$. The complexity of the suffix automaton is in many cases the same as of the factor automaton. But in some cases the factor automaton can have less states than the suffix automaton for the same string. The reason for thisis fact that some factor automata may be minimized because they have all states final.

Example 3.33
Let us have text $T = abb$. We will construct the suffix automaton accepting $Suff(abb)$ and the factor automaton accepting $Fact(abb)$. The nondeterministic and deterministic suffix automata have transition diagrams depicted

Type of automaton	No. of states	No. of transitions
Prefix automaton	$n+1$	n
Suffix automaton	$2n-2$	$3n-4$
Factor automaton	$2n-2$	$3n-4$
Subsequence automaton	$n+1$	$\|A\|.n$
Factor oracle automaton	$n+1$	$2n-1$

Table 3.8: Maximum state and transition complexities of automata accepting parts of string

in Fig. 3.43. The nondeterministic and deterministic factor automata have

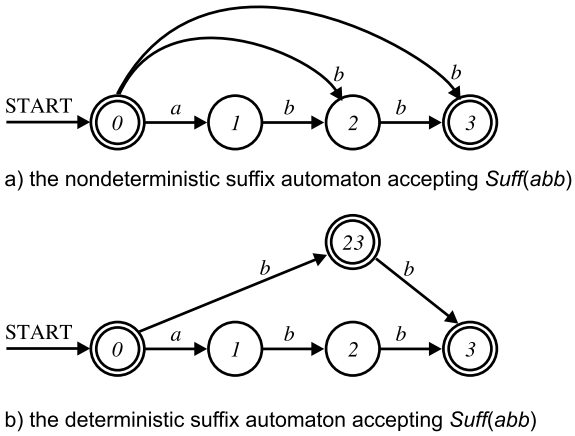


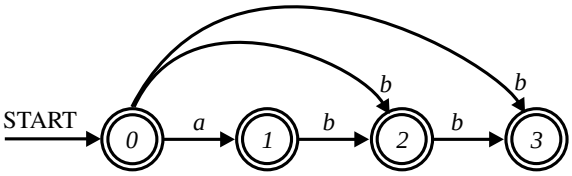
Figure 3.43: Transition diagrams of the suffix automata for string $x = abb$ from Example 3.33

transition diagrams depicted in Fig. 3.44. The deterministic factor automaton can be minimized as states 2 and 23 are equivalent. This is not true for the deterministic suffix automaton as 23 is a final state and 2 is not a final state. Therefore the minimal factor automaton accepting $Fact(abb)$ has transition diagram depicted in Fig. 3.45. □

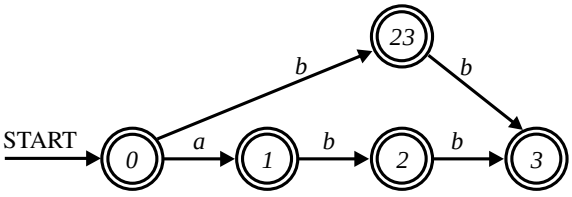
The reader can verify, that for string $x = ab^n, n > 1$, the factor automaton has less states than the suffix automaton.

3.8 Automata for parts of more than one string

All finite automata constructed above in this Chapter can be used as a base for the construction of the same types of automata for a finite set of strings. Construction of a finite automaton accepting set $Pref(S)$ (see Def. 1.6) where S is a finite set of strings from A^+ we will do in the similar



a) the nondeterministic factor automaton accepting *Fact*(*abb*)



b) the deterministic factor automaton accepting *Fact*(*abb*)

Figure 3.44: Transition diagrams of the factor automata for string $x = abb$ from Example 3.33

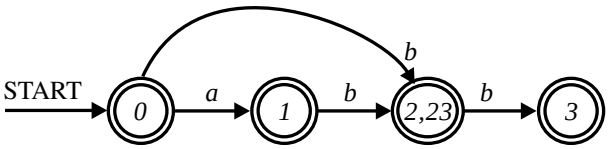


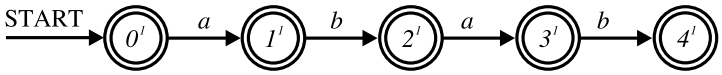
Figure 3.45: Transition diagram of the minimal factor automaton accepting $Fact(abb)$ from Example 3.33

way as for one string in Algorithm 3.4.

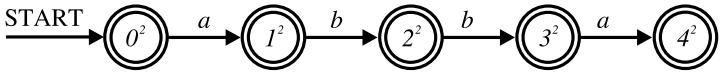
Algorithm 3.34
Construction of a finite automaton accepting set $Pref\,S, S \subset A^+$.
Input: A finite set of strings $S = \{x_1, x_2, \ldots, x_{|s|}\}$.
Output: Prefix automaton $M = (Q, A, \delta, q_0, F)$ accepting set $Pref(S)$.
Method:

1. Construct finite automata $M_i = (Q_i, A_i, \delta_i, q_{0i}, F_i)$ accepting set $Pref(x_i)$ for $i = 1, 2, \ldots, |S|$ using Algorithm alg@ktis3-01.
2. Construct deterministic finite automaton $M = (Q, A, \delta, q_0, F)$ accepting set $Pref(S) = Pref(x_1) \cup Pref(x_2) \cup \ldots \cup Pref(x_{|S|})$. □

Example 3.35
Let us construct the prefix automaton for set of strings $S = \{abab, abba\}$. Finite automata M_1 and M_2 have transition diagrams depicted in Fig. 3.46.



a) Transition diagram of finite automaton M_1 accepting set *Pref (abab)*



b) Transition diagram of finite automaton M_2 accepting set *Pref (abba)*

Figure 3.46: Transition diagrams of automata M_1 and M_2 from Example 3.35

Prefix automaton M accepting set $Pref(abab, abba)$ has transition diagram depicted in Fig. 3.47. □

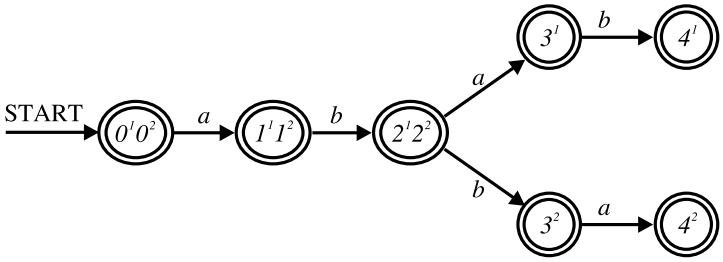


Figure 3.47: Transition diagram of prefix automaton accepting set $Pref(abab, abba)$ from Example 3.35

Construction of suffix and factor automata for a finite set of strings we will do in the similar way as for one string (see Sections 3.2 and 3.3). One of principles of their construction is formalized in the following Algorithm for suffix and factor automata. An X–automaton, in the next Algorithm, means suffix or factor automaton.

Algorithm 3.36
Construction of X–automaton for a finite set of strings.
Input: Finite set of strings $S = \{x_1, x_2, \ldots, x_{|S|}\}$.
Output: Deterministic X–automaton $M = (Q, A, \delta, q_0, F)$ for set S.
Method:

1. Construct X–automata $M_1, M_2, \ldots, M_{|S|}$ with ε–transitions (see Fig. 3.8) for all strings $x_1, x_2, \ldots, x_{|S|}$.
2. Construct automaton M_ε accepting language $L(M_\varepsilon) = L(M_1) \cup L(M_2) \cup \ldots \cup L(M_{|S|})$.

3. Construct automaton M_N by removing ε–transitions.
4. Construct deterministic finite automaton M equivalent to automaton M_N. □

Example 3.37
Let us construct the factor automaton for set of strings $S = \{abab, abba\}$. First, we construct factor automata M_1 and M_2 for both strings in S. Their transition diagrams are depicted in Figs 3.48 and 3.49, respectively.

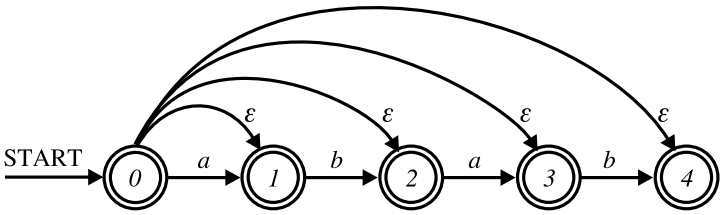


Figure 3.48: Transition diagram of factor automaton M_1 accepting $Fact(abab)$ from Example 3.37

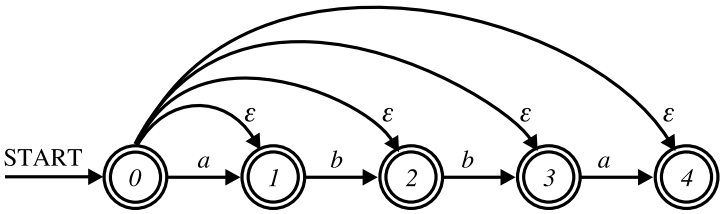


Figure 3.49: Transition diagram of factor automaton M_2 accepting $Fact(abba)$ from Example 3.37

In the second step we construct automaton M_ε accepting language $L(M) = Fact(abab) \cup Fact(abba)$. Its transition diagram is depicted in Fig. 3.50.

In the third step we construct automaton M_N by removing ε–transitions. Its transition diagram is depicted in Fig. 3.51.

The last step is the construction of deterministic factor automaton M. Its transition table is shown in Table 3.9.

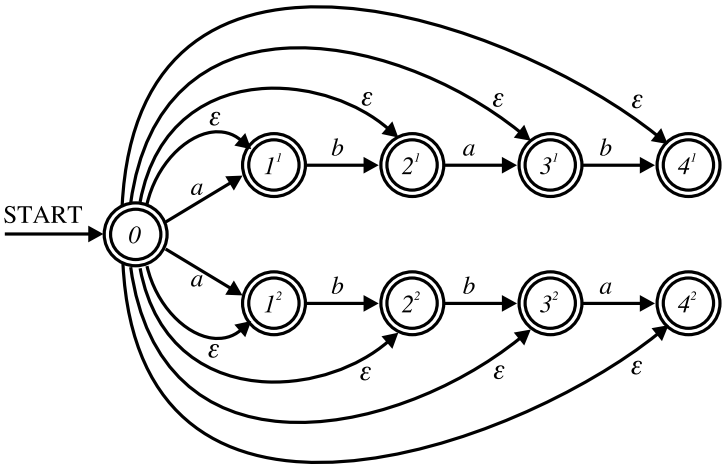


Figure 3.50: Transition diagram of factor automaton M_ε accepting set $Fact(abab) \cup Fact(abba)$ from Example 3.37

	a	b
0	$1^1 1^2 3^1 4^2$	$2^1 2^2 3^2 4^1$
$1^1 1^2 3^1 4^2$		$2^1 2^2 4^1$
$2^1 2^2 3^2 4^1$	$3^1 4^2$	3^2
$2^1 2^2 4^1$	3^1	3^2
$3^1 4^2$		4^1
3^1		4^1
3^2	4^2	
4^1		
4^2		

Table 3.9: Transition table of automaton M from Example 3.37

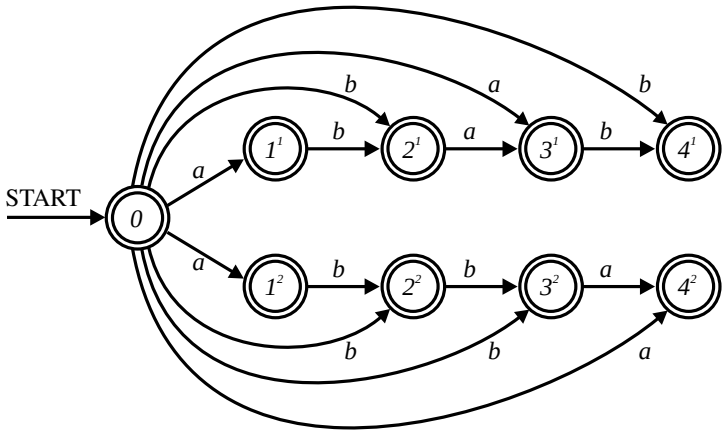


Figure 3.51: Transition diagram of factor automaton M_N accepting set $Fact(abab) \cup Fact(abba)$ from Example 3.37

Transition diagram of resulting deterministic factor automaton M is depicted in Fig. 3.52. □

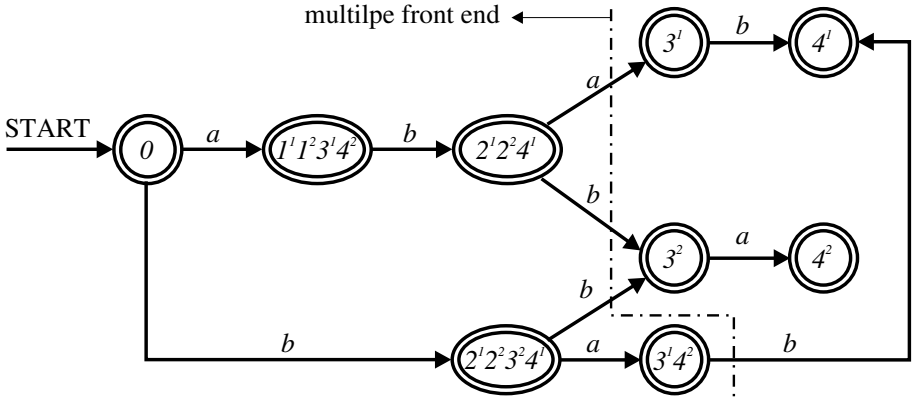


Figure 3.52: Transition diagram of deterministic factor automaton M accepting set $Fact(abab) \cup Fact(abba)$ from Example 3.37

3.9 Automata accepting approximate parts of a string

Finite automata constructed above in this Chapter are accepting *exact* parts of string (prefixes, suffixes, factors, subsequences). It is possible to use the lessons learned from their constructions for the construction of automata accepting *approximate* parts of string. The main principle of algorithms accepting approximate parts of string x is:

1. Construct finite automaton accepting set:
   $Approx(x) = \{y : D(x, y) \leq k\}$.
2. Construct finite automaton accepting approximate parts of string using principles similar to that for the construction of automata accepting the exact parts of string.

We show this principle using Hamming distance in the next example.

Definition 3.38
Set $H_k(P)$ of all strings similar to string P is:
$H_k(P) = \{X : X \in A^*, D_H(X, P) \leq k\}$.
where $D_H(X, P)$ is the Hamming distance. □

Example 3.39
Let string be $x = abba$. We construct approximate prefix automaton for Hamming distance $k = 1$. For this purpose we use a modification of Algorithm 2.5. The modification consists in removing the selfloop in the initial state q_0. After that we make all state final states. Transition diagram of resulting "Hamming" prefix automaton is depicted in Fig. 3.53.

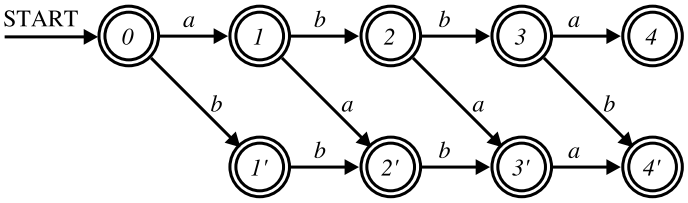


Figure 3.53: Transition diagram of the "Hamming" prefix automaton accepting $APref(abba)$ for Hamming distance $k = 1$ from Example 3.39

Example 3.40
Let string be $x = abba$. We construct approximate factor automaton using Hamming distance $k = 1$.

1. We use Algorithm 3.10 modified for factor automaton (see Algorithm 3.14). For the construction of finite automaton accepting string x and all strings with Hamming distance equal to 1, we use a modification of Algorithm 2.5. The modification consists in removing the selfloop in initial state q_0. Resulting finite automaton has transition diagram depicted in Fig. 3.54.

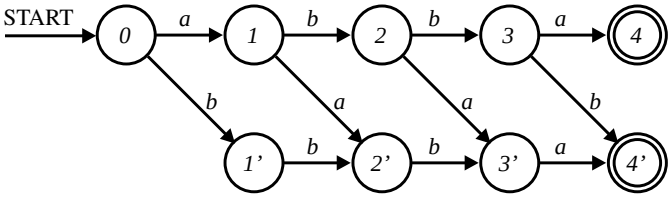


Figure 3.54: Transition diagram of the "Hamming" automaton accepting $H_1(abba)$ with Hamming distance $k = 1$ from Example 3.40

2. We use the principle of inserting the ε–transitions from state 0 to states 1,2,3 and 4. Moreover, all states are fixed as final states. Transition diagram with inserted ε–transition is depicted in Fig. 3.55.
3. We replace ε–transitions by non–ε–transitions. The resulting automaton has transition diagram depicted in Fig. 3.56.
4. The final operation is the construction of the equivalent deterministic finite automaton. Its transition table is shown in Table 3.10.

The transition diagram of the resulting deterministic approximate factor automaton is depicted in Fig. 3.57. All states of it are final states. □

Example 3.41
Let us construct backbone of the Hamming factor automaton from Exam-

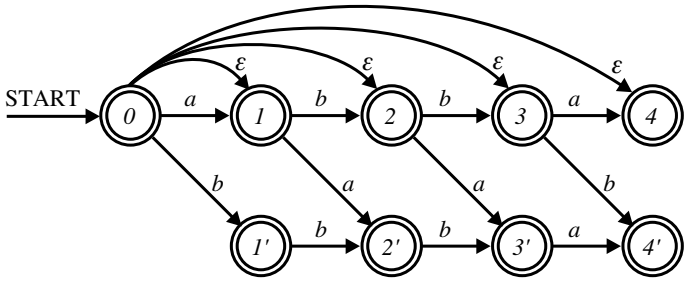


Figure 3.55: Transition diagram of the "Hamming" factor automaton with ε–transitions inserted and final states fixed from Example 3.40

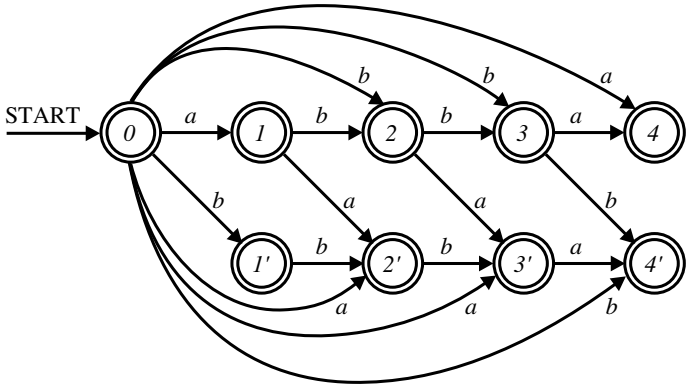


Figure 3.56: Transition diagram of the "Hamming" factor automaton after removal of ε–transitions from Example 3.40

ple 3.40. The construction of the backbone consists in intersection of Hamming factor automaton having transition diagram depicted in Fig. 3.57 and Hamming prefix automaton having transition diagram depicted in Fig. 3.53. The result of this intersection is shown in Fig. 3.58. In this automaton pairs of states:

$((2'4', 2'^P), (32'4', 2'^P))$ and

$((3'4', 3'^P), (3', 3'^P))$

are equivalent.

The backbone equivalent to one depicted in Fig. 3.58 we obtain using the following approach. We can recognize, that set of states of the Hamming factor automaton (see Fig. 3.57):

(2'4',32'4') and (3'4,3',3'4')

are equivalent. After minimization we obtain Hamming factor automaton depicted in Fig. 3.59. The backbone of the automaton we obtain by removal of transition drawn by the dashed line. □

	a	b
0	$142'3'$	$231'4'$
$142'3'$	$2'4'$	$23'$
$231'4'$	$3'4'$	$32'4'$
$2'4'$		$3'$
$23'$	$3'4'$	3
$3'4'$	$4'$	
$32'4'$	4	$3'4'$
$3'$	$4'$	
3	4	$4'$
$4'$		
4		

Table 3.10: Transition table of the deterministic "Hamming" factor automaton from Example 3.40

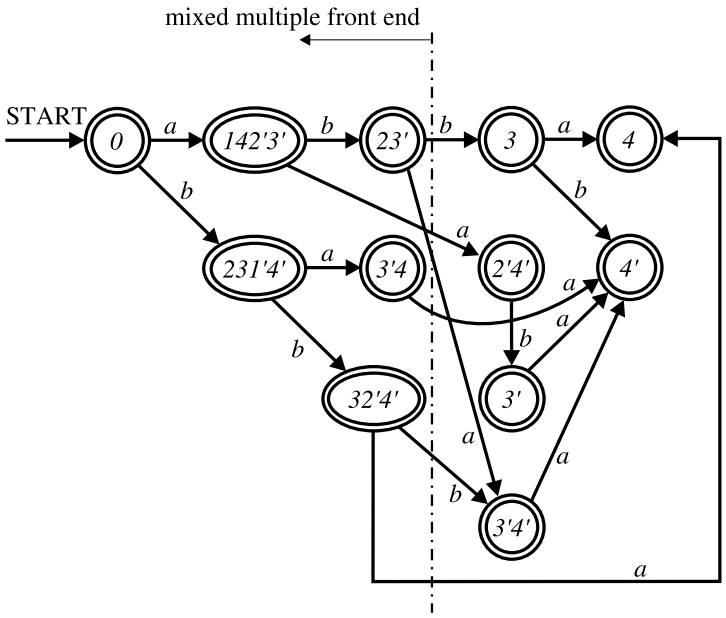


Figure 3.57: Transition diagram of the deterministic "Hamming" approximate factor automaton for $x = abba$, Hamming distance $k = 1$ from Example 3.40

	a	b
$(0, 0^P)$	$(142'3', 1^P)$	$(231'4', 1'^P)$
$(142'3', 1^P)$	$(2'4', 2'^P)$	$(23', 2^P)$
$(231'4', 1'^P)$		$(32'4', 2'^P)$
$(23', 2^P)$	$(3'4', 3'^P)$	$(3, 3^P)$
$(2'4', 2'^P)$		$(3', 3'^P)$
$(32'4', 2'^P)$		$(3'4', 3'^P)$
$(3'4', 3'^P)$	$(4', 4'^P)$	
$(3, 3^P)$	$(4, 4^P)$	$(4', 4'^P)$
$(3', 3'^P)$	$(4', 4'^P)$	

Table 3.11: Transition table of the backbone of Hamming factor automaton for string $x = abba$

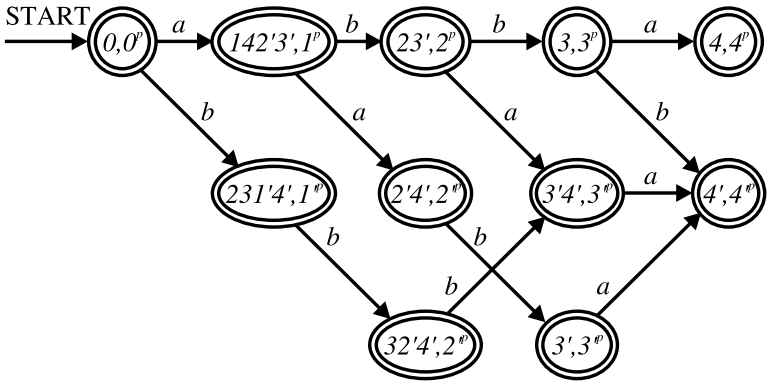


Figure 3.58: Transition diagram of the backbone of Hamming factor automaton for string $x = abba$, from Example 3.41

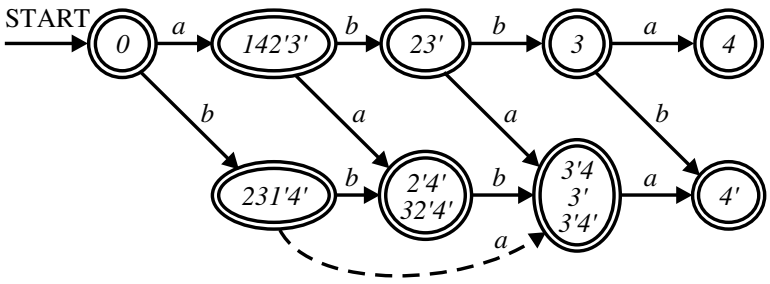


Figure 3.59: Minimized Hamming factor automaton for string $x = abba$ from Example 3.41

4 Borders, repetitions and periods

4.1 Basic notions

Definition 4.1 (Proper prefix)
The proper prefix is any element of $Pref(x)$ not equal to x. □

Definition 4.2 (Border)
Border of string $x \in A^+$ is any proper prefix of x, which is simultaneously its suffix. The set of all borders of string x is $bord(x) = (Pref(x) \setminus \{x\}) \cap Suff(x)$. The longest border of x is $Border(x)$. □

Definition 4.3 (Border of a finite set of string)
Border of set of strings $S = \{x_1, x_2, \ldots, x_{|S|}\}$ is any proper prefix of some $x_i \in S$ which is the suffix of some $x_j \in S$, $i, j \in < 1, |S| >$. The set of all borders of the set S is

$$mbord(S) = \bigcup_{i=1}^{|S|} \bigcup_{j=1}^{|S|} (Pref(x_i) \setminus \{x_i\}) \cap Suff(x_j).$$

The longest border which is the suffix of $x_i, i \in < 1, |S| >$ belongs to the set $mBorder(S)$.

$$mBorder(S) = \{\, u_i : u_i \in mbord(S), u_i \text{ is the longest suffix of } x_i, \\ i \in < 1, |S| > \}.$$ □

Definition 4.4 (Period)
Every string $x \in A^+$ can be written in the form:

$x = u^r v$,

where $u \in A^+$ and $v \in Pref(u)$. The length of the string u, $p = |u|$ is a *period* of the string x, r is an *exponent* of the string x and u is a *generator* of x. The shortest period of string x is $Per(x)$. The set of all periods of x is periods (x). String x is pure periodic if $v = \varepsilon$. □

Definition 4.5 (Normal form)
Every string $x \in A^+$ can be written in the *normal form*:

$x = u^r v$,

where $p = |u|$ is the shortest period $Per(x)$, therefore r is the highest exponent and $v \in Pref(u)$. □

Definition 4.6 (Primitive string)
If string $x \in A^+$ has the shortest period equal to its length, then we call it the *primitive string*. □

Let us mention that for primitive string x holds that $Border(x) = \{\varepsilon\}$.

Definition 4.7 (Border array)
The *border array* $\beta[1..n]$ of string $x \in A^+$ is a vector of the lengths of the longest borders of all prefixes of x:
$\beta[i] = |Border(x[1..i])|$ for $i = 1, 2, \ldots, n$. □

Definition 4.8 (Border array of a finite set of strings)
The *mborder* array $m\beta[1..n]$ of a set of strings $S = \{x_1, x_2, \ldots, x_{|S|}\}$ is a vector of the longest borders of all prefixes of strings from S:
$m\beta[h] = |mBorder(\{x_1, x_2, \ldots, x_{i-1}, x_i[1..j], x_{i+1}, \ldots x_{|S|}\})|$ for
$i \in \langle 1, |S|\rangle, j \in \langle 1, |x_i|\rangle$.

The values of variable h are used for the labelling of states of finite automaton accepting set S.

$$h \leq \sum_{l=1}^{|S|} |x_l|.$$

□

Definition 4.9 (Exact repetition in one string)
Let T be a string, $T = a_1a_2 \ldots a_n$ and $a_i = a_j, a_{i+1} = a_{j+1}, \ldots, a_{i+m} = a_{j+m}, i < j, m \geq 0$. String $x_2 = a_ja_{j+1} \ldots a_{j+m}$ is an exact *repetition* of string $x_1 = a_ia_{i+1} \ldots a_m$. x_1 or x_2 are called repeating factors in text T.
□

Definition 4.10 (Exact repetition in a set of strings)
Let S be a set of strings, $S = \{x_1, x_2, \ldots, x_{|S|}\}$ and $x_{pi} = x_{qj}, x_{pi+1} = x_{qj+1}, \ldots, x_{pm} = x_{qm}, k \neq l$ or $k = l$ and $i < j, m \geq 0$.
String $x_{qj}x_{qj+1} \ldots x_{qm}$ is an *exact repetition* of string $x_{pi}x_{pi+1} \ldots x_{pm}$.
□

Definition 4.11 (Aproximate repetition in one string)
Let T be a string, $T = a_1a_2 \ldots a_n$ and $D(a_ia_{i+1} \ldots a_{i+m}, a_ja_{j+1} \ldots a_{j+m'}) \leq k$, where $m, m' \geq 0$, D is a distance, $0 < k < n$. String $a_ja_{j+1} \ldots a_{j+m'}$ is an *approximate repetition* of string $a_ia_{i+1} \ldots a_{i+m}$. □

The approximate repetition in the set of strings can be defined in the similar way.

Definition 4.12 (Type of repetition)
Let $x_2 = a_ja_{j+1} \ldots a_{j+m}$ be an exact or approximate repetition of $x_1 = a_ia_{i+1} \ldots a_{i+m'}, i < j$, in one string.
Then if $j - i < m$ then the repetition is with an *overlapping* (O),
if $j - i = m$ then the repetition is a *square* (S),
if $j - i > m$ then the repetition is with a *gap* (G). □

4.2 Borders and periods

The Algorithms in this Section show, how to find borders of a string and its periods.

The main topic of our interest is pattern matching. The main goal is to find all occurences of a given pattern in a text. This is very simple when the pattern is a primitive string. There are no possible two occurences of the primitive pattern with an overlapping. This is not true for patterns having nonempty *Border*. In this situation there are possible two or more occurences of a pattern with an overlapping. Such situation for pattern p is visualised in Fig. 4.1. Fig. 4.2 shows "cluster" of occurences of "highly"

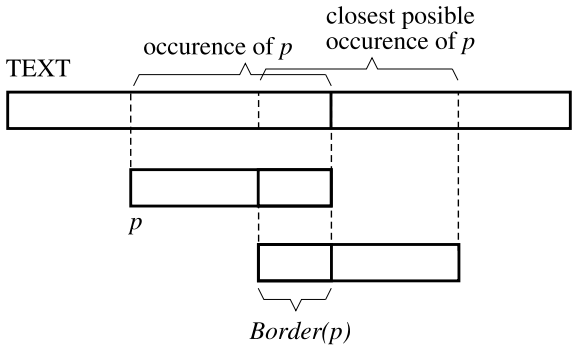


Figure 4.1: Visualisation of two possible occurences of pattern p with an overlapping

periodic pattern $p = ababa$. For pattern p holds:
$bord(p) = \{\varepsilon, a, aba\}$,
$Border(p) = aba$,
$periods(p) = \{2, 4\}$,
$Per(p) = 2$,
$p = (ab)^2a$ is the normal form of $p, r = 2$.

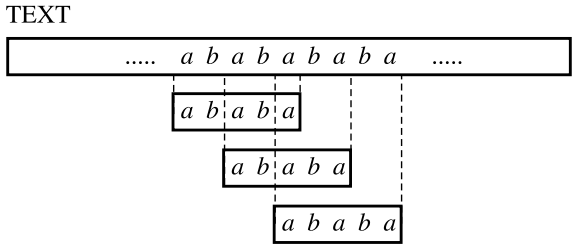


Figure 4.2: Visualisation of a "cluster" of occurences of "highly" periodic pattern $p = ababa$

The distance of two consecutive occurences in the depicted "cluster" is given by $Per(p) = 2$. The maximum number of occurences with overlapping in such cluster is given by the exponent and is equal to 3 in our case. It is equal to the exponent for pure periodic patterns.

4.2.1 Computation of borders

We use a deterministic suffix automaton and analysis of its backbone for the computation of borders for given string.

Algorithm 4.13
Computation of borders.
Input: String $x \in A^+$.
Output: $bord(x), Border(x)$.
Method:

1. Construct deterministic suffix automaton M for the string x.
2. Do analysis of the automaton M:
   (a) set $bord := \{\varepsilon\}$,
   (b) find all sequences of transitions on the backbone of the suffix automaton M leading from the initial state to some final state but the terminal one,
   (c) if the labelling of some sequence of transitions is x_i then $bord(x) := bord(x) \cup \{x_i\}$ for all $i = 1, 2, \ldots, h$, where h is the number of sequences of transitions found in step b)
3. Select the longest element y of the set $bord(x)$ and set $Border(x) := y$.

□

Note: The dashed lines will be used for the parts of suffix and factor automata which are out of the backbone.

Example 4.14
Let us compute $bord(x)$ and $Border(x)$ for string $x = ababab$ using Algorithm 4.13. Suffix automaton M has the transition diagram depicted in Fig. 4.3.

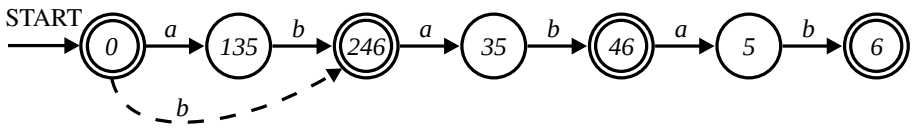


Figure 4.3: Transition diagram of the suffix automaton for string $x = ababab$ from Example 4.14

There are three sequences of transitions in M in question:
$0 \xrightarrow{\varepsilon} 0$
$0 \xrightarrow{a} 135 \xrightarrow{b} 246$
$0 \xrightarrow{a} 135 \xrightarrow{b} 246 \xrightarrow{a} 35 \xrightarrow{b} 46$
Then $bord(x) = \{\varepsilon, ab, abab\}$ and $Border(x) = abab$. □

Example 4.15
Let us compute *bord*(x) and *Border*(x) for string $x = abaaba$ using Algorithm 4.13. Suffix automaton M has the transition diagram depicted in Fig. 4.4.

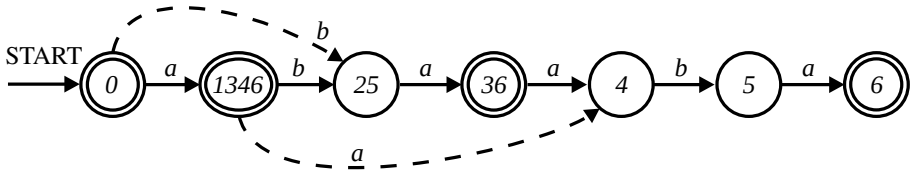


Figure 4.4: Transition diagram of the suffix automaton for string $x = abaaba$ from Example 4.15

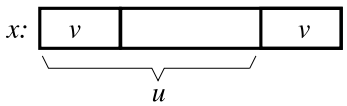
There are three sequences of transitions in M in question:
$0 \xrightarrow{\varepsilon} 0$
$0 \xrightarrow{a} 1346$,
$0 \xrightarrow{a} 1346 \xrightarrow{b} 25 \xrightarrow{a} 36$.
Then *bord*(x) $= \{\varepsilon, a, aba\}$, *Border*($x$) $= aba$. □

4.2.2 Computation of periods

The computation of periods for a given string can be based on the relation between borders and periods. Such relation is expressed in the following Lemma.

Lemma 4.16
Let $x \in A^+$ be a string having border v then $p = |x| - |v|$ is a period of string x. □

Proof:
We can display string x in the form:



Therefore we can write x in the form $x = uv$, where $v \in$ *Pref*(u) and thus $p = |u|$ is a period of x. □

Algorithm 4.17
Computation of periods.
Input: String $x \in A^+$.
Output: Set periods $(x) = \{p_1, p_2, \ldots, p_h\}$ $h \geq 0$, the shortest period *Per*(x).

Method:

1. Compute $bord(x) = \{x_1, x_2, \ldots, x_h\}$ and $Border(x)$ using Algorithm 4.13.
2. Compute $periods(x) = \{p_1, p_2, \ldots, p_h\}$, where $p_i = |x| - |x_i|,\ 1 \leq i \leq h,\ x_i \in bord(x)$.
3. Compute $Per(x) = |x| - |Border(x)|$. □

Example 4.18
Let us compute set of periods and the longest period $Per(x)$ for string $x = ababab$. The set of borders $bord(x)$ and $Border(x)$ computed in Example 4.14 are:

$bord(ababab) = \{\varepsilon, ab, abab\}$,
$Border(ababab) = abab$.

The resulting set $periods(x) = \{6, 4, 2\}$.
The shortest period $Per(x) = 2$.
The normal form of string $x = (ab)^3$. □

4.3 Border arrays

Let us remind the definition of the border array (see Def. 4.7). Borders and periods are related to situtations when a pattern is found. Border arrays are related to a situation when some prefix of the pattern is found. Figure 4.5 shows a situation when prefix u of pattern p is found. After that a mismatch occurs. There is possible that next occurence of prefix u of pattern p may occur again and it depends on the length of $Border(u)$.

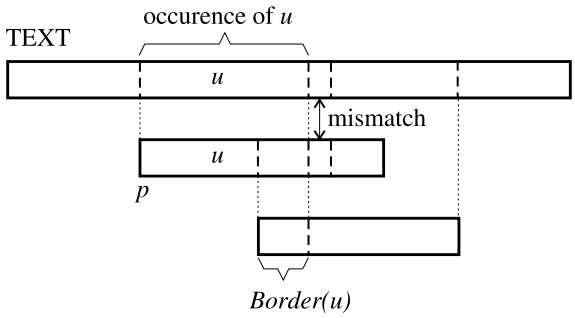


Figure 4.5: Visualisation of two possible occurences of prefix u of pattern p

The next Algorithm is devoted to the computation of a border array of one string.

Algorithm 4.19
Computing of border array.
Input: String $x \in A^+$.

Output: Border array $\beta[1..n]$, where $n = |x|$.

Method:

1. Construct nondeterministic factor automaton M_1 for x.
2. Construct equivalent deterministic factor automaton M_2 and preserve d–subsets.
3. Initialize all elements of border array $\beta[1..n]$ by the value zero.
4. Do analysis of multiple d–subsets of the deterministic factor automaton for states on the backbone from left to right:
   If the d–subset has the form $i_1, i_2, \ldots, i_h$ (this sequence is ordered), then set $\beta[j] := i_1$ for $j = i_2, i_3, \ldots, i_h$. □

Example 4.20
Let us construct the border array for string $x = abaababa$ (Fibonacci string f_5) using Algorithm 4.19. Nondeterministic factor automaton M_1 has transition diagram depicted in Fig. 4.6. Equivalent deterministic factor automaton

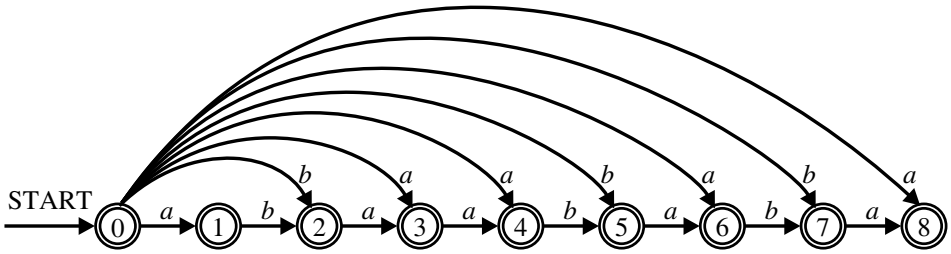


Figure 4.6: Transition diagram of nondeterministic factor automaton M_1 for string $x = abaababa$ from Example 4.20

M_2 with preserved d–subsets has the transition diagram depicted in Fig. 4.7. Now we do analysis of d–subsets starting with the d–subset $\{1, 3, 4, 6, 8\}$ and

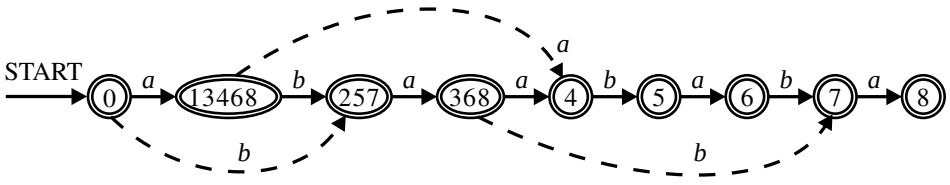


Figure 4.7: Transition diagram of deterministic factor automaton M_2 for string $x = abaababa$ from Example 4.20

continuing to the right. The result of this analysis is shown in the next table:

Analyzed state	Values of border array elements
13468	$\beta[3] = 1, \beta[4] = 1, \beta[6] = 1, \beta[8] = 1$
257	$\beta[5] = 2, \beta[7] = 2$
368	$\beta[6] = 3, \beta[8] = 3$

Resulting border array $\beta(abaababa)$ is summed up in the next table:

i	1	2	3	4	5	6	7	8
symbol	a	b	a	a	b	a	b	a
$\beta[i]$	0	0	1	1	2	3	2	3

□

Definition 4.21
The *backbone* of factor automaton M of a set of strings $S = \{x_1, x_2, \ldots, x_{|S|}\}$ is a part of factor automaton M enabling sequences of transitions for all strings from the set S starting in the initial state and nothing else. □

Definition 4.22
The *depth* of state q of the factor automaton on its backbone is the number of the backbone transitions which are necessary to reach state q from the initial state. □

Algorithm 4.23
Input: Set of strings $S = \{x_1, x_2, \ldots, x_{|S|}\}, x_i \in A^+, i \in <1, |S|>$.
Output: *mborder* array $m\beta[1..n]$,

$$n \leq \sum_{i=1}^{|S|} |x_i|.$$

Method:

1. Construct nondeterministic factor automaton M_1 for S.
2. Construct equivalent deterministic factor automaton M_2 and preserve d–subsets.
3. Extract the backbone of M_2 creating automaton $M_3 = (Q, A, \delta, q_0, F)$.
4. Set $n := |Q| - 1$.
5. Initialize all elements of *mborder* array $m\beta[1..n]$ by the value zero.
6. Do analysis of multiple d–subsets of automaton M_3 from left to right (starting with states having minimal depth):
   If the d–subset has the form $i_1, i_2, \ldots, i_h$ (this sequence is ordered according to the depth of each state), then set $m\beta[j] := i_1$ for $j = i_2, i_3, \ldots, i_h$. □

	a	b
0	$13^1 4^2$	$23^2 4^1$
$13^1 4^2$		24^1
$23^2 4^1$	$3^1 4^2$	3^2
24^1	3^1	3^2
$3^1 4^2$		4^1
3^1		4^1
3^2	4^2	
4^1		
4^2		

Table 4.1: Transition table of deterministic factor automaton M_2 from Example 4.24

Note: We will use the labelling of states reflecting the depth of them instead of the running numbering as in Definition 4.8.

Example 4.24
Let us construct the *mborder* array for set of strings $S = \{abab, abba\}$. In the first step, we construct nondeterministic factor automaton M_1 for set S. Its transition diagram is depicted in Fig. 4.8. Table 4.1 is the transition table

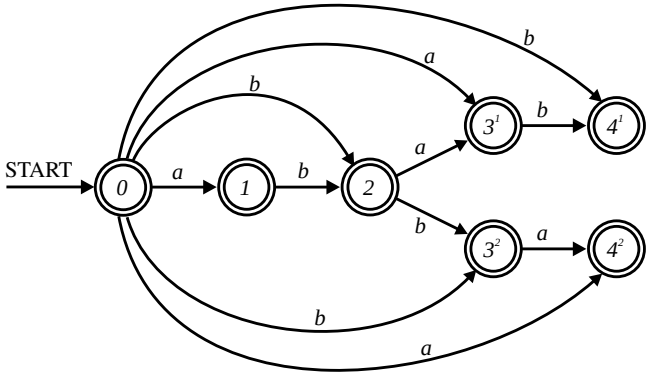


Figure 4.8: Transition diagram of nondeterministic factor automaton M_1 for set $x = \{abab, abba\}$ from Example 4.24

of deterministic factor automaton M_2. Its transition diagram is depicted in Fig. 4.9. The dashed lines and circles show the part of the automaton out of the backbone. Therefore the backbone of M_3 is drawn by solid lines. Now we do analysis of d–subsets. Result is in the next table:

Analyzed state	Values of mborder array elements
$13^1 4^2$	$m\beta(4^2) = m\beta(3^1) = 1$
24^1	$m\beta(4^1) = 2$

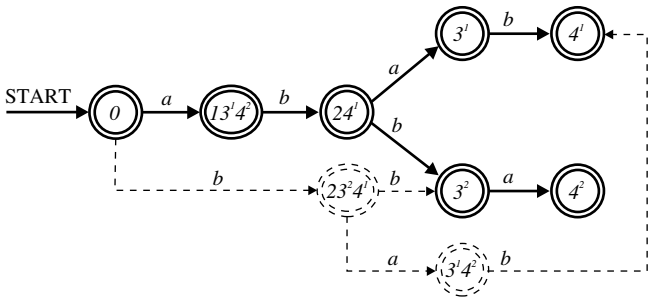


Figure 4.9: Transition diagram of deterministic factor automaton M_2 for the $x = \{abab, abba\}$ from Example 4.24

Resulting mborder array $m\beta(S)$ is shown in the next table:

state	1	2	3^1	3^2	4^1	4^2
symbol	a	b	a	b	b	a
$m\beta$[state]	0	0	1	0	2	1

□

4.4 Repetitions

4.4.1 Classification of repetitions

Problems of repetitions of factors in a string over a finite size alphabet can be classified according to various criteria. We will use five criteria for classification of repetition problems leading to five-dimensional space in which each point corresponds to the particular problem of repetition of a factor in a string. Let us make a list of all dimensions including possible "values" in each dimension:

1. Number of strings:
   - one,
   - finite number greater than one,
   - infinite number.
2. Repetition of factors (see Definition 4.12):
   - with overlapping,
   - square,
   - with gap.
3. Specification of the factor:
   - repeated factor is given,
   - repeated factor is not given,
     - length l of the repeated factor is given exactly,
     - length of the repeated factor is less than given l,
     - length of the repeated factor is greater than given l,
     - finding the longest repeated factor.
4. The way of finding repetitions:

- exact repetition,
- approximate repetition with Hamming distance (R-repetition),
- approximate repetition with Levenshtein distance (DIR-repetition),
- approximate repetition with generalized Levenshtein distance ($DIRT$-repetition),
- Δ-approximate repetition,
- Γ-approximate repetition,
- (Δ, Γ)-approximate repetition.

5. Importance of symbols in factor:
   - take care of all symbols,
   - don't care of some symbols.

The above classification is visualised in Figure 4.10. If we count the number of possible problems of finding repetitions in a string, we obtain $N = 3 * 3 * 2 * 7 * 2 = 272$.

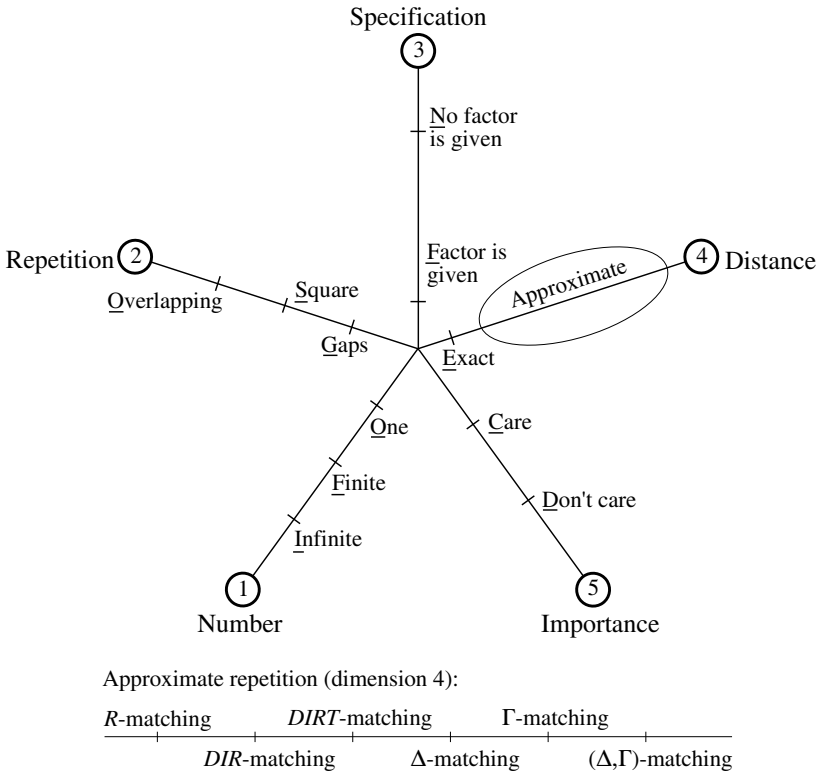


Figure 4.10: Classification of repetition problems

In order to facilitate references to a particular problem of repetition in a string, we will use abbreviations for all problems. These abbreviations are summarized in Table 4.2.

Dimension	1	2	3	4	5
	O	O	F	E	C
	F	S	N	R	D
	I	G		D	
				T	
				Δ	
				Γ	
				(Δ, Γ)	

Table 4.2: Abbreviations of repetition problems

Using this method, we can, for example, refer to the overlapping exact repetition in one string of a given factor where all symbols are considered as the $OOFEC$ problem.

Instead of the single repetition problem we will use the notion of a family of repetitions in string problems. In this case we will use symbol ? instead of a particular symbol. For example $?S???$ is the family of all problems concerning square repetitions.

Each repetition problem can have several instances:

1. verify whether some factor is repeated in the text or not,
2. find the first repetition of some factor,
3. find the number of all repetitions of some factor,
4. find all repetitions of some factor and where they are.

If we take into account all possible instances, the number of repetitions in string problems grows further.

4.4.2 Exact repetitions in one string

In this section we will introduce how to use a factor automaton for finding exact repetitions in one string ($O?NEC$ problem). The main idea is based on the construction of the deterministic factor automaton. First, we construct a nondeterministic factor automaton for a given string. The next step is to construct the equivalent deterministic factor automaton. During this construction, we memorize d-subsets. The repetitions that we are looking for are obtained by analyzing these d-subsets. The next algorithm describes the computation of d-subsets of a deterministic factor automaton.

Algorithm 4.25
Computation of repetitions in one string.
Input: String $T = a_1 a_2 \dots a_n$.
Output: Deterministic factor automaton M_D accepting $Fact(T)$ and

d–subsets for all states of M_D.
Method:

1. Construct nondeterministic factor automaton M_N accepting $Fact(T)$:
   (a) Construct finite automaton M accepting string $T = a_1 a_2 \dots a_n$ and all its prefixes.
   $M = (\{q_0, q_1, q_2, \dots, q_n\}, A, \delta, q_0, \{q_0, q_1, \dots, q_n\})$,
   where $\delta(q_i, a_{i+1}) = q_{i+1}$ for all $i \in \langle 0, n-1 \rangle$.
   (b) Construct finite automaton M_ε from the automaton M by inserting ε–transitions:
   $\delta(q_0, \varepsilon) = \{q_1, q_2, \dots, q_{n-1}, q_n\}$.
   (c) Replace all ε–transitions by non–ε–transitions. The resulting automaton is M_N.
2. Construct deterministic factor automaton M_D equivalent to automaton M_N and memorize the d–subsets during this construction.
3. Analyze d–subsets to compute repetitions. □

Factor automaton M_ε constructed by Algorithm 4.25 has, after step 1.b, the transition diagram depicted in Fig. 4.11. Factor automaton M_N has, after

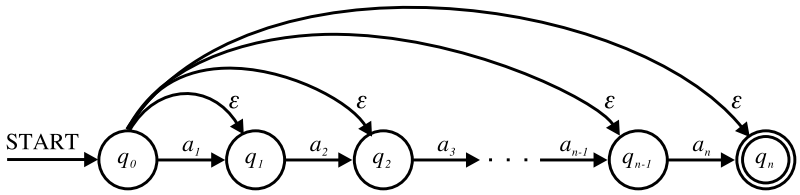


Figure 4.11: Transition diagram of factor automaton M_ε with ε–transitions constructed in step 1.b of Algorithm 4.25

step 1.c of Algorithm 4.25, the transition diagram depicted in Fig. 4.12.

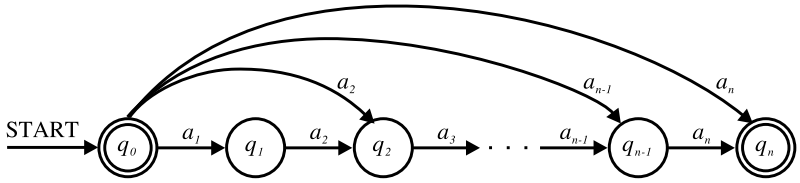


Figure 4.12: Transition diagram of factor automaton M_N after the removal of ε–transitions in step 1.c of Algorithm 4.25

The next example shows the construction of the deterministic factor automaton and the analysis of the d–subsets.

Let us make a note concerning labelling: Labels used as the names of states are selected in order to indicate positions in the string. This labelling will be useful later.

Example 4.26
Let us use text $T = ababa$. At first, we construct nondeterministic factor automaton $M_\varepsilon(ababa) = (Q_\varepsilon, A, \delta_\varepsilon, 0, Q_\varepsilon)$ with ε-transitions. Its transition diagram is depicted in Figure 4.13.

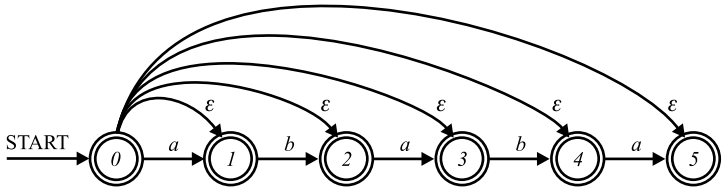


Figure 4.13: Transition diagram of factor automaton $M_\varepsilon(ababa)$ from Example 4.26

Then we remove ε-transitions and resulting nondeterministic factor automaton $M_N(ababa) = (Q_N, A, \delta_N, 0, Q_N)$ is depicted in Figure 4.14 and its transition table is Table 4.3. □

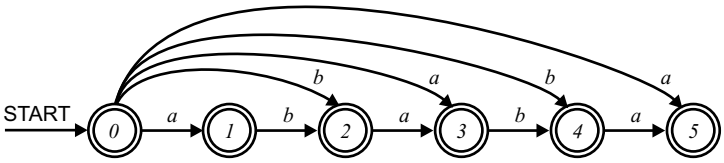


Figure 4.14: Transition diagram of nondeterministic factor automaton $M_N(ababa)$ from Example 4.26

State	a	b
0	$1, 3, 5$	$2, 4$
1		2
2	3	
3		4
4	5	
5		

Table 4.3: Transition table of nondeterministic factor automaton $M_N(ababa)$ from Example 4.26

As next step, we construct equivalent deterministic factor automaton $M_D(ababa) = (Q_D, A, \delta_D, 0, Q_D)$. During this operation we memorize the

created d-subsets. We suppose, taking into account the labelling of the states of the nondeterministic factor automaton, that d-subsets are ordered in the natural way. The extended transition table (with ordered d-subsets) of deterministic factor automaton $M_D(ababa)$ is shown in Table 4.4. Transition diagram of M_D is depicted in Figure 4.15.

State	d-subset	a	b
D_0	0	$1,3,5$	$2,4$
D_1	$1,3,5$		$2,4$
D_2	$2,4$	$3,5$	
D_3	$3,5$		4
D_4	4	5	
D_5	5		

Table 4.4: Transition table of automaton $M_D(ababa)$ from Example 4.26

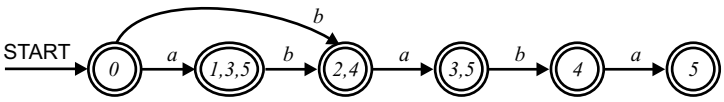


Figure 4.15: Transition diagram of deterministic factor automaton $M_D(ababa)$ from Example 4.26

Now we start the analysis of the resulting d-subsets:
d-subset $d(D_1) = \{1, 3, 5\}$ shows that factor a repeats at positions $1, 3$ and 5 of the given string, and its length is one. d-subset $d(D_2) = \{2, 4\}$ shows that factor ab repeats, and its occurrence in the string ends at positions 2 and 4 and its length is two. Moreover, suffix b of this factor also repeats at the same positions as factor ab. d-subset $d(D_3) = \{3, 5\}$ shows that factor aba repeats, and its occurrence in the string ends at positions 3 and 5 and its length is three. Moreover, its suffix ba also repeats at the same positions. Suffix a of factor aba also repeats at positions 3 and 5, but we have already obtained this information during analysis of the d-subset $d(D_1) = \{1, 3, 5\}$. Analysis of the d-subsets having only single states brings no further information on repeating factors.

A summary of these observations we collect in a *repetition table*. The repetition table contains one row for each d-subset. It contains d-subset, repeating factor, and a list of repetitions. The list of repetitions indicates position of the repeating factor and the type of repetition. □

Definition 4.27
Let T be a string. The repetition table for T contains the following items:

1. d-subset,

2. corresponding factor,
3. list of repetitions of the factor containing elements of the form (i, x_i), where i is the position of the factor in string T,
   X_i is the type of repetition:
   F - the first occurrence of the factor,
   O - repetition with overlapping,
   S - repetition as a square,
   G - repetition with a gap. □

Repetition table for string $T = ababa$ from Example 4.26 is shown in Table 4.5.

d-subset	Factor	List of repetitions
$1, 3, 5$	a	$(1, F)$ $(3, G), (5, G)$
$2, 4$	ab	$(2, F)$ $(4, S)$
$2, 4$	b	$(2, F)$ $(4, G)$
$3, 5$	aba	$(3, F)$ $(5, O)$
$3, 5$	ba	$(3, F)$ $(5, S)$

Table 4.5: Repetition table of *ababa*

Construction of the repetition table is based on the following observations illustrated in Figure 4.16 and Lemmata 4.28 and 4.29 show its correctness.

Lemma 4.28
Let T be a string and $M_D(T)$ be the deterministic factor automaton for T with states labelled by corresponding d-subsets. If factor $u = a_1a_2 \dots a_m$, $m \geq 1$, repeats in string T and its occurrences start at positions $x + 1$ and $y + 1$, $x \neq y$ then there exists a d-subset in $M_D(T)$ containing the pair $\{x + m, y + m\}$.

Proof
Let $M_N(T) = (Q_N, A, \delta_N, q_0, Q_N)$ be the nondeterministic factor automaton for T and let $u = a_1a_2 \dots a_m$ be the factor starting at positions $x + 1$ and $y + 1$ in T, $x \neq y$. Then there are transitions in $M_N(T)$ from state 0 to states $x + 1$ and $y + 1$ for symbol a_1, ($\delta_N(0, a_1)$ contains $x + 1$ and $y + 1$). It follows from the construction of $M_N(T)$ that:

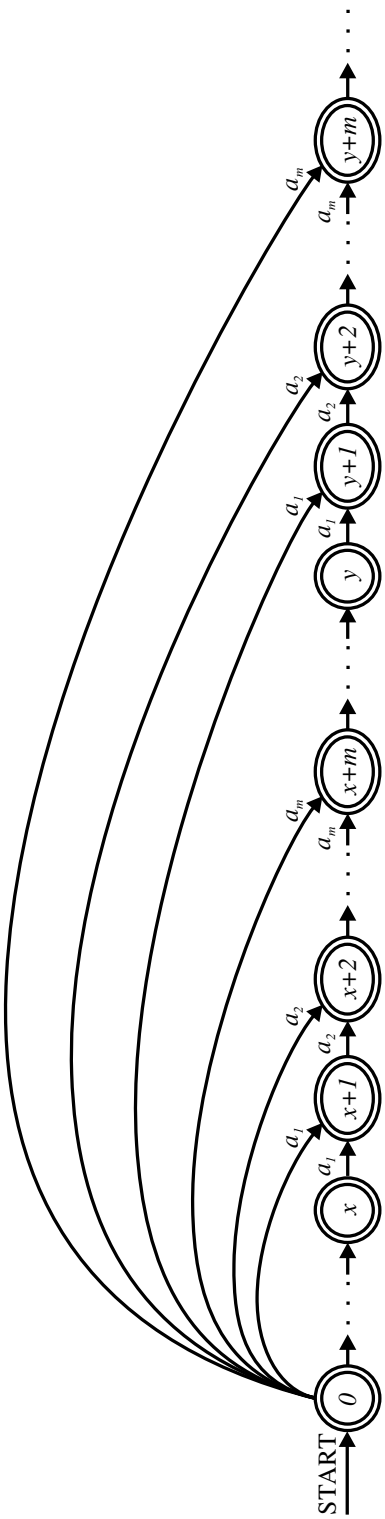


Figure 4.16: Repeated factor $u = a_1 a_2 \dots a_m$ in $M_N(T)$

$$\begin{array}{ll}
\delta_N(x+1,a_2)=\{x+2\}, & \delta_N(y+1,a_2)=\{y+2\},\\
\delta_N(x+2,a_3)=\{x+3\}, & \delta_N(y+2,a_3)=\{y+3\},\\
\vdots & \vdots\\
\delta_N(x+m-1,a_m)=\{x+m\}, & \delta_N(y+m-1,a_m)=\{y+m\}.
\end{array}$$

Deterministic factor automaton $M_D(T) = (Q_D, A, \delta_D, D_0, Q_D)$ then contains states $D_0, D_1, D_2, \ldots D_m$ having this property:

$$\begin{array}{ll}
\delta_D(D_0,a_1)=D_1, & \{x+1,y+1\}\subset D_1,\\
\delta_D(D_1,a_2)=D_2, & \{x+2,y+2\}\subset D_2,\\
\vdots & \vdots\\
\delta_D(D_{m-1},a_m)=D_m, & \{x+m,y+m\}\subset D_m.
\end{array}$$

We can conclude that the d-subset D_m contains the pair $\{x+m, y+m\}$. □

Lemma 4.29
Let T be a string and let $M_D(T)$ be the deterministic factor automaton for T with states labelled by corresponding d-subsets. If a d-subset D_m contains two elements $x+m$ and $y+m$ then there exists factor $u = a_1a_2\ldots a_m$, $m \geq 1$, starting at both positions x and y in string T.

Proof
Let $M_N(T)$ be the nondeterministic factor automaton for T. If a d-subset D_m contains elements from $\{x+m, y+m\}$ then it holds for δ_N of $M_N(T)$:
$\{x+m, y+m\} \subset \delta_N(0, a_m)$, and
$\delta_N(x+m-1, a_m) = \{x+m\}$,
$\delta_N(y+m-1, a_m) = \{y+m\}$ for some $a_m \in A$.
Then d-subset D_{m-1} such that $\delta_D(D_{m-1}, a_m) = D_m$ must contain
$x+m-1, y+m-1$ such that $\{x+m-1, y+m-1\} \subset \delta_N(0, a_{m-1})$,
$\delta_N(x+m-2, a_{m-1}) = \{x+m-1\}$,
$\delta_N(y+m-2, a_{m-1}) = \{y+m-1\}$
and for the same reason D-subset D_1 must contain $x+1, y+1$ such that $\{x+1, y+1\} \subset \delta_N(0, a_1)$ and $\delta_N(x, a_1) = \{x+1\}$, $\delta_N(y, a_1) = \{y+1\}$.
Then there exists the sequence of transitions in $M_D(T)$:

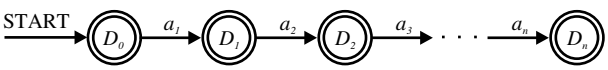


Figure 4.17: Repeated factor $u = a_1a_2\ldots a_m$ in $M_D(T)$

$$\begin{array}{lll}
(D_0, a_1a_2\ldots a_m) & \vdash & (D_1, a_2\ldots a_m)\\
 & \vdash & (D_2, a_3\ldots a_m)
\end{array}$$

$$\begin{aligned}
&\vdots\\
&\vdash\ (D_{m-1}, a_m)\\
&\vdash\ (D_m, \varepsilon),
\end{aligned}$$

where

$$\begin{aligned}
&\{x+1, y+1\} \subset D_1,\\
&\vdots\\
&\{x+m, y+m\} \subset D_m.
\end{aligned}$$

This sequence of transitions corresponds to two different sequences of transitions in $M_N(T)$ going through state $x+1$:

$$\begin{aligned}
(0, a_1a_2\ldots a_m) &\vdash\ (x+1, a_2\ldots a_m)\\
&\vdash\ (x+2, a_3\ldots a_m)\\
&\vdots\\
&\vdash\ (x+m-1, a_m)\\
&\vdash\ (x+m, \varepsilon),
\end{aligned}$$

$$\begin{aligned}
(x, a_1a_2\ldots a_m) &\vdash\ (x+1, a_2\ldots a_m)\\
&\vdash\ (x+2, a_3\ldots a_m)\\
&\vdots\\
&\vdash\ (x+m-1, a_m)\\
&\vdash\ (x+m, \varepsilon).
\end{aligned}$$

Similarly two sequences of transitions go through state $y+1$:

$$\begin{aligned}
(0, a_1a_2\ldots a_m) &\vdash\ (y+1, a_2\ldots a_m)\\
&\vdash\ (y+2, a_3\ldots a_m)\\
&\vdots\\
&\vdash\ (y+m-1, a_m)\\
&\vdash\ (y+m, \varepsilon),
\end{aligned}$$

$$\begin{aligned}
(y, a_1a_2\ldots a_m) &\vdash\ (y+1, a_2\ldots a_m)\\
&\vdash\ (y+2, a_3\ldots a_m)\\
&\vdots\\
&\vdash\ (y+m-1, a_m)\\
&\vdash\ (y+m, \varepsilon).
\end{aligned}$$

It follows from this that the factor $u = a_1a_2\ldots a_m$ is present twice in string T in different positions $x+1, y+1$. □

The following Lemma is a simple consequence of Lemma 4.29.

Lemma 4.30
Let u be a repeating factor in string T. Then all factors of u are also repeating factors in T. □

Definition 4.31
If u is a repeating factor in text T and there is no longer factor of the form vuw, v or $w \neq \varepsilon$ but not both, which is also a repeating factor, then we will call u the *maximal repeating factor*. □

Definition 4.32
Let $M_D(T)$ be a deterministic factor automaton. The *depth* of each state D of M_D is the length of the longest sequence of transitions leading from the initial state to state D. □

If there exists a sequence of transitions from the initial state to state D which is shorter than the depth of D, it corresponds to the suffix of the maximal repeating factor.

Lemma 4.33
Let u be a maximal repeating factor in string T. The length of this factor is equal to the depth of the state in $M_D(T)$ indicating the repetition of u.
□

Proof
The path for maximal repeating factor $u = a_1a_2 \dots a_m$ starts in the initial state, because states $x+1$ and $y+1$ of nondeterministic factor automaton $M_N(T)$ are direct successors of its initial state and therefore $\delta_D(D_0, a_1) = D_1$ and $\{x+1, y+1\} \subset D_1$. Therefore there exists a sequence of transitions in deterministic factor automaton $M_D(T)$:

$$\begin{aligned}(D_0, a_1a_2 \dots a_m) &\vdash (D_1, a_2 \dots a_m)\\ &\vdash (D_2, a_3 \dots a_m)\\ &\vdots\\ &\vdash (D_{m-1}, a_m)\\ &\vdash (D_m, \varepsilon)\end{aligned}$$

□

There follows one more observation from Example 4.26.

Lemma 4.34
If some state in $M_D(T)$ has a corresponding d-subset containing one element only, then its successor also has a corresponding d-subset containing one element.

Proof
This follows from the construction of the deterministic factor automaton. The transition table of nondeterministic factor automaton $M_N(T)$ has more than one state in the row for the initial state only. All other states have at most one successor for a particular input symbol. Therefore in the equivalent deterministic factor automaton $M_D(T)$ the state corresponding to a d-subset having one element may have only one successor for one symbol, and this state has a corresponding d-subset containing just one element. □

We can use this observation during the construction of deterministic factor automaton $M_D(T)$ in order to find some repetition. It is enough to construct only the part of $M_D(T)$ containing d-subsets with at least two elements. The rest of $M_D(T)$ gives no information on repetitions.

Algorithm 4.35
Constructing a repetition table containing exact repetitions in a given string.
Input: String $T = a_1 a_2 \dots a_n$.
Output: Repetition table R for string T.
Method:

1. Construct deterministic factor automaton
   $M_D(T) = (Q_D, A, \delta_D, 0, Q_D)$ for given string T.
   Memorize for each state $q \in Q_D$:
   (a) d-subset $D(q) = \{r_1, r_2, \dots, r_p\}$,
   (b) $d = depth(q)$,
   (c) maximal repeating factor for state q $maxfactor(q) = x, |x| = d$.

2. Create rows in repetition table R for each state q having $D(q)$ with more than one element:

   (a) the row for maximal repeating factor x of state q has the form:
   $(\{r_1, r_2, \dots, r_p\}, x, \{(r_1, F), (r_2, X_2), (r_3, X_3), \dots, (r_p, X_p)\}$,
   where $X_i,\ 2 \leq i \leq p$, is equal to
      i. O, if $r_i - r_{i-1} < d$,
      ii. S, if $r_i - r_{i-1} = d$,
      iii. G, if $r_i - r_{i-1} > d$,

   (b) for each suffix y of x (such that the row for y was not created before) create the row of the form:
   $(\{r_1, r_2 \dots, r_p\}, y, \{(r_1, F), (r_2, X_2), (r_3, X_3), \dots, (r_p, X_p)\}$,
   where $X_i,\ 2 \leq i \leq p$, is deduced in the same manner. □

An example of the repetition table is shown in Example 4.26 for string $T = ababa$.

4.4.3 Complexity of computation of exact repetitions

The time and space complexity of the computation of exact repetitions in string is treated in this Chapter.
The time complexity is composed of two parts:

1. The complexity of the construction of the deterministic factor automaton. If we take the number of states and transitions of the resulting factor automaton then the complexity is linear. More exactly the number of its states is

   $NS \leq 2n - 2$,
   and the number of transitions is

   $NT \leq 3n - 4$.

2. The second part of the overall complexity is the construction of repetition table. The number of rows of this table is the number of different multiple d-subsets. The highest number of multiple d-subsets has the factor automaton for text $T = a^n$. Repeating factors of this text are $a, a^2, \ldots, a^{n-1}$.

There is necessary, for the computation of repetitions using factor automata approach, to construct the part of deterministic factor automaton containing only all multiple states. It is the matter of fact, that a simple state has at most one next state and it is simple one, too. Therefore, during the construction of deterministic factor automaton, we can stop construction of the next part of this automaton as soon as we reach a simple state.

Example 4.36
Let us have text $T = a^n,\ n > 0$. Let us construct deterministic factor automaton $M_D(a^n)$ for text T. Transition diagram of this automaton is depicted in Fig. 4.18. Automaton $M_D(a^n)$ has $n+1$ states and n transitions.

Figure 4.18: Transition diagram of deterministic factor automaton $M_D(a^n)$ for text $T = a^n$ from Example 4.36

Number of multiple states is $n - 1$.

To construct this automaton in order to find all repetitions, we must construct the whole automaton including the initial state and the state n (terminal state). Repetition table R has the form shown in Table 4.6. □

The opposite case to the previous one is the text composed of symbols which are all different. The length of such text is limited by the size of alphabet.

d-subset	Factor	List of repetitions
$1, 2, \ldots, n$	a	$(1, F), (2, S), (3, S), \ldots, (n, S)$
$2, \ldots, n$	aa	$(2, F), (3, O), (4, O), \ldots, (n, O)$
$\vdots$		
$n-1, n$	a^{n-1}	$(n-1, F), (n, O)$

Table 4.6: Repetition table R for text $T = a^n$ from Example 4.36

Example 4.37
Let the alphabet be $A = \{a, b, c, d\}$ and text $T = abcd$. Deterministic factor automaton $M_D(abcd)$ for text T has transition diagram depicted in Fig. 4.19. Automaton $M_D(abcd)$ has $n+1$ states and $2n-1$ transitions. All respective

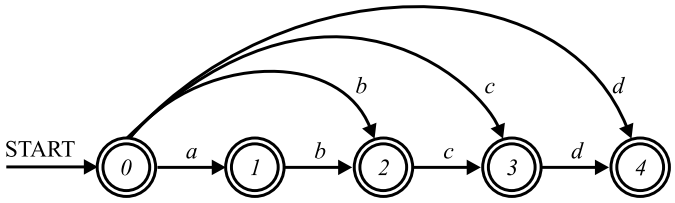


Figure 4.19: Transition diagram of deterministic factor automaton $M_D(abcd)$ for text $T = abcd$ from Example 4.36

d-subsets are simple. To construct this automaton in order to find all repetitions, we must construct all next states of the initial state for all symbols of the text. The number of these states is just n. The repetition table is empty. □

Now, after the presentation both limit cases, we will try to find some case inbetween with the maximal complexity. We guess, that the next example is showing it. The text selected in such way, that all proper suffixes of the prefix of the text appear in it and therefore they are repeating.

Example 4.38
Let the text be $T = abcdbcdcdd$. Deterministic factor automaton $M_D(T)$ has the transition diagram depicted in Fig. 4.20. Automaton M_D has 17 states and 25 transitions while text T has 10 symbols. The number of multiple d-subsets is 6. To construct this automaton in order to find all repetitions, we must construct all multiple states and moreover the states corresponding to single d-subsets: $0, 1, 5, 8, A$.

The results is, that we must construct 11 states from the total number of 17 states. Repetition table R is shown in Table 4.7. □

It is known, that the maximal state and transition complexity of the

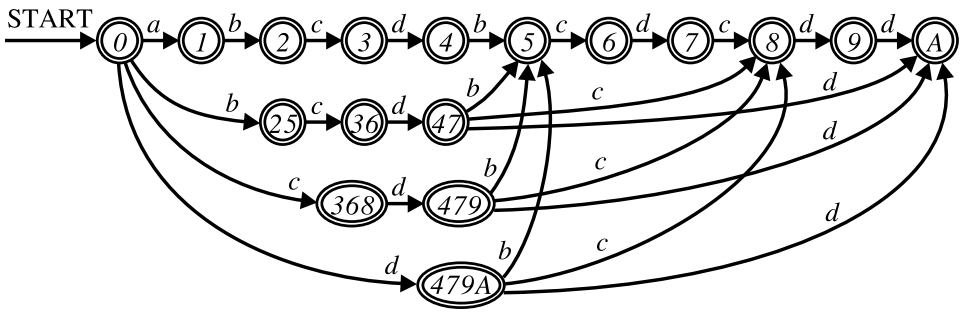


Figure 4.20: Transition diagram of deterministic factor automaton $M_D(T)$ for text $T = abcdbcdcdd$ from Example 4.38

d-subset	Factor	List of repetitions
25	b	$(2, F), (5, G)$
36	bc	$(3, F), (6, G)$
47	bcd	$(4, F), (7, S)$
368	c	$(3, F), (6, G), (8, G)$
479	cd	$(4, F), (7, G), (9, G)$
$479A$	d	$(4, F), (7, G), (9, G), (10, S)$

Table 4.7: Repetition table R for text $T = abcdbcdcdd$ from Example 4.38

factor automaton is reached for text $T = ab^{n-2}c$. Let us show such factor automaton in this context.

Example 4.39
Let the text be $T = ab^4c$. Deterministic factor automaton $M_D(T)$ has transition diagram depicted in Fig. 4.21. Automaton $M_D(T)$ has 10 $(2 * 6 - 2)$

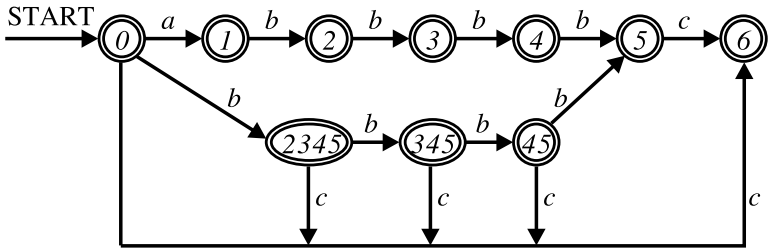


Figure 4.21: Transition diagram of deterministic factor automaton $M_D(T)$ for text $T = ab^4c$ from Example 4.39

states and 14 $(3 * 6 - 4)$ transitions while the text has 6 symbols. The number of multiple states is 3 $(6 - 3)$. To construct this automaton in order to find all repetitions, we must construct the 3 multiple states and moreover 3

simple states. Therefore we must construct 6 states from the total number of 10 states. Repetition table R is shown in Table 4.8. □

d-subset	List of repetitions
2345	$(b, F)(2, F), (3, S), (4, S), (5, S)$
345	$(bb, F)(3, F), (4, O), (5, O)$
45	$(bbb, F)(4, F), (5, O)$

Table 4.8: Repetition table R for text $T = ab^4c$ from Example 4.39

We have used, in the previous examples, three measures of complexity:

1. The number of multiple states of the deterministic factor automaton. This number is equal to the number of rows in the resulting repetition table, because each row of the repetition table corresponds to one multiple d-subset. Moreover, it corresponds to the number of repeating factors.
2. The number of states which are necessary to construct in order to get all information on repetitions. We must reach the simple state on all pathes starting in the initial state. We already know that there is at most one successor of a simple state and it is a simple state, too (Lemma 4.34). The number of such states which is necessary to construct is therefore greater than the number of multiple states.
3. The total number of repetitions (occurrences) of all repeating factors in text. This number corresponds to the number of items in the last column of the repetition table headed by "List of repetitions".

The results concerning the measures of complexity from previous examples are summarized in the Table 4.9.

Text	No. of multiple states	No. of necessary states	No. of repetitions
a^n	$n-1$	$n+1$	$(n^2+n-2)/2$
$a_1a_2\dots a_n$ (all symbols unique)	0	$n+1$	0
$a_1a_2\dots a_ma_2\dots a_m\dots a_{m-1}a_ma_m$	$(m^2-m)/2$	$(m^2+m)/2$	$\sum_{i=1}^{m-1} i(m-i+1)$
$ab^{n-2}c$	$n-3$	n	$(n^2-3n)/2$

Table 4.9: Measures of complexity from Examples 4.36, 4.37, 4.38, 4.39

Let us show how the complexity measures from Table 4.9 have been computed.

Example 4.40
Text $T = a^n$ has been used in Example 4.36. The number of multiple states is $n-1$ which is the number of repeating factors. The number of necessary states is $n+1$ because the initial and the terminal states must be constructed. The number of repetitions is given by the sum:

$$n + (n-1) + (n-2) + \ldots + 2 = \frac{n^2+n-2}{2}.$$ □

Example 4.41
Text $T = abcd$ has been used in Example 4.37. This automaton has no multiple state. The number of necessary states is $n+1$. It means, that in order to recognize that no repetition exists in such text all states of this automaton must be constructed. □

Example 4.42
Text $T = abcdbcdcdd$ used in Example 4.38 has very special form. It consists of prefix $abcd$ followed by all its proper suffixes. It is possible to construct such text only for some n. Length n of the text must satisfy condition:

$$n = \sum_{i=1}^{m} i = \frac{m^2+m}{2},$$

where m is the length of the prefix in question. It follows that

$$m = \frac{-1 \pm \sqrt{1+8n}}{2}$$

and therefore $m = O(\sqrt{n})$.

The number of multiple states is

$$(m-1) + (m-2) + \ldots + 1 = \frac{m^2-m}{2}.$$

The number of necessary states we must increase by m which is the number of simple states being next states of the multiple states and the initial state. Therefore the number of necessary states is $(m^2+m)/2$. The number of repetitions is

$$m + 2(m+1) + 3(m-2) + \ldots + (m-1)2 = \sum_{i=1}^{m-1} i(m-i+1).$$

Therefore this number is $O(m^2) = O(n)$. □

Example 4.43
Text $T = ab^{n-2}c$ used in Example 4.39 leads to the factor automaton having maximal number of states and transitions. The number of multiple states is

equal to $n-3$ and the number of necessary states is equal to n. The number of repetitions is

$$(n-2)+(n-3)+\ldots+2=\frac{n^2-3n}{2}.$$

□

It follows from the described experiments that the complexity of deterministic factor automata and therefore the complexity of computation of repetitions for a text of length n has these results:

1. The number of multiple states is linear. It means that the repetition table has $O(n)$ rows. It is the space complexity of the computation all repeated factors.
2. The number of necessary states is again linear. It means that time complexity of the computation all repeated factors is $O(n)$.
3. The number of repetitions is $O(n^2)$ which is the time and space complexity of the computation of all occurrences of repeated factors.

4.4.4 Exact repetitions in a finite set of strings

The idea of the use of a factor automaton for finding exact repetitions in one string can also be used for finding exact repetitions in a finite set of strings (*F?NEC* problem). Next algorithm is an extension of Algorithm 4.25 for a finite set of strings.

Algorithm 4.44
Computation of repetitions in a finite set of strings.
Input: Set of strings $S=\{x_1,x_2,\ldots,x_{|S|}\},\ x_i\in A^+,\ i=1,2,\ldots,|S|$.
Output: Multiple front end $MFE(M_D)$ of deterministic factor automaton M_D accepting $Fact(S)$ and d-subsets for all states of $MFE(M_D)$.
Method:

1. Construct nondeterministic factor automata $M_{i\varepsilon}$ for all strings $x_i,\ i=1,2,\ldots,|S|$:
   (a) Construct finite automaton M_i accepting string x_i and all its prefixes for all $i=1,2,\ldots,|S|$.
   (b) Construct finite automaton $M_{i\varepsilon}$ from automaton M_i by inserting ε-transitions from the initial state to all other states for all $i=1,2,\ldots,|S|$.
2. Construct automaton M_ε by merging of initial states of automata $M_{i\varepsilon}$ for $i-1,2,\ldots,|S|$.
3. Replace all ε-transitions by non-ε-transitions. The resulting automaton is M_N.

4. Construct the multiple front end of deterministic $MFE(M_D)$ factor automaton M_D equivalent to automaton M_N using Algorithm 3.21 and save the d-subset during this construction. □

We show in the next example the construction of a factor automaton and the analysis of d–subsets created during the construction of factor automaton for a finite set of strings.

Example 4.45
Let us construct the factor automaton for the set of strings $S = \{abab, abba\}$. First, we construct factor automata $M_{1\varepsilon}$ and $M_{2\varepsilon}$ for both strings in S. Their transition diagrams are depicted in Figs 4.22 and 4.23, respectively.

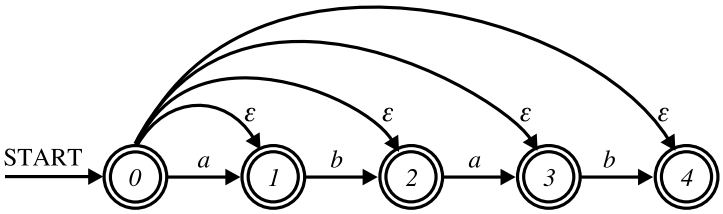


Figure 4.22: Transition diagram of factor automaton $M_{1\varepsilon}$ accepting $Fact(abab)$ from Example 4.45

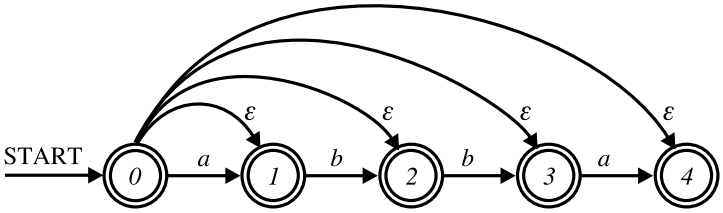


Figure 4.23: Transition diagram of factor automaton $M_{2\varepsilon}$ accepting $Fact(abba)$ from Example 4.45

In the second step we construct automaton M_ε accepting language $L(M_\varepsilon) = Fact(abab) \cup Fact(abba)$. Its transition diagram is depicted in Fig. 4.24.

In the third step we construct automaton M_N by removing ε–transitions from automaton M_ε. Its transition diagram is depicted in Fig. 4.25.

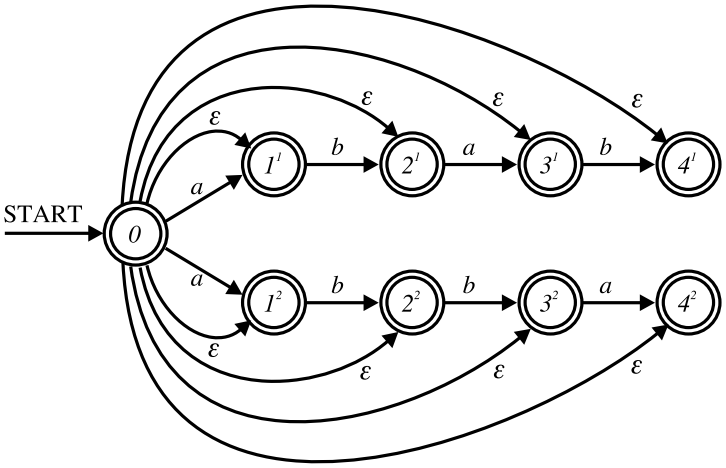


Figure 4.24: Transition diagram of factor automaton M_ε accepting set $Fact(abab) \cup Fact(abba)$ from Example 4.45

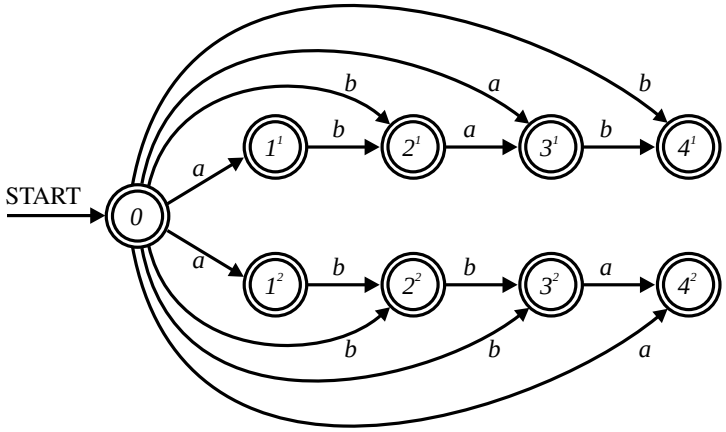


Figure 4.25: Transition diagram of nondeterministic factor automaton M_N accepting set $Fact(abab) \cup Fact(abba)$ from Example 4.45

	a	b
0	$1^1 1^2 3^1 4^2$	$2^1 2^2 3^2 4^1$
$1^1 1^2 3^1 4^2$		$2^1 2^2 4^1$
$2^1 2^2 3^2 4^1$	$3^1 4^2$	3^2
$2^1 2^2 4^1$	3^1	3^2
$3^1 4^2$		4^1
3^1		4^1
3^2	4^2	
4^1		
4^2		

Table 4.10: Transition table of deterministic factor automaton M_D from Example 4.45

The last step is to construct deterministic factor automaton M_D. Its transition table is shown in Table 4.10. The transition diagram of the resulting deterministic factor automaton M_D is depicted in Fig. 4.26. Now we

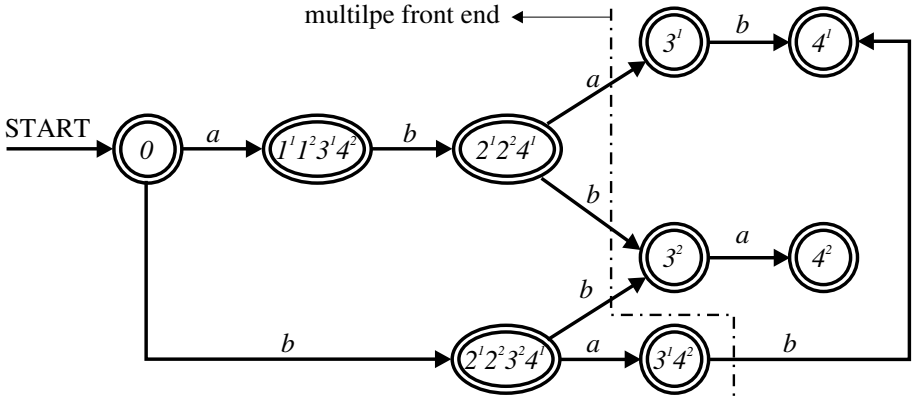


Figure 4.26: Transition diagram of deterministic factor automaton M_D accepting set $Fact(abab) \cup Fact(abba)$ from Example 4.45

do the analysis of d–subsets of resulting automaton M_D. The result of this analysis is the repetition table shown in Table 4.11 for set $S = \{abab, abba\}$. □

Definition 4.46
Let S be a set of strings $S = \{x_1, x_2, \ldots, x_{|S|}\}$. The repetition table for S contains the following items:

1. d–subset,
2. corresponding factor,

d–subset	Factor	List of repetitions
$1^1 1^2 3^1 4^2$	a	$(1,1,F),(2,1,F),(1,3,G),(2,4,G)$
$2^1 2^2 4^1$	ab	$(1,2,F),(2,2,F),(1,4,S)$
$2^1 2^2 3^2 4^1$	b	$(1,2,F),(2,2,F),(2,3,S),(1,4,G)$
$3^1 4^2$	ba	$(1,3,F),(2,4,F)$

Table 4.11: Repetition table for set $S = \{abab, abba\}$ from Example 4.45

3. list of repetitions of the factor containing elements of the form (i, j, X_{ij}), where i is the index of the string in S,
   j is the position in string x_i,
   X_{ij} is the type of repetition:
   F - the first occurrence of the factor in string x_i,
   O - repetition of the factor in x_i with overlapping,
   S - repetition as a square in x_i,
   G - repetition with a gap in x_i.

□

Let us suppose that each element of d-subset constructed by Algorithm 4.44 keeps two kind of information:

- index of the string in S to which it belongs,
- depth (position) in this string.

Moreover we suppose, that it is possible to identify the longest factor (maximal repeating factor) to which the d-subset belongs.

Algorithm 4.47
Constructing a repetition table containing exact repetitions in a finite set of strings.
Input: Multiple front end of factor automaton for set of strings $S = \{x_1, x_2, \ldots, x_{|S|}\},\ x_i \in A^+,\ i = 1, 2, \ldots, |S|$.
Output: Repetition table R for set S.
Method: Let us suppose, that d-subset of multiple state q has form $\{r_1, r_2, \ldots, r_p\}$. Create rows in repetition table R for each multiple state q: the row for maximal repeating factor x of state q has the form:
$(\{r_1, r_2, \ldots, r_p\}, x, \{(i_1, j_1, F), (i_2, j_2, X_{i_2,j_2}), \ldots, (i_p, j_p, X_{i_p,j_p})$,
where i_l is the index of the string in $S,\ l = 1, 2, \ldots, |S|$,
j_l is the position in string $x_i,\ l = 1, 2, \ldots, |S|$,
X_{ij} is the type of repetition:
i: O, if $j_l - j_{l-1} < |x|$,
ii: S, if $j_l - j_{l-1} = |x|$,
iii: G, if $j_l - j_{l-1} > |x|$. □

4.4.5 Computation of approximate repetitions

We have used the factor automata for finding exact repetitions in either one string or in a finite set of strings. A similar approach can be used for finding approximate repetitions as well. We will use approximate factor automata for this purpose. As before, a part of deterministic approximate factor automaton will be useful for the repetition finding. This part we will call mixed multiple front end.
Let us suppose that finite automaton M is accepting set
$Approx(x) = \{y : D(x, y) \leq k\}$,
where D is a metrics and k is the maximum distance. We can divide automaton M into two parts:

- "*exact part*" which is used for accepting string x,
- "*approximate part*" which is used when other strings in $Approx(x)$ are accepting.

For the next algorithm we need to distinquish states either in exact part or in approximate part. Let us do this distinction by labelling of states in question.

Definition 4.48
A *mixed multiple front end* of deterministic approximate factor automaton is a part of this automaton containing only multiple states with:

a) d-subsets containing only states from *exact part* of nondeterministic automaton,

b) d-subset containing mix of states from *exact part* and *approximate part* of nondeterministic automaton with at least one state from exact part.

Let us call such states *mixed multiple states*. □

We can construct the mixed multiple front end by a little modified of Algorithm 3.21. The modification consists in modification of point 3(b) in this way:
(b) if $\delta'(q, a)$ is a mixed multiple state then $Q' = Q' \cup \delta'(q, a)$.

4.4.6 Approximate repetitions – Hamming distance

In this Section, we show how to find approximate repetitions using the Hamming distance ($O?NRC$ problem).

Example 4.49
Let string $x = abba$. We construct an approximate factor automaton using Hamming distance $k = 1$.

1. We construct a finite automaton accepting string x and all strings with Hamming distance equal to 1. The set of these strings is denoted $H_1(abba)$. The resulting finite automaton has the transition diagram depicted in Fig. 4.27.

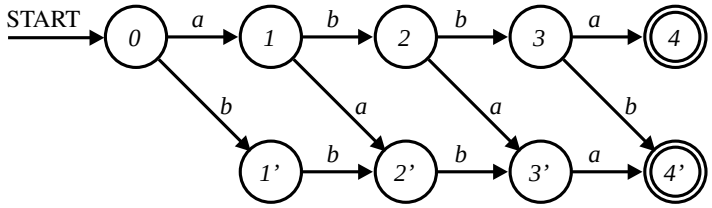


Figure 4.27: Transition diagram of the "Hamming" automaton accepting $H_1(abba)$ with Hamming distance $k = 1$ from Example 4.49

2. We use the principle of inserting the ε–transitions from state 0 to states 1, 2, 3 and 4 and we fix all states as final states. The transition diagram with inserted ε–transition and fixed final states is depicted in Fig. 4.28.

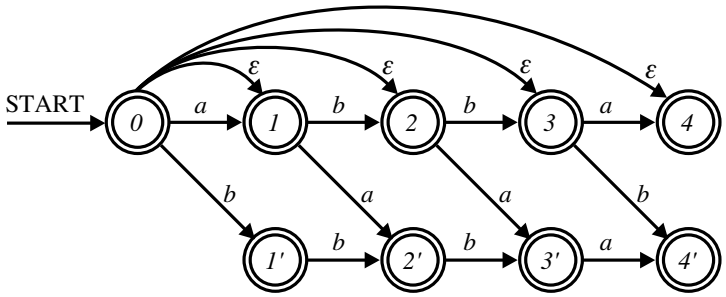


Figure 4.28: Transition diagram of the "Hamming" factor automaton with fixed final states and inserted ε–transitions from Example 4.49

3. We replace ε–transitions by non–ε–transitions. The resulting automaton has the transition diagram depicted in Fig. 4.29.
4. The final operation is the construction of the equivalent deterministic finite automaton. Its transition table is Table 4.12

The transition diagram of the resulting deterministic approximate factor automaton is depicted in Fig. 4.30.

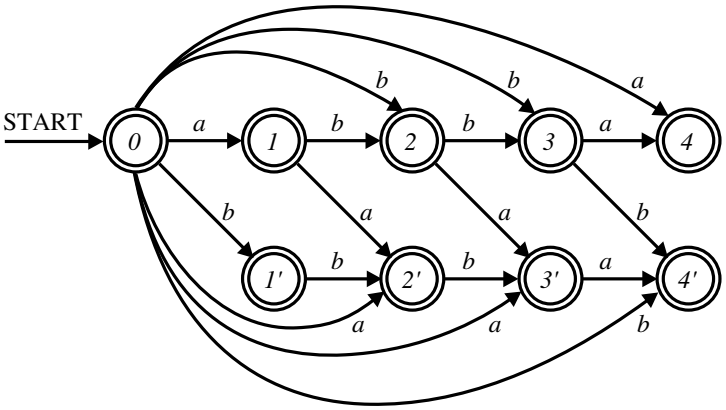


Figure 4.29: Transition diagram of the nondeterministic "Hamming" factor automaton after removal of ε–transitions from Example 4.49

	a	b
0	$142'3'$	$231'4'$
$142'3'$	$2'4'$	$23'$
$231'4'$	$3'4$	$32'4'$
$2'4'$		$3'$
$23'$	$3'4'$	3
$3'4$	$4'$	
$32'4'$	4	$3'4'$
$3'$	$4'$	
$3'4'$	$4'$	
3	4	$4'$
$4'$		
4		

Table 4.12: Transition table of the deterministic "Hamming" factor automaton from Example 4.49

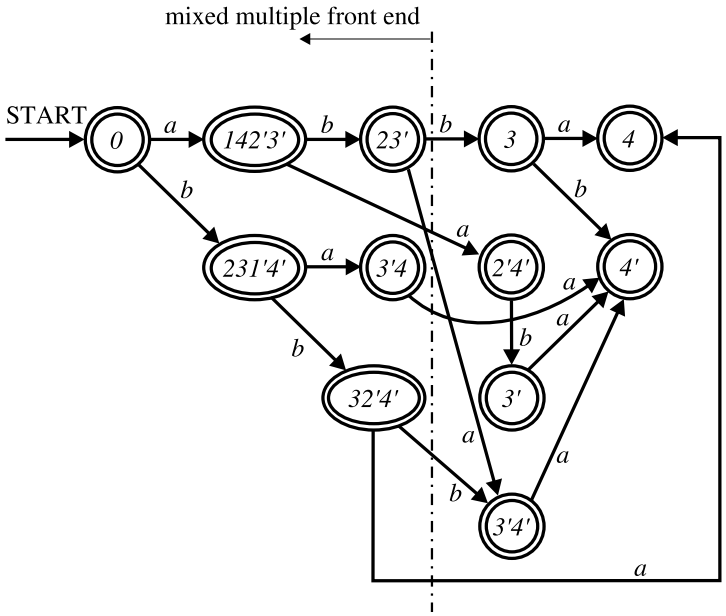


Figure 4.30: Transition diagram of the deterministic "Hamming" factor automaton for $x = abba$, Hamming distance $k = 1$ from Example 4.49

Construction of repetition table is based on the following observation concerning mixed multiple states:

1. If only exact states are in the respective d-subset then exact repetitions take place.
2. Let us suppose that a mixed multiple state corresponding to factor x has d-subset containing pair (r_1, r_2), where r_1 is an exact state and r_2 is an approximate state. It means that state r_1 corresponds to exact factor x. There is a sequence of transitions for factor x also to the state r_2. Therefore in the exact part must be a factor y such that the distance of x and y less than k. It means that factor y is an approximate repetition of x.

Now we construct the approximate repetition table. We take into account the repetition of factors which are longer than k. In this case $k = 1$ and therefore we select repetitions of factors having length greater or equal to two. The next table contains information on the approximate repetition of factors of the string $x = abba$.

d–subset	Factor	Approximate repetitions
$23'$	ab	$(2, ab, F), (3, bb, O)$
$32'4'$	bb	$(3, bb, F), (2, ab, O), (4, ba, O)$
$3'4$	ba	$(4, ba, F), (3, bb, O)$

□

As we can see, the approximate repetition table is similar to the repetition table expressing the exact repetitions (see Definition 4.27). But there are two important differences:

1. There are no all multiple d-subsets in the first column. The only d-subsets used are such having at least one state corresponding to the "exact" part of the nondeterministic approximate factor automaton.
2. There are, in the last column, triples containing repeating factors. This is motivated by the fact, that approximate factor can be different from original factor. Triple (i, x, F) always corresponds to the first "exact" factor.

Let us go back to the approximate repetition table in Example 4.49. The first row contains d-subset 23'. It means, that factor ab at position 2 has approximate repetition bb at position 3.

4.4.7 Approximate repetitions – Levenshtein distance

Let us note that Levenshtein distance between strings x and y is defined as the minimum number of editing operations delete, insert and replace which are necessary to convert string x into string y. In this section we show solution of $O?NDC$ problem.

Example 4.50
Let string $x = abba$ and Levenshtein distance $k = 1$. Find all approximate repetitions in this string.
We construct an approximate factor automaton using Levenshtein distance $k = 1$.

1. We construct a finite automaton accepting string x and all strings with Levenshtein distance equal to 1. The set of these strings is denoted $L_1(abba)$. The resulting finite automaton has the transition diagram depicted in Fig. 4.31.

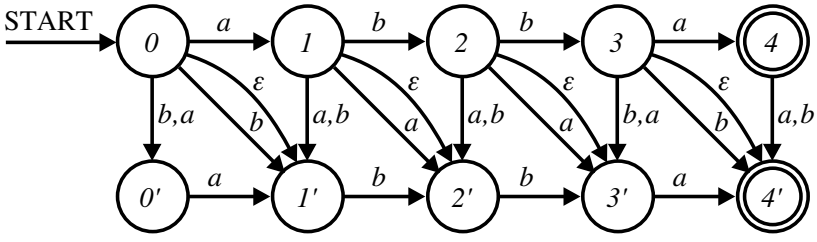


Figure 4.31: Transition diagram of the "Levenshtein" automaton accepting $L_1(abba)$, with Levenshtein distance $k = 1$ from Example 4.50

2. We use the principle of inserting the ε–transitions from state 0 to states 1,2,3, and 4 and we fix all states as final states. The transition diagram with inserted ε–transitions and fixed final states is depicted in Fig. 4.32.

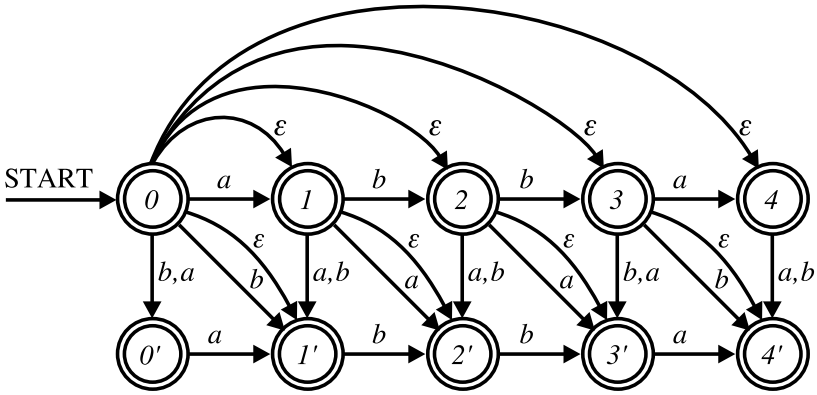


Figure 4.32: Transition diagram of the "Levenshtein" factor automaton with final states fixed and ε–transitions inserted from Example 4.50

3. We replace all ε–transitions by non–ε–transitions. The resulting automaton has the transition diagram depicted in Fig. 4.33. Its transition table is shown in Tab. 4.13

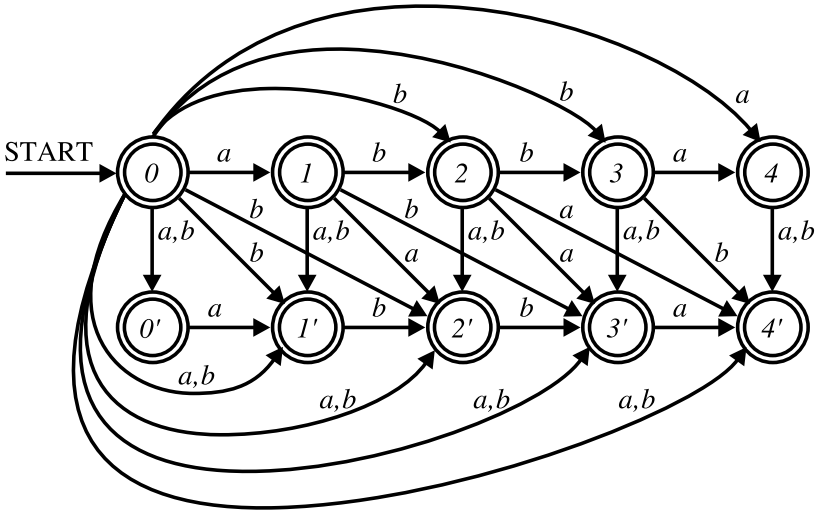


Figure 4.33: Transition diagram of the nondeterministic "Levenshtein" factor automaton after removal of ε–transitions from Example 4.50

4. The final operation is to construct the equivalent deterministic finite automaton. Its transition table is shown in Table 4.14.

	a	b
0	$140'1'2'3'4'$	$230'1'2'3'4'$
1	$1'2'$	$21'3'$
2	$2'3'4'$	$32'$
3	$43'$	$3'4'$
4	$4'$	$4'$
$0'$	$1'$	
$1'$		$2'$
$2'$		$3'$
$3'$	$4'$	
$4'$		

Table 4.13: Transition table of the nondeterministic "Levenshtein" factor automaton from Example 4.50

	a	b
0	$140'1'2'3'4'$	$230'1'2'3'4'$
$140'1'2'3'4'$	$1'2'4'$	$21'2'3'4'$
$230'1'2'3'4'$	$41'2'3'4'$	$32'3'4'$
$1'2'4'$		$2'3'$
$21'2'3'4'$	$2'3'4'$	$32'3'$
$32'3'$	$43'4'$	$3'4'$
$41'2'3'4'$	$4'$	$4'2'3'$
$32'3'4'$	$43'4'$	$3'4'$
$43'4'$	$4'$	$4'$
$2'3'$	$4'$	$3'$
$2'3'4'$	$4'$	$3'$
$3'4'$	$4'$	
$3'$	$4'$	
$4'$		

Table 4.14: Transition table of the deterministic "Levenshtein" factor automaton from Example 4.50

The transition diagram of the resulting deterministic "Levenshtein" factor automaton is depicted in Fig. 4.34.

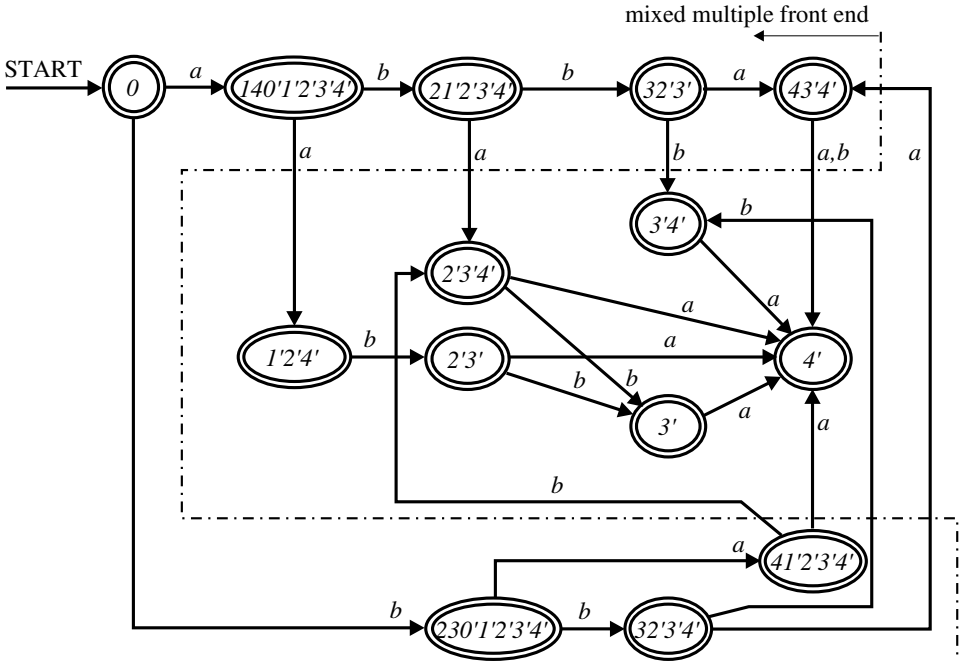


Figure 4.34: Transition diagram of the deterministic "Levenshtein" factor automaton for the string $x = abba$ from Example 4.50

Now we construct the repetition table. We take into account the repetition of factors longer than k (the number of allowed errors). Approximate repetition table R is shown in Table 4.15.

4.4.8 Approximate repetitions – Δ distance

Let us note that the Δ distance is defined by Def. 1.18. This distance is defined as the local distance for each position of the string. The number of errors is not cumulated as in the previous (and following) cases of finding approximate repetitions. In this section we show solution of $O?N\Delta C$ problem.

Example 4.51
Let string $x = abbc$ over ordered alphabet $A = \{a, b, c\}$. We construct an approximate factor automaton using Δ-distance equal to one.

1. We construct a finite automaton accepting string x and all strings having Δ-distance equal to one. The set of all these strings is denoted $\Delta_1(abbc)$. This finite automaton has the transition diagram depicted in Fig. 4.35.

d–subset	Factor	Approximate repetitions
21'2'3'4'	ab	$(2, ab, F)(3, abb, O)$
32'3'	abb	$(3, abb, F)(2, ab, O), (3, bb, O)$
43'4'	$abba$	$(4, abba, F)(3, abb, O), (4, bba, O)$
32'3'4'	bb	$(3, bb, F)(2, ab, O), (3, abb, O), (4, ba, O), (4, bba, O)$
41'2'3'4'	ba	$(4, ba, F)(3, bb, S), (4, bba, O)$
43'4'	bba	$(4, bba, F)(3, bb, O), (4, ba, O)$

Table 4.15: Approximate repetition table R for string $x = abba$ with Levenshtein distance $k = 1$ from Example 4.50

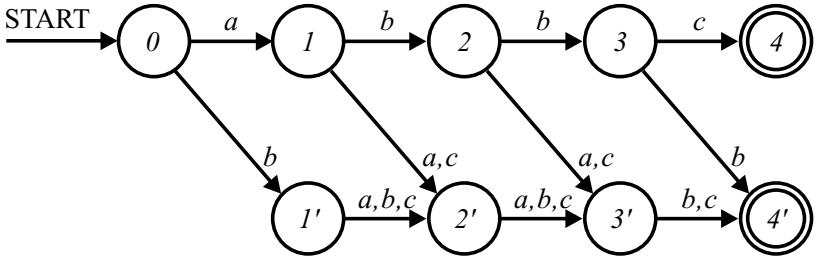


Figure 4.35: Transition diagram of the "Δ" automaton accepting $\Delta_1(abbc)$ with Δ-distance $k = 1$ from Example 4.51

2. We use the principle of inserting ε–transitions from state 0 to states 1, 2, 3, and 4 and making all states final states. The transition diagram of the automaton with inserted ε–transitions and fixed final states is depicted in Fig. 4.36.
3. We replace ε–transitions by non–ε–transitions. The resulting automaton has the transition diagram depicted in Fig. 4.37.

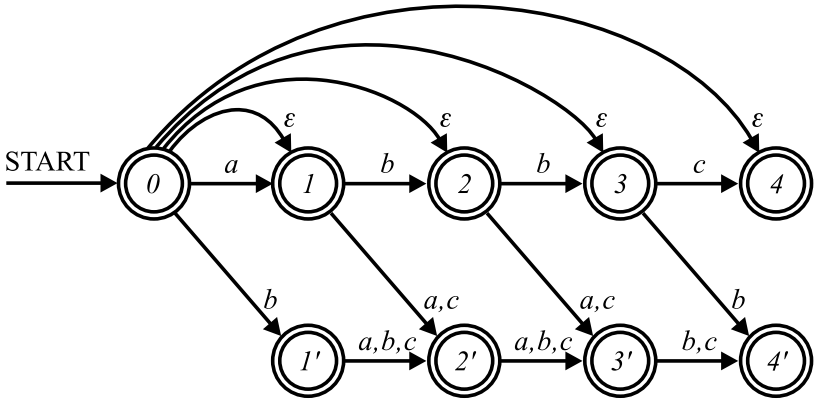


Figure 4.36: Transition diagram of the "Δ" factor automaton with final states fixed and ε–transitions inserted from Example 4.51

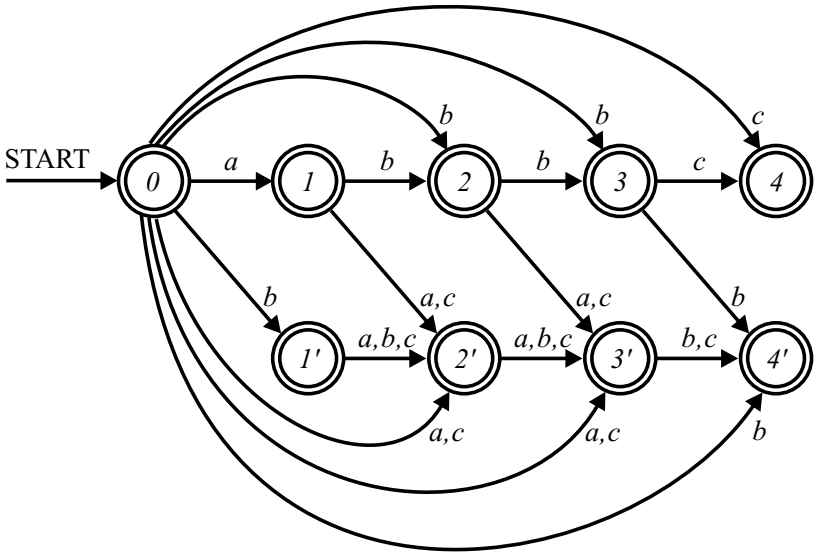


Figure 4.37: Transition diagram of the nondeterministic "Δ" factor automaton after the removal of ε–transitions from Example 4.51

4. The final operation is to construct the equivalent deterministic factor automaton. Table 4.16 is its transition table and its transition diagram is depicted in Fig. 4.38.

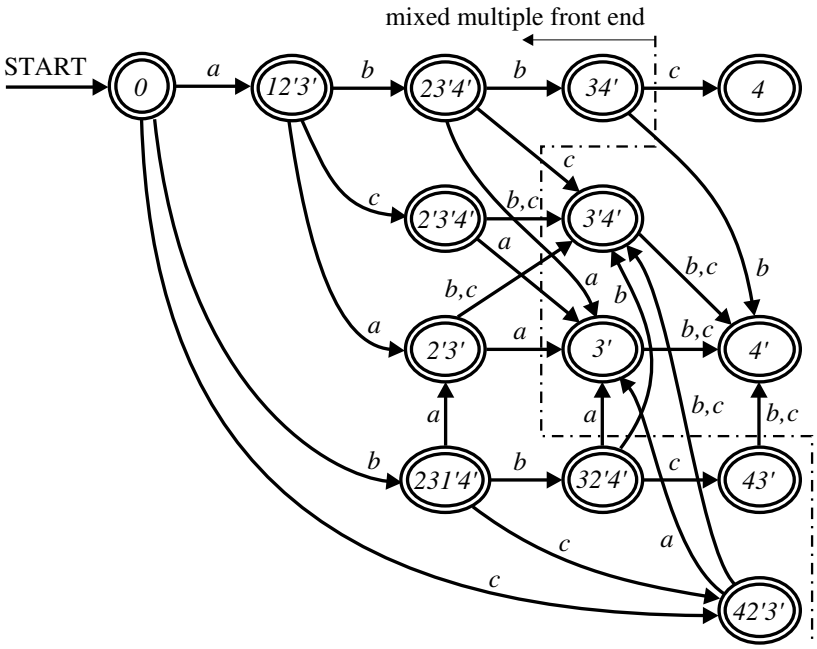


Figure 4.38: Transition diagram of the deterministic "Δ" factor automaton for $x = abbc, \Delta$-distance=1, from Example 4.51

Now we construct the repetition table. We take into account the repetitions of factors longer than the allowed distance. In this case, the distance is equal to 1 and therefore we select repetitions of factors having the length greater or equal to two. Table 4.17 contains information on the Δ–approximate repetitions of factors of the string $x = abbc$. □

4.4.9 Approximate repetitions – Γ distance

Γ-distance is defined by Def. 1.18. This distance is defined as a global distance, which means that the local errors are cumulated. In this section we show solution of $O?N\Gamma C$ problem.

Example 4.52
Let string $x = abbc$ over ordered alphabet $A = \{a, b, c\}$. We construct an approximate factor automaton using Γ-distance equal to two.

1. We construct a finite automaton accepting string x and all strings having Γ-distance equal to two. The set of all these strings is denoted $\Gamma_2(abbc)$. This finite automaton has the transition diagram depicted in Fig. 4.39.

	a	b	c
0	$12'3'$	$231'4'$	$42'3'$
$12'3'$	$2'3'$	$23'4'$	$2'3'4'$
$23'4'$	$3'$	$34'$	$3'4'$
$231'4'$	$2'3'$	$32'4'$	$42'3'$
$32'4'$	$3'$	$3'4'$	$43'$
$34'$		$4'$	4
$43'$		$4'$	$4'$
4			
$42'3'$	$3'$	$3'4'$	$3'4'$
$2'3'4'$	$3'$	$3'4'$	$3'4'$
$2'3'$	$3'$	$3'4'$	$3'4'$
$3'4'$		$4'$	$4'$
$3'$		$4'$	$4'$
$4'$			

Table 4.16: Transition table of the deterministic "Δ" factor automaton from Example 4.51

d–subset	Factor	Approximate repetitions
23'4'	ab	$(2, ab, F), (3, bb, O), (4, bc, S)$
34'	abb	$(3, abb, F), (4, bbc, O)$
32'4'	bb	$(3, bb, F), (2, ab, O), (4, bc, O)$
42'3'	bc	$(4, bc, F), (2, ab, S), (3, bb, O)$
43'	bbc	$(4, bbc, F), (3, abb, O)$

Table 4.17: Approximate repetition table for string $x = abbc$ from Example 4.51

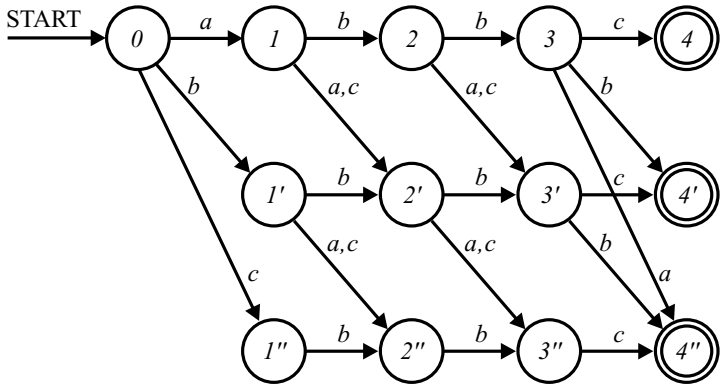


Figure 4.39: Transition diagram of the "Γ" automaton accepting $\Gamma_2(abbc)$ with Γ-distance $k = 2$ from Example 4.52

2. Now we insert ε–transitions from state 0 to states 1, 2, 3, and 4 and we make all states final. The transition diagram of the automaton with inserted ε–transitions and fixed final states is depicted in Fig. 4.40

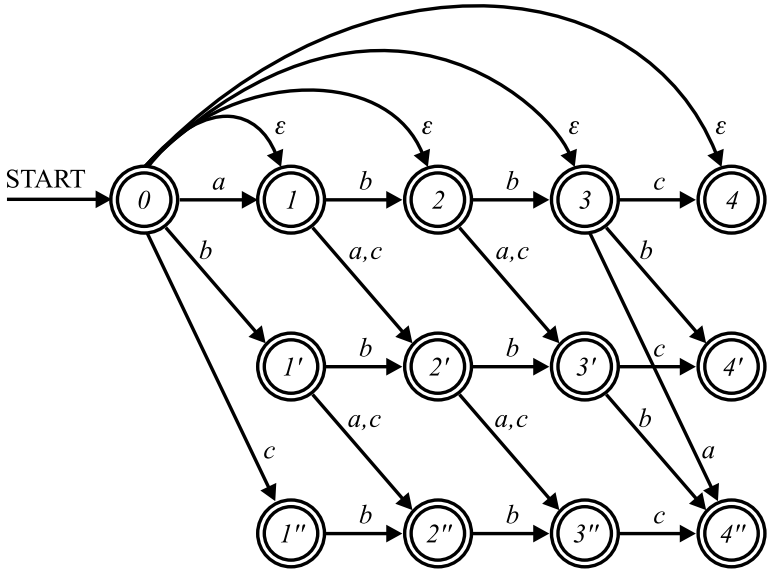


Figure 4.40: Transition diagram of the "Γ" factor automaton with final states fixed and ε–transitions inserted from Example 4.52

3. We replace the ε–transitions by non–ε–transitions. The resulting non-deterministic factor automaton has the transition diagram depicted in Fig. 4.41.
4. The final operation is to construct the equivalent deterministic finite automaton.

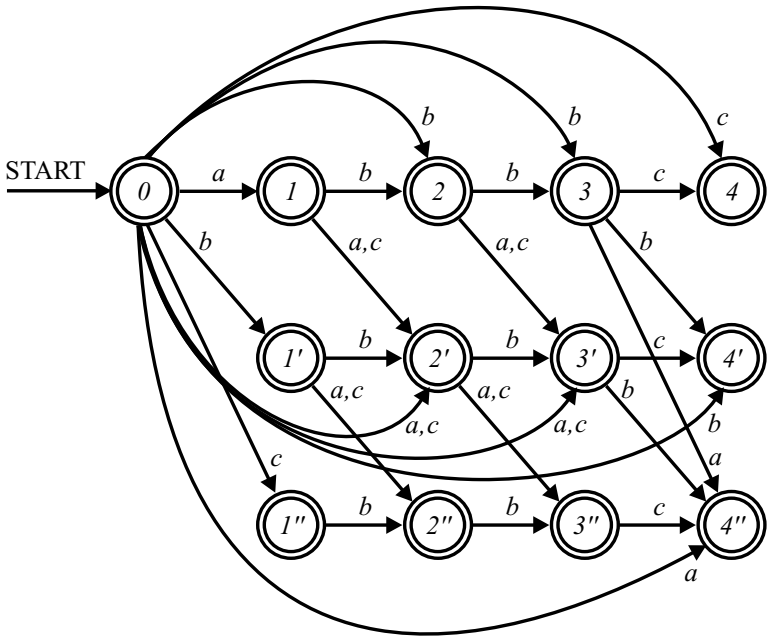


Figure 4.41: Transition diagram of the nondeterministic "Γ" factor automaton after the removal of ε–transitions from Example 4.52

Analyzing Table 4.18 we can recognize that the following sets are sets of equivalent states:

$\{2'4'3'', 2'3''\}, \{3'4', 3', 3'4''\}, \{43'', 4'3'', 3''4'', 3''\}, \{43'2'', 3'2''4''\}$.

Only states $43''$ and $43'2''$ have an impact on the repetition table. Let us replace all equivalent states by the respective sets. Then we obtain the transition diagram of the optimized deterministic "Γ" factor automaton depicted in Fig. 4.42. Now we construct the repetition table. We take into account the repetitions of factors longer than two (allowed distance). The Table 4.19 contains information on approximate repetition of one factor. The repetition of factor *bc* indicated by *d*-subset 43'2" is not included in the repetition table because the length of this factor is $|bc| = 2$ □

4.4.10 Approximate repetitions – (Δ, Γ) distance

(Δ, Γ)-distance is defined by Def. 1.18. This distance is defined as a global distance, which means that the local errors are cumulated. In this section we show solution of $O?N(\Delta, \Gamma)C$ problem.

Example 4.53
Let string $x = abbc$ over ordered alphabet $A = \{a, b, c\}$. We construct an approximate factor automaton using (Δ, Γ)-distance equal to (1,2).

	a	b	c
0	$12'3'4''$	$231'4'$	$42'3'1''$
$12'3'4''$	$2'3''$	$23'4''$	$2'4'3''$
$231'4'$	$3'2''4''$	$32'4'$	$43'2''$
$23'4''$	$3'$	$34''$	$3'4'$
$32'4'$	$3''4''$	$3'4'$	$43''$
$34''$	$4''$	$4'$	4
$42'3'1''$	$3''$	$3'2''4''$	$4'3''$
$43'2''$		$3''4''$	$4'$
$43''$			$4''$
4			
$2'4'3''$	$3''$	$3'$	$3''4''$
$2'3''$	$3''$	$3'$	$3''4''$
$3'2''4''$		$3''4''$	$4'$
$3'4'$		$4''$	$4'$
$3'$		$4''$	$4'$
$4'3''$			$4''$
$4'$			
$3''4''$			$4''$
$3'4''$		$4''$	$4'$
$3''$			$4''$
$4''$			

Table 4.18: Transition table of the deterministic "Γ" factor automaton for the string $x = abbc$ from Example 4.52

1. We construct a finite automaton accepting string x and all strings having (Δ, Γ)-distance equal to (1,2). The set of all these strings is denoted $(\Delta, \Gamma)_{1,2}(abbc)$. This finite automaton has the transition diagram depicted in Fig. 4.43.
2. Now we insert ε–transitions from state 0 to states 1, 2, 3, and 4 and we make all states final. The transition diagram of the automaton with inserted ε–transitions and fixed final states is depicted in Fig. 4.44.
3. We replace ε–transitions by non–ε–transitions. The resulting nondeterministic factor automaton has the transition diagram depicted in Fig. 4.45.

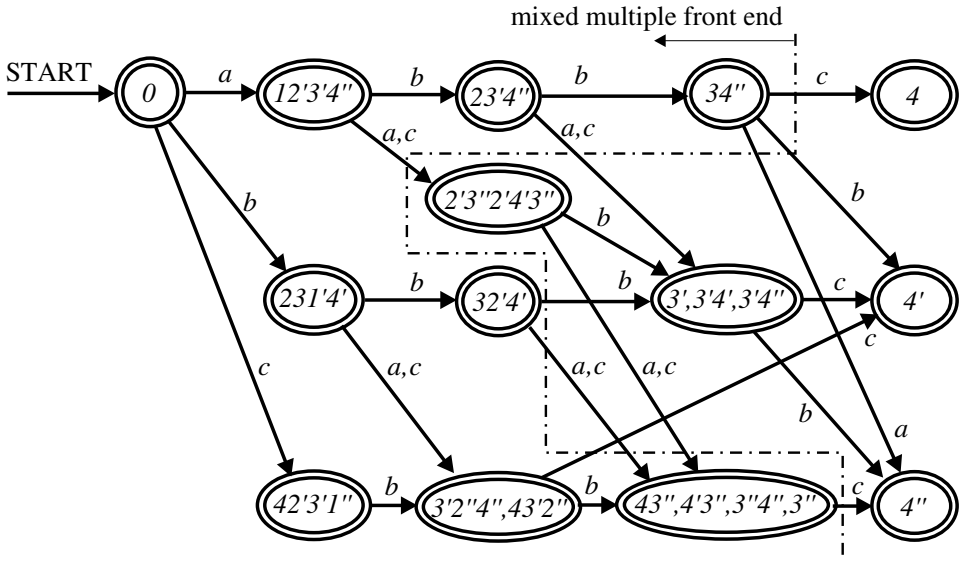


Figure 4.42: Transition diagram of the optimized deterministic "Γ" factor automaton for string $x = abbc$ from Example 4.52

d–subset	Factor	Approximate repetitions
34"	abb	$(3, abb, F), (4, bbc, O)$
43"	bbc	$(4, bbc, F), (3, abb, O)$

Table 4.19: Approximate repetition table for string $x = abbc$, Γ distance equal to two, from Example 4.52

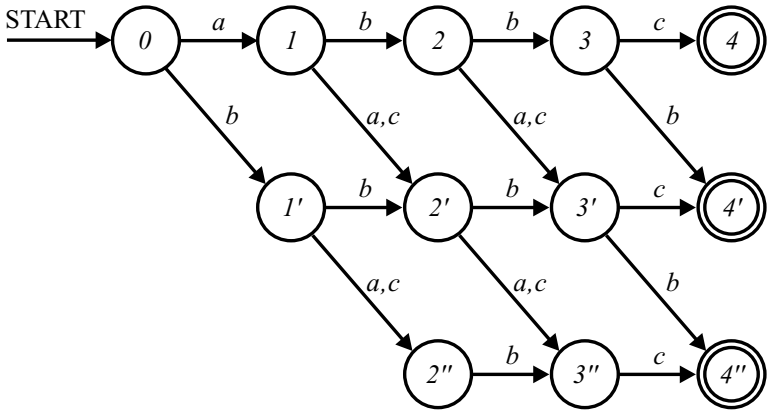


Figure 4.43: Transition diagram of the "(Δ, Γ)" automaton accepting $(\Delta, \Gamma)_{1,2}(abbc)$ with (Δ, Γ)-distance $(k, l) = (1, 2)$ from Example 4.53

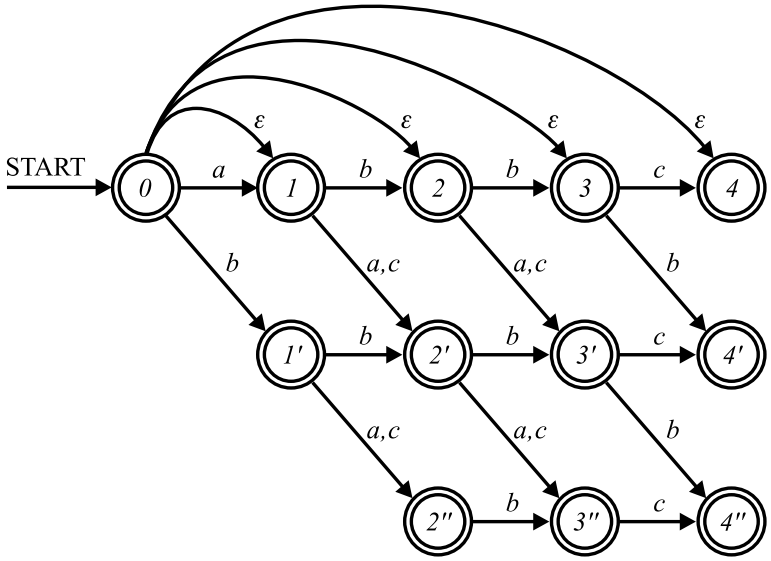


Figure 4.44: Transition diagram of the "(Δ, Γ)" factor automaton with final states fixed and ε–transitions inserted from Example 4.53

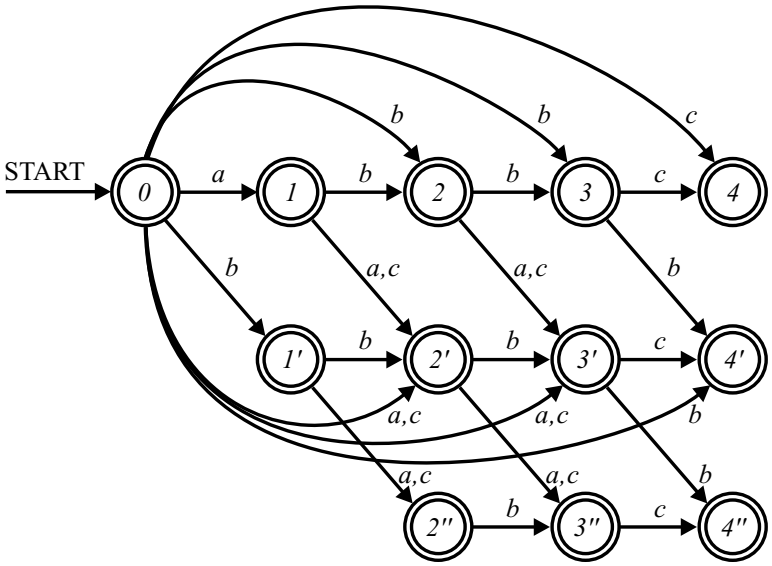


Figure 4.45: Transition diagram of the nondeterministic "(Δ, Γ)" factor automaton after the removal of ε–transitions from Example 4.53

4. The final operation is to construct the equivalent deterministic finite automaton. Table 4.20 is its transition table.

	a	b	c
0	$12'3'$	$231'4'$	$42'3'$
$12'3'$	$2'3''$	$23'$	$2'3''$
$231'4'$	$3'2''$	$32'4'$	$43'2''$
$23'$	$3'$	$34''$	$3'4'$
$32'4'$	$3''$	$3'4'$	$43''$
$34''$		$4'$	4
$42'3'$	$3''$	$3'4''$	$4'3''$
$43'2''$		$3''4''$	$4'$
$43''$			$4''$
4			
$2'3''$	$3''$	$3'$	$3''4''$
$3'2''$		$3''4''$	$4'$
$3'4'$		$4''$	$4'$
$3'4''$		$4''$	$4'$
$3'$		$4''$	$4'$
$4'3''$			$4''$
$4'$			
$3''4''$			$4''$
$3''$			$4''$
$4''$			

Table 4.20: Transition table of the deterministic "(Γ, Δ)" factor automaton from Example 4.53

Analyzing Table 4.20 we can recognize that the following sets are sets of equivalent states:

$\{3'4', 3', 3'4''\}, \{3''4'', 3'', 43'', 4'3''\}, \{43'2'', 3'2''\}$.

We replace all equivalent states by the respective sets. Then we obtain the transition diagram of the optimized deterministic "(Δ, Γ)" factor automaton depicted in Fig. 4.46. Now we construct the repetition table. We take into account the repetitions of factors longer than two (allowed distance). Table 4.21 contains information on approximate repetition of one factor. □

4.4.11 Exact repetitions in one string with don't care symbols

The "don't care" symbol ($\circ$) is defined by Def. 1.14. Next example shows principle of finding repetitions in the case of presence of don't care symbols (*O?NED* problem).

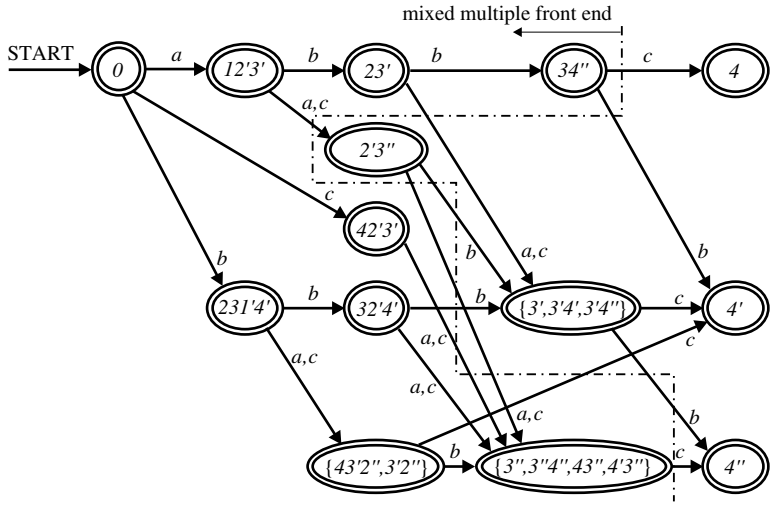


Figure 4.46: Transition diagram of the optimized deterministic "(Δ, Γ)" factor automaton for the string $x = abbc$ from Example 4.53

d–subset	Factor	Approximate repetitions
34"	abb	$(3, abb, F), (4, bbc, O)$
43"	bbc	$(4, bbc, F), (3, abb, O)$

Table 4.21: Approximate repetition table for string $x = abbc$, (Γ, Δ) distance equal to (1,2), from Example 4.53

Example 4.54
Let string $x = a \circ aab$ over alphabet $A = \{a, b, c\}$. Symbol $\circ$ is the don't care symbol. We construct a don't care factor automaton.

1. We construct a finite automaton accepting set of strings described by string x with don't care symbol. This set is $DC(x) = \{aaaab, abaab, acaab\}$. This finite automaton has the transition diagram depicted in Fig. 4.47.

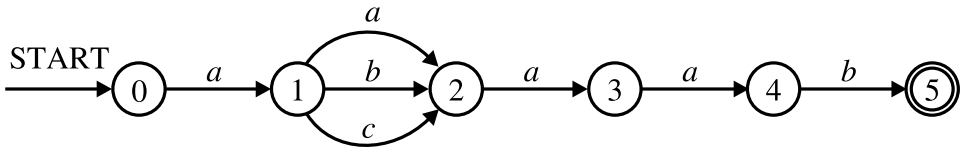


Figure 4.47: Transition diagram of the DC automaton accepting $DC(x)$ from Example 4.54

2. We insert ε-transitions from state 0 to states 1,2,3,4, and 5 and we

make all states final. Transition diagram of $DC_\varepsilon(a \circ aab)$ factor automaton with inserted ε-transitions and fixed final states is depicted in Fig. 4.48.

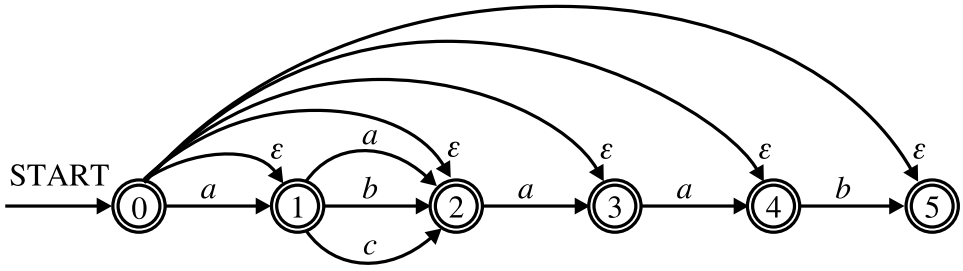


Figure 4.48: Transition diagram of the $DC_\varepsilon(a \circ aab)$ factor automaton with inserted ε-transitions and fixed final states from Example 4.54

3. We replace ε-transitions by non ε-transitions. Resulting nondeterministic factor automaton $DC_N(a \circ aab)$ has the transition diagram depicted in Fig. 4.49.

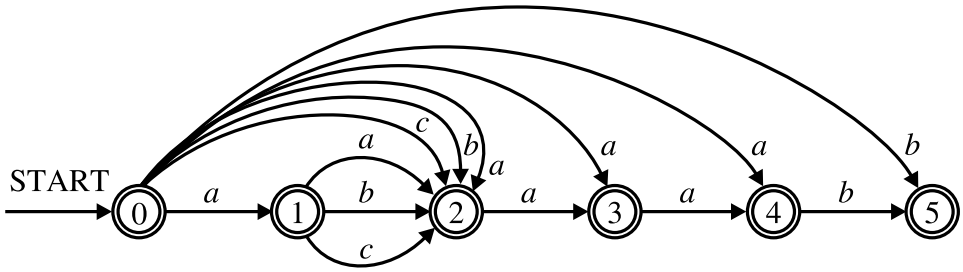


Figure 4.49: Transition diagram of nondeterministic factor automaton $DC_N(a \circ aab)$ after the removal of ε-transitions from Example 4.54

4. The final operation is construction of equivalent deterministic factor automaton $DC_D(a \circ aab)$. Table 4.22 is transition table of the nondeterministic factor automaton having transition diagram depicted in Fig. 4.49. Table 4.23 is transition table of deterministic factor automaton $DC_D(a \circ aab)$. Transition diagram of deterministic factor automaton $DC_D(a \circ aab)$ is depicted in Fig. 4.50.

	a	b	c
0	$1, 2, 3, 4$	$2, 5$	2
1	2	2	2
2	3		
3	4		
4		5	
5			

Table 4.22: Transition table of nondeterministic factor automaton $DC_N(a \circ aab)$ from Example 4.54

	a	b	c
0	1234	25	2
1234	234	25	2
2	3		
25	3		
234	34	5	
3	4		
34	4	5	
4		5	
5			

Table 4.23: Transition table of deterministic factor automaton $DC_D(a \circ aab)$ from Example 4.54

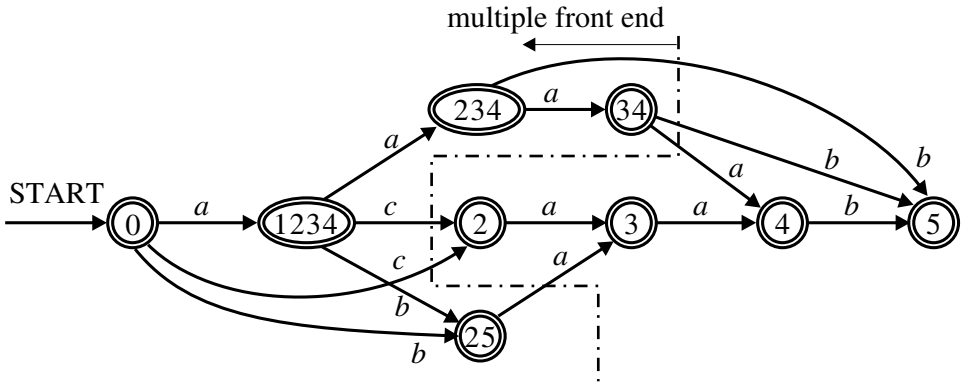


Figure 4.50: Transition diagram of deterministic factor automaton $DC_D(a \circ aab)$ from Example 4.54

The last step is the construction of the repetition table. It is shown in Table 4.24.

d–subset	Factor	Repetitions
1234	a	$(1,F),(2,S),(3,S),(4,S)$
25	ab	$(2,F),(5,G)$
234	aa	$(2,F),(3,O),(4,O)$
34	aaa	$(3,F),(4,O)$

Table 4.24: Repetition table for string $x = a \circ aab$ from Example 4.54

4.5 Computation of periods revisited

Let us remind the definition of a period (see Def. 4.4). We have seen in Section 4.2.2 the principle of computation of periods in the framework of computation of borders. Now we show the computation of periods without regard to the computation of borders. For the computation of periods of given string we can use the backbone of a factor automaton and we will construct the repetition table. Such repetition table is a *prefix repetition table*. The next Algorithm shows, how to find periods of a string.

Algorithm 4.55
Input: String $x \in A^+$.
Output: Sequence of periods $p_1, p_2, \ldots, p_h,\ h \geq 0$, and the shortest period $Per(x)$.
Method:

1. Construct factor automaton M for string x.

2. Compute the prefix repetition table using the backbone of the deterministic factor automaton for string x.
3. Do inspection of the prefix repetition table in order to find all rows, where squares are indicated. If these squares covers the prefix of x long enough, then the lengths of repeated strings are the periods of x. More precisely:
   If the prefix of length m is repeating as squares
   $(m, F), (2 * m, S), (3 * m, S), \ldots, (j * m, S)$,
   $|x| - j * m \leq m$, and $x[j * m + 1..|x|] \in \mathit{Pref}(x[1..m])$, then m is the period of x.
4. Do inspection of the prefix repetition table in order to find all rows where are repetitions of prefixes longer than one half of string x with overlapping. If such prefix has length $m > |x|/2$ and the suffix of x $x[m + 1..|x|] \in \mathit{Pref}(x[1..m])$, then m is the period of x.
5. $|x|$ is also the period of x.
6. The shortest period is $\mathit{Per}(x)$. □

Note: If $\mathit{Per}(x) = |x|$ then x is primitive string.

Example 4.56
Let us compute periods of string $x = abababababa$. The nondeterministic factor automaton has the transition diagram depicted in Fig. 4.51. Transition table of the deterministic factor automaton is shown in Table 4.25. The backbone of the deterministic factor automaton has the transition diagram depicted in Fig. 4.52. The prefix repetition table is Table 4.26.

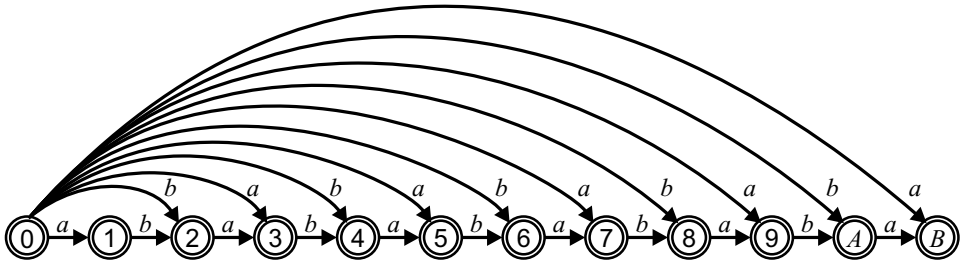


Figure 4.51: The nondeterministic factor automaton for string $x = abababababa$ from Example 4.56, A and B represent numbers 10 and 11, respectively

1. During inspection of this table we find two rows with squares: row 2 and row 4. In the row 2 we see, that the prefix ab is repeated four times: $(4, S), (6, S), (8, S)$, and (A, S). The rest of the string is a which is the prefix of ab. Therefore the first period is 2. In row 4 we see,

	a	b
0	$13579B$	$2468A$
$13579B$		$2468A$
$2468A$	$3579B$	
$3579B$		$468A$
$468A$	$579B$	
$579B$		$68A$
$68A$	$79B$	
$79B$		$8A$
$8A$	$9B$	
$9B$		A
A	B	
B		

Table 4.25: Transition table of the deterministic factor automaton from Example 4.56

d–subset	*Prefix*	Repetitions of prefixes
$13579B$	a	$(1,F),(3,G),(5,G),(7,G),(9,G),(B,G)$
$2468A$	ab	$(2,F),(4,S),(6,S),(8,S),(A,S)$
$3579B$	aba	$(3,F),(5,O),(7,O),(9,O),(B,O)$
$468A$	$abab$	$(4,F),(6,O),(8,S),(A,O)$
$579B$	$ababa$	$(5,F),(7,O),(9,O),(B,O)$
$68A$	$ababab$	$(6,F),(8,O),(A,O)$
$79B$	$abababa$	$(7,F),(9,O),(B,O)$
$8A$	$ababbab$	$(8,F),(A,O)$
$9B$	$abababaa$	$(9,F),(B,O)$

Table 4.26: The prefix repetition table from Example 4.56

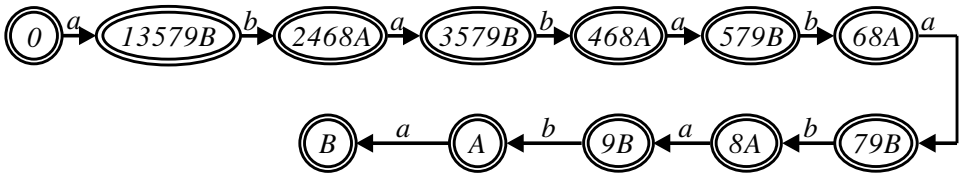


Figure 4.52: Transition diagram of the backbone of the deterministic factor automaton for string $x = abababababa$ from Example 4.56

that the prefix $abab$ is repeated once: $(8, S)$. The rest of the string is aba which is the prefix of $abab$. Therefore the second period is 4.

2. Moreover, there are four cases with overlapping of prefixes longer than 5:
   (6) $abababab$, $ababa$
   (7) $abababa$, $baba$
   (8) $abababab$, aba
   (9) $ababababa$, ba
   (10) $ababababab$, a

   Suffixes of cases 6, 8, 10 are prefixes of $x : ababa, aba, a$. It means that 6, 8 and 10 are periods of x, too.
3. The last period of x is $|x|$.

The set of all periods is $\{2, 4, 6, 8, 10, 11\}$. $Per(x) = 2$. □

Example 4.57
Let us compute periods of string $x = a^4$. Ttransition diagram of the nondeterministic factor automaton is depicted in Fig. 4.53. Transition diagram

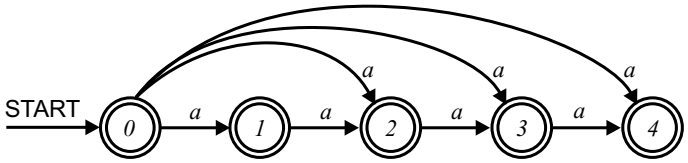


Figure 4.53: Transition diagram of nondeterministic factor automaton for string $x = a^4$ from Example 4.57

of the backbone of deterministic factor automaton is depicted in Fig. 4.54.

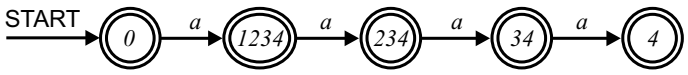


Figure 4.54: Transition diagram of the backbone of deterministic factor automaton for string $x = a^4$ from Example 4.57

The prefix repetition table has the form:

d–subset	Prefix	Repetitions of prefixes
1234	a	$(1,F),(2,S),(3,S),(4,S)$
234	aa	$(2,F),(3,O),(4,S)$
34	aaa	$(3,F),(4,O)$

We see that there are four periods: 1, 2, 3, 4. $Per(a^4) = 1$.
Transition table of the deterministic factor automaton is shown in Table 4.27.

	a
0	1234
1234	234
234	34
34	4

Table 4.27: Transition table of the deterministic factor automaton from Example 4.57

Example 4.58
Let us compute periods of string $x = abcd$. Transition diagram of the factor automaton is depicted in Fig. 4.55. This factor automaton is deterministic.

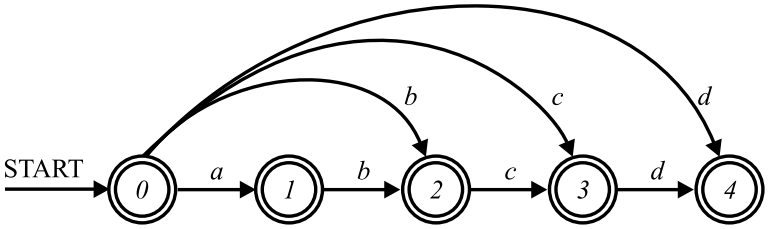


Figure 4.55: Transition diagram of the factor automaton for string $x = abcd$

It means, that nothing is repeated and the shortest period is equal to the length of string and therefore string $x = abcd$ is the primitive string (see Def. 4.6).

5 Simulation of nondeterministic pattern matching automata – fail function

Deterministic pattern matching automata have in some cases large space complexity. This is especially true for automata for approximate pattern matching. This situation led to the construction of algorithms for the simulation of nondeterministic pattern matching automata. These simulation algorithms have an acceptable space complexity but theirs time complexity is, in some cases, greater than linear. We can divide methods used for simulation of nondeterministic pattern matching automata into three categories:

1. using fail function,
2. dynamic programming,
3. bit parallelism.

The use of the fail function we will discuss in this Chapter. The dynamic programming and bit parallelism will be covered in the next Chapter.

5.1 Searching automata

The group of methods which use fail function is based on the following principle:

The nondeterministic pattern matching automaton is used but the minimum number of its selfloops in the initial state are removed in order to obtain deterministic finite automaton. If this operation succeed and the resulting finite automaton is deterministic, then it can be used. Afterwards some transitions called backward transitions are added. No input symbol is read during such transitions. They are used in case when the forward transitions cannot be used. Well known Morris–Pratt (MP), Knuth–Morris–Pratt (KMP) and Aho–Corasick (AC) algorithms belong to this category.

The base of algorithms of this category is a notion of *searching automaton* which is an extended deterministic finite automaton.

Definition 5.1
Searching automaton is sixtuple $SA = (Q, A, \delta, \varphi, q_0, F)$, where
Q is a finite set of states,
A is a finite input alphabet,
$\delta : Q \times A \to Q \cup \{fail\}$ is the forward transition function,
$\varphi : (Q - \{q_0\}) \times A^* \to Q$ is the backward transition function,
q_0 is the initial state,
$F \subset Q$ is the set of final states.
A configuration of searching automaton SA is pair (q, w), where $q \in Q, w \in A^*$. The initial configuration is (q_0, w), where w is the complete input text. The final configuration is (q, w), where $q \in F$ and $w \in A^*$ is an unread part of the text. This configuration means that the pattern was found and its

position is in the text just before w. The searching automaton performs forward and backward transitions. Transition relation

$\vdash \subset (Q \times A^*) \times (Q \times A^*)$

is defined in this way:

1. if $\delta(q, a) = p$, then $(q, aw) \vdash (p, w)$ is a forward transition,
2. if $\varphi(q, x) = p$, then $(q, w) \vdash (p, w)$ is a backward transition, where x is the suffix of the part of the text read before reaching state q. □

Just one input symbol is read during forward transition. If $\delta(q, a) = fail$ then backward transition is performed and no symbol is read. Forward and backward transition functions δ and φ have the following properties:

1. $\delta(q_0, a) \neq fail$ for all $a \in A$,
2. If $\varphi(q, x) = p$ then the depth of p is strictly less than the depth of q, where the depth of state q is the length of the shortest sequence of forward transitions from state q_0 to state q.

The first condition ensures that no backward transition is performed in the initial state. The second condition ensures that the total number of backward transitions is less than the number of forward transitions. It follows that the total number of performed transitions is less than $2n$, where n is the length of the text.

5.2 *MP* and *KMP* algorithms

MP and *KMP* algorithms are the simulators of the *SFOECO* automaton (see Section 2.2.1) for exact matching of one pattern. We will show both algorithms in the following example. Let us mention, that the backward transition function is simplified and

$\varphi : (Q - \{q_0\}) \rightarrow Q$.

Example 5.2
Let us construct *MP* and *KMP* searching automata for pattern $P = ababb$ and compare it with pattern matching automaton for P. The construction of both, deterministic pattern matching automaton and *MP* searching automaton is shown in Fig. 5.1. We can construct, for the resulting *MP* and *KMP* searching automata, the following Table 5.1 containing forward transition function δ, backward transition function φ for *MP* algorithm, and optimized backward transition function φ_{opt} for *KMP* algorithm. The reason for the introduction of the optimized backward transition function φ_{opt} will follow from the next example. □

Example 5.3
The *MP* searching automaton for pattern $P = ababb$ and text $T = abaababbb$ performs the following sequence of transitions (backward transition function φ is used):

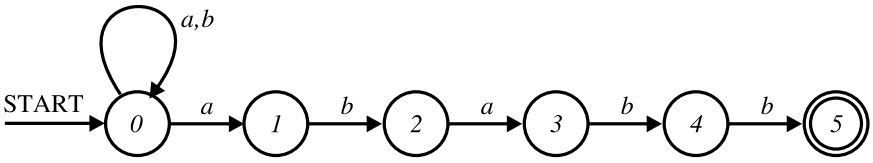
δ	a	b	φ	φ_{opt}
0	1	0		
1	$fail$	2	0	0
2	3	$fail$	0	0
3	$fail$	4	1	0
4	$fail$	5	2	2
4	$fail$	$fail$	0	0

Table 5.1: Forward transition function δ, backward transition functions φ and φ_{opt} from Example 5.2

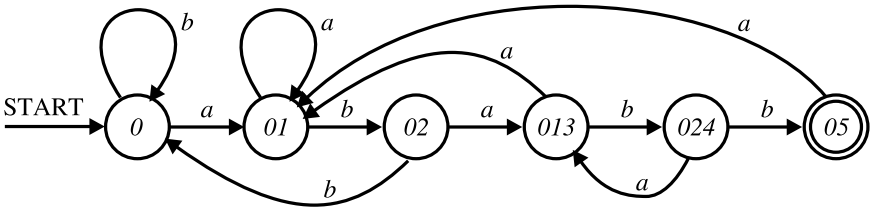
$$
\begin{aligned}
(0, abaababbb) &\vdash (1, baababbb) \\
&\vdash (2, aababbb) \\
&\vdash (3, ababbb)\ fail \\
&\vdash (1, ababbb)\ fail \\
&\vdash (0, ababbb) \\
&\vdash (1, babbb) \\
&\vdash (2, abbb) \\
&\vdash (3, bbb) \\
&\vdash (4, bb) \\
&\vdash (5, b)
\end{aligned}
$$

The pattern is found in state 5 at position 8. □

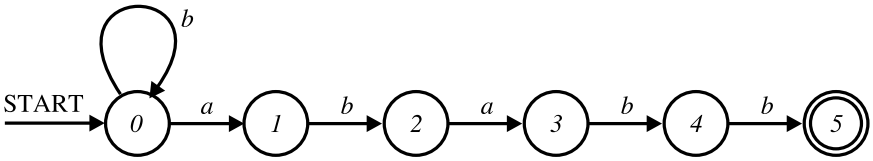
Let us mention one important observation. We can see, that there are performed two subsequent backward transitions from state 3 for the input symbol a leading to states 1 and 0. The reason is that $\delta(3, a) = \delta(1, a) = fail$. This variant of searching automaton is called MP (Morris–Pratt) automaton and the related algorithm shown below is called MP algorithm. There is possible to compute optimized backward transition function φ_{opt} having in this situation value $\varphi_{opt}(3) = 0$. The result is, that each such sequence of backward transitions is replaced by just one backward transition. The algorithm using optimized backward transition function φ_{opt} is KMP (Knuth–Morris–Pratt) algorithm. After this informal explanation, we show the direct construction of MP searching automaton, computation of backward and optimized backward transition functions. We start with MP searching algorithm.



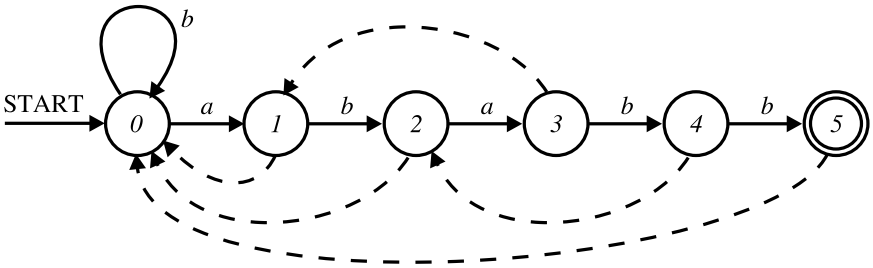
a) the nondeterministic pattern matching automaton *(SFOECO)* for pattern *P=ababb*



b) the deterministic pattern matching automaton for pattern *P=ababb*



c) the forward transition function of *MP* and *KMP* searching automata for pattern *P=ababb*



d) complete *MP* searching automaton, backward transition function φ is shown by dashed lines

Figure 5.1: *SFOECO* and *MP* searching automata for pattern $P = ababb$ from Example 5.2

```
var TEXT:array[1..N] of char;
    PATTERN:array[1..M] of char;
    I,J:  integer;
    FOUND: boolean;
    ...


    I:=1; J:=1;
    while (I <= N) and (J <= M) do
      begin
        while (TEXT[I] <> PATTERN[J]) and (J > 0) do J:=PHI[J];
        J := J + 1;
        I := I + 1
        FOUND := J > M;
      end;
    ...
```

Variables used in *MP* searching algorithm are shown in Figure 5.2. The com-

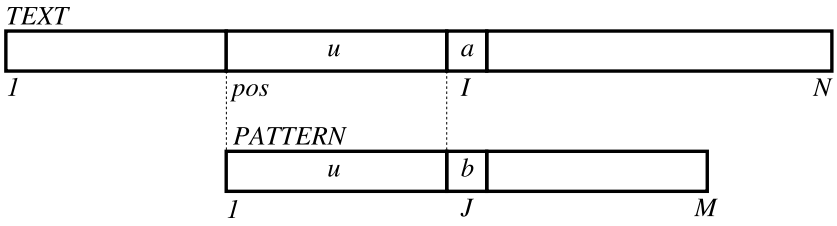


Figure 5.2: Variables used in *MP* searching algorithm, $pos = I - J + 1$

putation of backward transition function is based on the notion of repetitions of prefixes of the pattern in the pattern itself. The situation is depicted in Fig. 5.3. If prefix $u = u_1 u_2 \dots u_{j-1}$ of the pattern matches the substring of

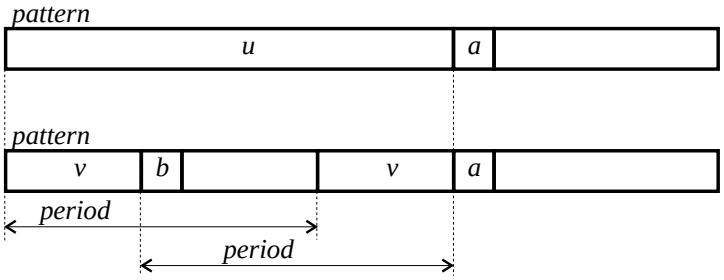


Figure 5.3: Repetition of prefix v in prefix u of the pattern

the text $u = t_{i-j+1} t_{i-j} \dots t_{i-1}$ and $u_j \neq t_i$ then it is not necessary to compare prefix v of the pattern with substring $t_{i-j+2} t_{i-j+3} \dots$ of text string at

the next position. Instead of this comparison we can do shift of the pattern to the right. The length of this shift is the length of $Border(u)$.

Example 5.4
Let us show repetitions and periods of prefixes of pattern $P = ababb$ in Fig. 5.4. The shift is represented by the value of backward transition function

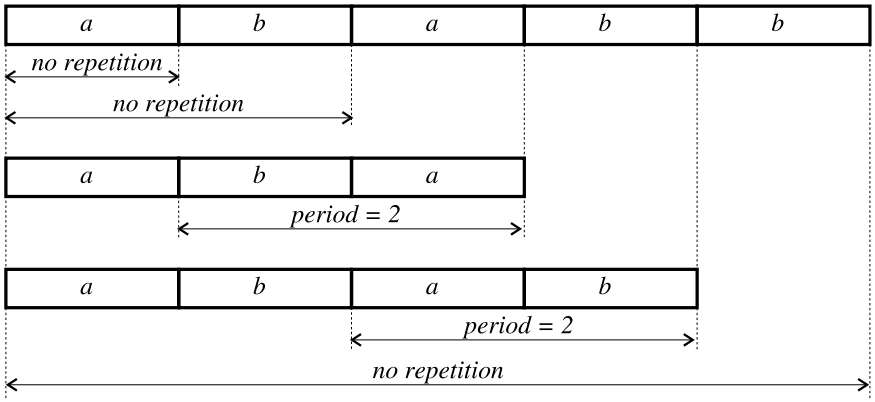


Figure 5.4: Repetitions and periods of $P = ababb$

φ for position j in the pattern is
$\varphi(j) = |Border(p_1 p_2 \dots p_j)|$.
If there is no repetition of the prefix of the pattern in itself, then the shift is equal to j, because the period is equal to zero. □

For the computation of the function φ for the pattern P we will use the fact, that the value of $\varphi(j)$ is equal to the element of the border array $\beta[j]$ for the pattern P.

Example 5.5
Let us compute the border array for pattern $P = ababb$. Transition diagram of the nondeterministic factor automaton is depicted in Fig. 5.5. Table 5.2 is

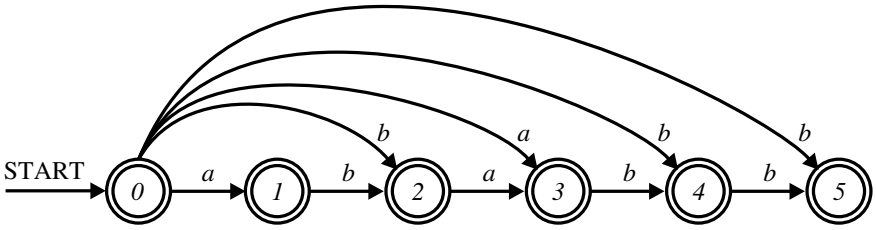


Figure 5.5: Transition diagram of the nondeterministic factor automaton for pattern $P = ababb$ from Example 5.5

transition table of the equivalent deterministic factor automaton. Transition diagram of the deterministic factor automaton is depicted in Fig. 5.6.

	a	b
0	13	245
13		24
24	3	5
245	3	5
3		4
4		5
5		

Table 5.2: Transition table of the deterministic factor automaton from Example 5.5.

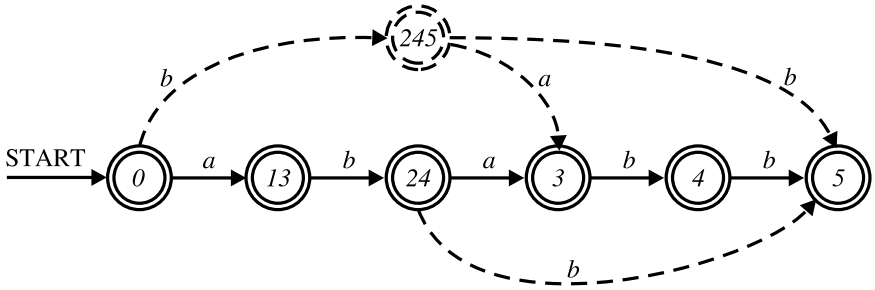


Figure 5.6: Transition diagram of the deterministic factor automaton for pattern $P = ababb$ from Example 5.2

The analysis od d–subsets on the backbone of deterministic factor automaton is shown in this table:

Analyzed state	Value of border array element
13	$\beta[3] = 1$
24	$\beta[4] = 2$

Values of elements of border array are shown in this table:

j	1	2	3	4	5
$symbol$	a	b	a	b	a
$\beta[j]$	0	0	1	2	0

Let us recall, that $\varphi(j) = \beta[j]$. □

The next algorithm constructs MP searching automaton.

Algorithm 5.6
Construction of MP searching automaton.
Input: Pattern $P = p_1 p_2 \dots p_m$.

Output: *MP* searching automaton.
Method:

1. The initial state is q_0.
2. Each state q of *MP* searching automaton corresponds to prefix $p_1p_2 \ldots p_j$ of the pattern. $\delta(q, p_{j+1}) = q'$, where q' corresponds to prefix $p_1p_2 \ldots p_jp_{j+1}$.
3. The state corresponding to complete pattern $p_1p_2 \ldots p_m$ is the final state.
4. Define $\delta(q_0, a) = q_0$ for all a for which no transitions was defined in step 2.
5. $\delta(q, a) = fail$ for all $a \in A$ and $q \in Q$ for which $\delta(q, a)$ was not defined in steps 2 and 3.
6. Function φ is the backward transition function. This is equal to the border array β for pattern P. □

The next algorithm computes optimized backward transition function φ_{opt} on the base of backward transition function φ.

Algorithm 5.7
Computation of optimized backward transition function φ_{opt}.
Input: Backward transition function φ.
Output: Optimized backward transition function φ_{opt}.
Method:

1. $\varphi_{opt} = \varphi$ for all states of the depth equal to one.
2. Let us suppose that the function φ_{opt} has been computed for all states having the depth less or equal to d. Let q has the depth equal to $d+1$. Let us suppose that $\varphi(q) = p$. If $\delta(q, a) = fail$ and $\delta(p, a) = fail$ then $\varphi_{opt}(q) = \varphi_{opt}(\varphi(q))$ else $\varphi_{opt}(q) = \varphi(q)$. □

Example 5.8
Let us construct *KMP* searching automaton for pattern $P = abaabaa$ and alphabet $A = \{a, b\}$. First, we construct the forward transition function. It is depicted in Fig. 5.7. Now we will construct both, the nonoptimized

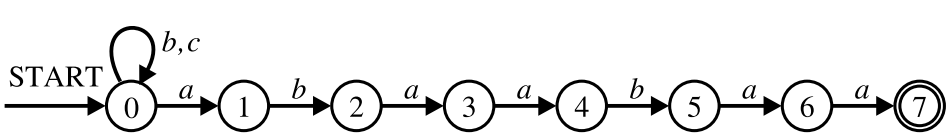


Figure 5.7: Forward transition function of *KMP* searching automaton for pattern $P = abaabaa$ from Example 5.8

backward transition function φ and optimized backward transition function φ_{opt} using Algorithms 4.19 and 5.7. To construct the nonoptimized backward transition function, we construct the border array β for pattern $P = abaabaa$. The nondeterministic factor automaton has the transition diagram depicted in Fig. 5.8. The transition diagram of the useful part

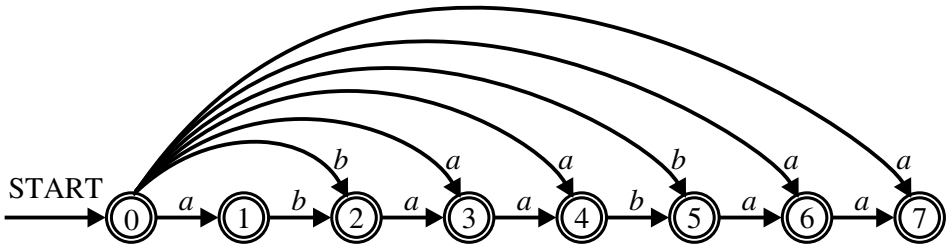


Figure 5.8: Transition diagram of the nondeterministic factor automaton for pattern $P = abaabaa$ from Example 5.8

of the deterministic factor automaton is depicted in Fig. 5.9. The analy-

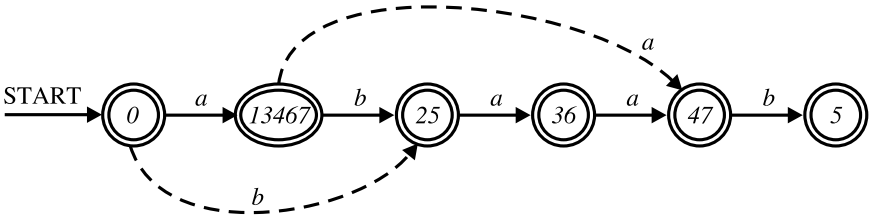


Figure 5.9: Part of the transition diagram of the deterministic factor automaton for pattern $P = abaabaa$ from Example 5.8

sis of d–subsets of the deterministic factor automaton is summarized in the Table 5.3.

d–subset	Values of elements of border array
13467	$\beta[3] = 1, \beta[4] = 1, \beta[5], = 1, \beta[6] = 1, \beta[7] = 1$
25	$\beta[5] = 2$
36	$\beta[6] = 3$
47	$\beta[7] = 4$

Table 5.3: Computation of the border array for pattern $P = abaabaa$ from Example 5.8

The resulting border array is in the Table 5.4: The values $\beta[1]$ and $\beta[2]$

$Index$	1	2	3	4	5	6	7
$Symbol$	a	b	a	a	b	a	a
β	0	0	1	1	2	3	4

Table 5.4: The border array for pattern $P = abaabaa$ from Example 5.8

are equal to zero because strings a and ab have borders of zero length. This border array is equal to the backward transition function φ. The optimized backward transition function φ_{opt} is for some indices different than function φ. Both functions φ and φ_{opt} have values according to the Table 5.5. The

I	$\varphi[I]$	$\varphi_{opt}[I]$
1	0	0
2	0	0
3	1	1
4	1	0
5	2	0
6	3	1
7	4	4

Table 5.5: Functions φ and φ_{opt} for pattern $P = abaabaa$ from Example 5.8

complete MP and KMP automata are depicted in Fig. 5.10. Let us have

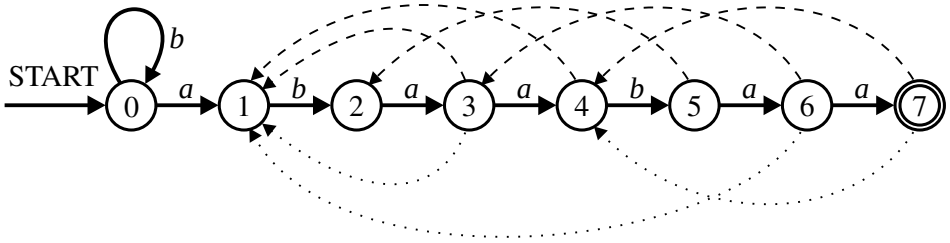


Figure 5.10: KMP searching automaton for pattern $P = abaabaa$ from Example 5.8; nontrivial values of φ are shown by dashed lines, nontrivial values of φ_{opt} are shown by dotted lines, trivial values of both functions are leading to the initial state 0

text starting with prefix $T = abaabac\ldots$. We show the behaviours of both variants of KMP automaton. The first one is MP variant and it is using φ.

$$
\begin{aligned}
(0, abaabac\ldots) &\vdash (1, baabac\ldots)\\
&\vdash (2, \ aabac\ldots)\\
&\vdash (3, \ \ abac\ldots)\\
&\vdash (4, \ \ \ bac\ldots)\\
&\vdash (5, \ \ \ \ ac\ldots)\\
&\vdash (6, \ \ \ \ \ c\ldots) fail\\
&\vdash (3, \ \ \ \ \ c\ldots) fail\\
&\vdash (1, \ \ \ \ \ c\ldots) fail\\
&\vdash (0, \ \ \ \ \ c\ldots)\\
&\vdash (0, \ \ \ \ \ \ \ldots)\\
&\vdash \ \ldots
\end{aligned}
$$

The second one is *KMP* variant using φ_{opt}.

$$
\begin{aligned}
(0, abaabac\ldots) &\vdash (1, baabac\ldots)\\
&\vdash (2, \ aabac\ldots)\\
&\vdash (3, \ \ abac\ldots)\\
&\vdash (4, \ \ \ bac\ldots)\\
&\vdash (5, \ \ \ \ ac\ldots)\\
&\vdash (6, \ \ \ \ \ c\ldots) fail\\
&\vdash (1, \ \ \ \ \ c\ldots) fail\\
&\vdash (0, \ \ \ \ \ c\ldots)\\
&\vdash (0, \ \ \ \ \ \ \ldots)\\
&\vdash \ \ldots
\end{aligned}
$$

We can see from these two sequences of transitions, that *MP* variant compares symbol c 4 times and *KMP* variant compares symbol c 3 times. □

Both variants of *KMP* algorithm have linear time and space complexities. If we have text $T = t_1 t_2 \ldots t_n$ and pattern $P = p_1 p_2 \ldots p_m$ then *KMP* algorithm requires [Cro97]:

$2m - 3$	symbol comparisons during preprocessing phase (computation of function φ or φ_{opt}),
$2n - 1$	symbol comparisons during searching phase,
m	elements of memory to store the values of function φ or φ_{opt}.

The final result is that the time complexity is $\mathcal{O}(n + m)$ and the space complexity is $\mathcal{O}(m)$. Let us note, that this complexity is not influenced by the size of alphabet on the contrary with the deterministic finite automata.

5.3 *AC* algorithm

The *AC* (Aho–Corasick) algorithm is the simulator of the *SFFECO* automaton (see Section 2.2.5) for exact matching of a finite set of patterns.

It is based on the same principle as *KMP* algorithm and use the searching automaton with restricted backward transition function:

$\varphi : (Q - \{q_0\}) \rightarrow Q$.

We start with the construction of forward transition function.

Algorithm 5.9
Construction of the forward transition function of AC automaton.
Input: Finite set of patterns $S = (P_1, P_2, \ldots, P_{|S|})$, where $P_i \in A^+$, $1 \leq i \leq |S|$.
Output: Deterministic finite automaton $M = (Q, A, \delta, q_0, F)$ accepting the set S.
Method:

1. $Q = \{q_0\}$, q_0 is the initial state.
2. Create all possible states. Each new state q of AC automaton corresponds to some prefix $a_1 a_2 \ldots a_j$ of one or more patterns. Define $\delta(q, a_{j+1}) = q'$, where q' corresponds to prefix $a_1 a_2 \ldots a_j a_{j+1}$ of one or more patterns. $Q = Q \cup \{q'\}$.
3. For state q_0 define $\delta(q_0, a) = q_0$ for all such a that $\delta(q_0, a)$ was not defined in step 2.
4. $\delta(q, a) = fail$ for all q and a for which $\delta(q, a)$ was not defined in steps 2 or 3.
5. Each state corresponding to the complete pattern will be the final state. It holds also in case when one pattern is either prefix or substring of another pattern. □

The same result as by Algorithm 5.9 we can obtain by modification of *SFFECO* automaton. This modification consists in two steps:

1. Removing some number of selfloops in the initial state. There are selfloops for symbols for which exist transitions from the initial state to the next states.
2. Determinization of the automaton resulting from step 1.

Step 2 must be used only in case when some patterns in set $S = (P_1, P_2, \ldots, P_{|s|})$ have equal prefixes. Otherwise the resulting automaton is deterministic after step 1.
The next algorithm constructs the backward transition function. Let us note that the depth of state q is the number of forward transitions from state q_0 to state q.

Algorithm 5.10
Construction of backward transition function of AC automaton for set $S =$

$\{P_1, P_2, \ldots, P_{|S|}\}$.
Input: Deterministic finite automaton M accepting set S.
Output: Complete AC automaton with backward transition function φ.
Method:

1. Let $Q = Q_0 \cup Q_1 \cup \ldots \cup Q_{maxd}$, where elements of Q_d are states with depth d. Let us note:
   (a) $Q_0 = \{q_0\}$,
   (b) $Q_i \cap Q_j = \emptyset$, for $i \neq j$, $0 \leq i, j \leq maxd$.
2. $\varphi(q) = q_0$ for all $q \in Q_1$.
3. For $d = 2, \ldots, maxd$ and all states in Q_d construct *mborder* array $m\beta[2..n]$. □

The computation of backward transition function φ is depicted in Fig. 5.11.

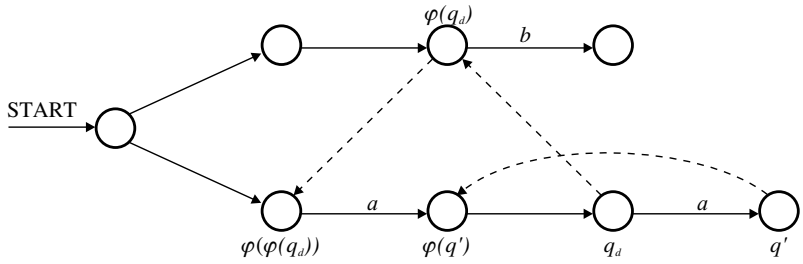


Figure 5.11: Computation of $\varphi(q')$

This backward transition function is not optimized similarly to the case of *MP* searching automaton. The optimized version φ_{opt} of function φ is constructed by the next algorithm.

Algorithm 5.11
Construction of optimized backward transition function φ_{opt} of AC searching automaton.
Input: AC searching automaton with backward transition function φ (not optimized).
Output: Optimized backward transition function φ_{opt} for the input AC searching automaton.
Method:

1. Let $Q = \{q_0\} \cup Q_1 \cup \ldots \cup Q_{maxd}$, where elements of Q_d are states with the depth d.
2. $\varphi_{opt}(q) = \varphi(q) = q_0$ for all $q \in Q_1$.
3. For $d = 2, 3, \ldots, maxd$ and all states in $q \in Q_d$ do:
   (a) $X = \{a : \delta(q, a) = p, a \in A\}$,

(b) $Y = \{a : \delta(\varphi(q), a) = r, a \in A\}$,
(c) if $Y \subseteq X$ or $X = \emptyset$ then $\varphi_{opt}(q) = \varphi_{opt}(\varphi(q))$ else $\varphi_{opt}(q) = \varphi(q)$.
□

The principle of the construction of optimized backward transition function φ_{opt} is depicted in Fig. 5.12.

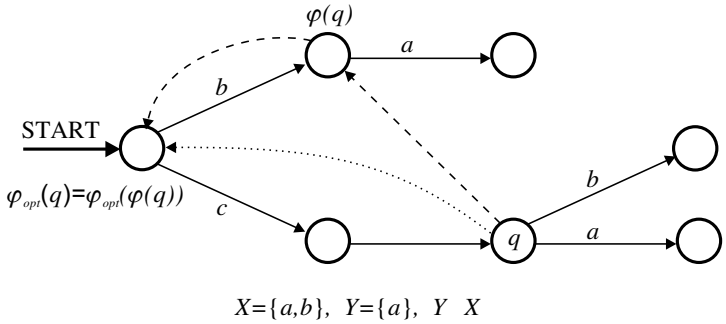


Figure 5.12: Construction of $\varphi_{opt}(q)$

Example 5.12
Let us construct AC searching automaton for set of patterns $S = \{ab, babb, bb\}$ and alphabet $A = \{a, b, c\}$. First we construct the forward transition function. It is depicted in Fig. 5.13. Further we will construct both backward

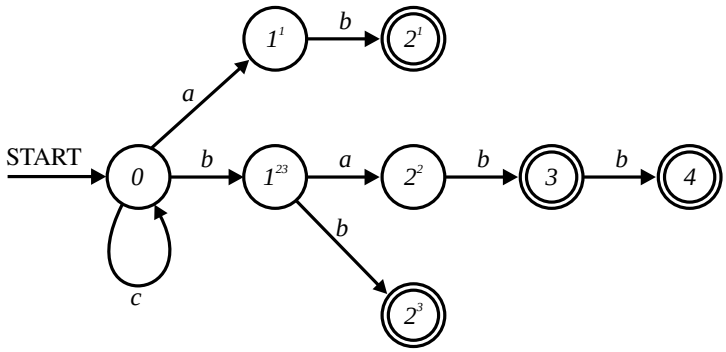


Figure 5.13: Forward transition function of AC searching automaton for set of patterns $S = \{ab, babb, bb\}$ and $A = \{a, b, c\}$ from Example 5.12

transition functions φ and φ_{opt}. To construct the nonoptimized backward transition function φ we construct the *mborder* array $m\beta$ for set of patterns $S = \{ab, babb, bb\}$. The nondeterministic factor automaton has the transition diagram depicted in Fig. 5.14. Table 5.6 is the transition table of the deterministic factor automaton for set of patterns $S = \{ab, babb, bb\}$. The transition diagram of this factor automaton is depicted in Fig. 5.15. The

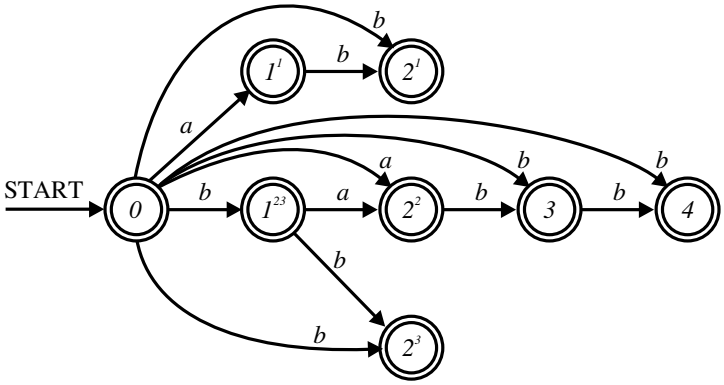


Figure 5.14: Transition diagram of the nondeterministic factor automaton for set of patterns $S = \{ab, babb, bb\}$ from Example 5.12

	a	b
0	$1^1 2^2$	$1^{23} 2^1 2^3 34$
$1^1 2^2$		$2^1 3$
$2^1 3$		4
$1^{23} 2^1 2^3 34$	2^2	$2^3 4$
2^2		3
3		4
4		
$2^3 4$		

Table 5.6: Transition table of the deterministic factor automaton for set $S = \{ab, babb, bb\}$ from Example 5.12

analysis of d–subsets of the deterministic factor automaton is summarized in the Table 5.7. The next table shows the border array.

State	1^1	1^{23}	2^1	2^2	2^3	3	4
Symbol	a	b	b	a	b	b	b
$m\beta$	0	0	1^{23}	1^1	1^{23}	2^1	2^3

The values of $\beta[1^1]$ and $\beta[1^{23}]$ are equal to zero because strings a and b have borders of zero length. This border array is equal to backward transition function φ. Optimized backward transition function φ_{opt} differs for some states of function φ. Forward transition function δ and both functions φ and φ_{opt} have values according to the Table 5.8. The complete AC searching automaton is depicted on Fig. 5.16 The backward transition function φ is shown by dashed lines, the optimized backward transition function is shown by dotted lines in cases when $\varphi \neq \varphi_{opt}$. The Fig. 5.17 shows the transition diagram of the deterministic automaton for nondeterministic *SF*-

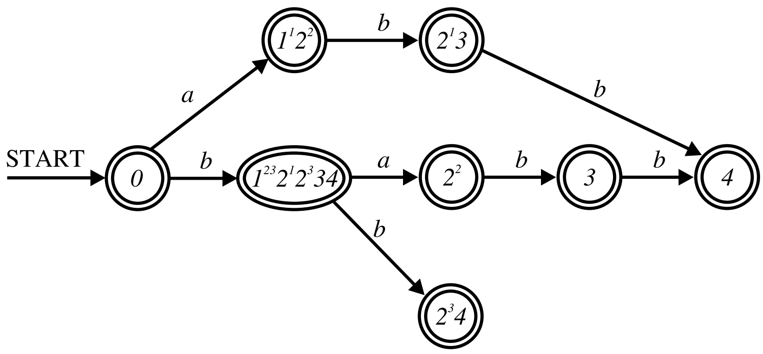


Figure 5.15: Transition diagram of the deterministic factor automaton for set of patterns $S = \{ab, babb, bb\}$ from Example 5.12

d–subset	Values of elements of border array
$1^{23}2^12^334$	$m\beta[2^1] = 1^{23}, m\beta[2^3] = 1^{23}, m\beta[3] = 1^{23}, m\beta[4] = 1^{23}$
1^12^2	$m\beta[2^2] = 1^1$
2^13	$m\beta[3] = 2^1$
2^34	$m\beta[4] = 2^3$

Table 5.7: Computation of *mborder* array for set of patterns $S = \{ab, babb, bb\}$ from Example 5.12

FECO automaton for $S = \{ab, babb, bb\}$. It is included in order to enable comparison of deterministic finite automaton and AC searching automaton.

Let us show the sequence of transitions of resulting AC searching automaton for input string *bbabb*.

$$
\begin{aligned}
(0, bbabb) &\vdash (1^{23}, babb) \\
&\vdash (2^3, \quad abb)\ bb\ \textit{found, fail} \\
&\vdash (1^{23}, \ abb) \\
&\vdash (2^2, \quad bb) \\
&\vdash (3, \quad b)\ ab\ \textit{found} \\
&\vdash (4, \quad \varepsilon)\ bb\ \textit{and}\ babb\ \textit{found}.
\end{aligned}
$$

For comparison we show the sequence of transitions of deterministic finite automaton:

$$
\begin{aligned}
(0, bbabb) &\vdash (01^{23}, \quad babb) \\
&\vdash (01^{23}2^3, \quad abb)\ bb\ \textit{found} \\
&\vdash (01^12^2, \quad bb) \\
&\vdash (01^{23}2^13, \quad b)\ ab\ \textit{found} \\
&\vdash (01^{23}2^34, \quad \varepsilon)\ bb\ \textit{and}\ babb\ \textit{found}.
\end{aligned}
$$
□

δ	a	b	c	φ	φ_{opt}
0	1^1	1^{23}	0		
1^1	$fail$	2^1	$fail$	0	0
1^{23}	2^2	2^3	$fail$	0	0
2^1	$fail$	$fail$	$fail$	1^{23}	1^{23}
2^2	$fail$	3	$fail$	1^1	0
2^3	$fail$	$fail$	$fail$	1^{23}	1^{23}
3	$fail$	4	$fail$	2^1	1^{23}
4	$fail$	$fail$	$fail$	2^3	1^{23}

Table 5.8: The forward and both backward transition functions for set of patterns $S = \{ab, babb, bb\}$ from Example 5.12

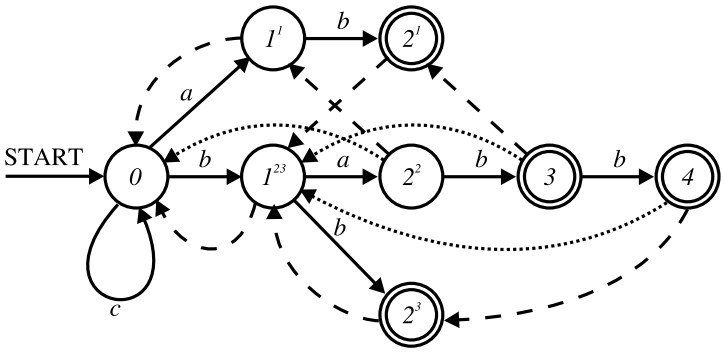


Figure 5.16: Complete AC searching automaton for set of patterns $S = \{ab, babb, bb\}$ from Example 5.12

Similarly as the KMP algorithm, the AC algorithm have linear time and space complexities. If we have alphabet A, text $T = t_1 t_2 \dots t_n$ and set of patterns $S = \{P_1, P_2, \dots P_{|s|}\}$, where $|S| = \sum_{i=1}^{|s|} |P_i|$, then AC algorithm needs [CH97b]:

- $\mathcal{O}(|S|)$ time for preprocessing phase (construction of AC searching automaton),
- $\mathcal{O}(n)$ time for searching phase,
- $\mathcal{O}(|S| * |A|)$ space to store AC searching automaton.

The space requirement can be reduced to $\mathcal{O}(|S|)$. In this case the searching phase has time complexity $\mathcal{O}(|n| * log|A|)$. See more details in Crochemore and Hancart [CH97b].

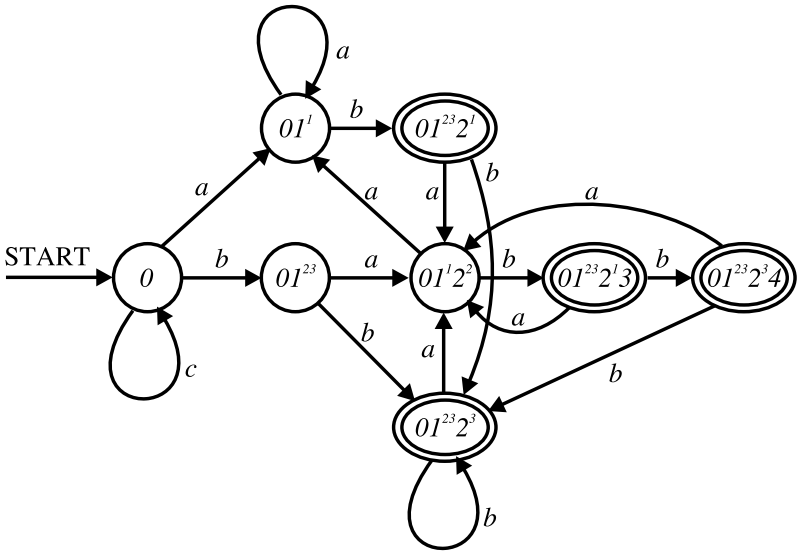


Figure 5.17: Transition diagram of the deterministic finite automaton for $S = \{ab, babb, bb\}$ from Example 5.12; transitions for symbol c from all states are leading to state 0

6 Simulation of nondeterministic finite automata – dynamic programming and bit parallelism

In the case when the space complexity or the preprocessing time of *DFA* makes it unusable, we can use some of deterministic simulation methods of corresponding *NFA*. At the beginning of this section we describe a basic simulation method, which is the base of the other simulation methods presented further in this section. This method can be used for any general *NFA*. The other simulation methods improve complexities of simulation, but on the other hand they set some requirements to *NFA*s in order to be more efficient.

The simulation methods will be presented on *NFA*s for exact and approximate string matching. In this Section we will use another version of *NFA*s for approximate string matching, where all transitions for edit operations *replace* and *insert* are labeled by whole alphabet instead of complement of matching symbol. This simplifies the formulae for the simulation while the behavior of *NFA*s practically does not change.

6.1 Basic simulation method

In Algorithm 6.1 we show the basic algorithm, which is very similar to the transformation of *NFA* to the equivalent *DFA*. In each step of the run of *NFA* a new set of active states is computed by evaluating all transitions from all states of the previous set of active states. This provides that all possible paths labeled by input string are considered in *NFA*. The simulation finishes when the end of the input text is reached or when there is no active state (e.g., no accepting path exists).

Algorithm 6.1 (Simulation of run of *NFA*—basic method)
Input: *NFA* $M = (Q, A, \delta, q_0, F)$, input text $T = t_1 t_2 \dots t_n$.
Output: Output of run of *NFA*.
Method: Set S of active states is used.

$S := \varepsilon - CLOSURE(\{q_0\})$
$i := 1$
while $i \leq n$ **and** $S \neq \emptyset$ **do**
/* transitions are performed for all elements of S */
$S := \bigcup_{q \in S} \varepsilon - CLOSURE(\delta(q, t_i))$
if $S \cap F \neq \emptyset$ **then**
write(information associated with each final state in $S \cap F$)
endif
$i := i + 1$
endwhile

In the transformation of *NFA* to the equivalent *DFA*, all the possible configurations of set S of active states are evaluated as well as all the possible transitions among these configurations and a deterministic state is assigned to each such configuration of S. Using the simulation method from Algorithm 6.1 only the current configuration of set S is evaluated in each step of the simulation of *NFA*—only used configurations are evaluated during the simulation.

It is also possible to combine the simulation and the transformation. When processing an input text, we can store the used configurations of S in some state-cache and assign them deterministic states. In such a way we transform *NFA* to *DFA* incrementally, but we evaluate only used states and transitions. If the state-cache is full, we can use one of cache techniques for making room in the state-cache, e.g., removing least recently used states.

This solution has the following advantages: we can control the amount of used memory (size of state-cache) and for the most frequent configurations we do not need always to compute the most frequent transitions, but we can use directly the information stored in the state-cache (using a *DFA* transition has better time complexity than computing a new configuration).

Theorem 6.2
The basic simulation method shown in Algorithm 6.1 simulates run of *NFA*.

Proof
Let $M = (Q, A, \delta, q_0, F)$ be an *NFA* and $T = t_1 t_2 \dots t_n$ be an input text. Algorithm 6.1 considers really all paths (i.e., sequences of configurations) leading from q_0.

At the beginning of the algorithm, set S of active states contains ε–$CLOSURE(\{q_0\})$—each path must start in q_0 and then some of ε-transitions leading from q_0 can be used. In this way all configurations (q_j, T), $q_j \in Q$, $(q_0, T) \vdash_M^* (q_j, T)$, are considered.

In each i-th step of the algorithm, $1 \leq i \leq n$, all transitions (relevant to T) leading from all states of S are evaluated, i.e., both t_i labeled transitions as well as ε-transitions. At first all transitions reachable by transitions labeled by t_i from each state of S are inserted in new set S and then $\varepsilon - CLOSURE$ of new set S is also inserted in new set S. In this way all configurations $(q_j, t_{i+1} t_{i+2} \dots t_n)$, $q_j \in Q$, $(q_0, T) \vdash_M^* (q_j, t_{i+1} t_{i+2} \dots t_n)$, are considered in i-th step of the algorithm.

In each step of the algorithm, the set S of active states is tested for final state and for each such found active final state the information associated with such state is reported. □

6.1.1 Implementation

6.1.1.1 *NFA* without ε-transitions This basic method can be implemented using bit vectors. Let $M = (Q, A, \delta, q_0, F)$ be an *NFA* without

ε-transitions and $T = t_1 t_2 \ldots t_n$ be an input text of M. We implement a transition function δ as a transition table $\mathcal{T}$ (of size $|Q| \times |A|$) of bit vectors:

$$\mathcal{T}[i,a] = \begin{bmatrix} \tau_0 \\ \tau_1 \\ \vdots \\ \tau_{|Q|-1} \end{bmatrix} \tag{5}$$

where $a \in A$ and bit $\tau_j = 1$, if $q_j \in \delta(q_i, a)$, or $\tau_j = 0$ otherwise. Then we also implement a set F of final states as a bit vector $\mathcal{F}$:

$$\mathcal{F} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{|Q|-1} \end{bmatrix} \tag{6}$$

where bit $f_j = 1$, if $q_j \in F$, or $f_j = 0$ otherwise. In each step i, $0 \leq i \leq n$, (i.e., after reading symbol t_i) of the simulation of the *NFA* run, set S of active states is represented by bit vector $\mathcal{S}$:

$$\mathcal{S}_i = \begin{bmatrix} s_{0,i} \\ s_{1,i} \\ \vdots \\ s_{|Q|-1,i} \end{bmatrix} \tag{7}$$

where bit $s_{j,i} = 1$, if state q_j is active (i.e. $q_j \in \mathcal{S}$) in i-th simulation step, or $s_{j,i} = 0$ otherwise.

When constructing $\mathcal{S}_{i+1}$, we can evaluate all transitions labeled by t_{i+1} leading from state q_j, which is active in i-th step, at once—just using bitwise operation *or* for bit-vector $\mathcal{S}_{i+1}$ and bit-vector $\mathcal{T}[j, t_{i+1}]$. This implementation is used in Algorithm 6.3.

Note, that the first **for** cycle of Algorithm 6.3 is multiplication of vector $\mathcal{S}_i$ by matrix $\mathcal{T}[*, t_i]$. This approach is also used in quantum automata [MC97], where quantum computation is used.

Theorem 6.4
The simulation of the run of general *NFA* runs in time $\mathcal{O}(n|Q|\lceil\frac{|Q|}{w}\rceil)$ and space[1] $\mathcal{O}(|A||Q|\lceil\frac{|Q|}{w}\rceil)$, where n is the length of the input text, $|Q|$ is the

[1]For the space complexity of this theorem we expect complete transition table for representation of transition function.

Algorithm 6.3 (Simulation of run of *NFA*—bit-vector implementation of basic method)
Input: Transition table $\mathcal{T}$ and set $\mathcal{F}$ of final states of *NFA*, input text $T = t_1 t_2 \dots t_n$.
Output: Output of run of *NFA*.
Method:

```
S_0 := [100...0]                    /* only q_0 is active at the beginning */
i := 1
while i ≤ n and S_{i-1} ≠ [00...0] do
  S_i := [00...0]
  for j := 0, 1, ... |Q| - 1 do
    if s_{j,i-1} = 1 then           /* q_j is active in (i-1)-th step */
      S_i := S_i OR T[j, t_i]       /* evaluate transitions for q_j */
    endif
  endfor
  for j := 0, 1, ... |Q| - 1 do
    if s_{j,i} = 1 and f_j = 1 then /* if q_j is active final state */
      write(information associated with final state q_j)
    endif
  endfor
  i := i + 1
endwhile
```

number of states of the *NFA*, A is the input alphabet, and w is the size of the used computer word in bits.

Proof
See the basic simulation method in Algorithm 6.3. The main while-cycle is performed at most n-times and both inner while-cycles are performed just $|Q|$ times. If $|Q| > w$ (i.e., more than one computer word must be used to implement bit-vector for all states of *NFA*), each elementary bitwise operations must be split into $\lceil \frac{|Q|}{w} \rceil$ bitwise operations. It gives us time complexity $\mathcal{O}(n|Q|\lceil \frac{|Q|}{w} \rceil)$. The space complexity is given by the implementation of transition function δ by transition table $\mathcal{T}$, which contains $(|Q| \times |A|)$ bit-vectors each of size $\lceil \frac{|Q|}{w} \rceil$. □

Example 6.5
Let M be an *NFA* for the exact string matching for pattern $P = aba$ and $T = accabcaaba$ be an input text.

Transition table δ and its bit-vector representation $\mathcal{T}$ are as follows:

δ_M	a	b	$A \setminus \{a, b\}$
0	{0,1}	{0}	{0}
1	$\emptyset$	{2}	$\emptyset$
2	{3}	$\emptyset$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

$\mathcal{T}_M$	a	b	$A \setminus \{a, b\}$
0	1	1	1
	1	0	0
	0	0	0
	0	0	0
1	0	0	0
	0	0	0
	0	1	0
	0	0	0
2	0	0	0
	0	0	0
	0	0	0
	1	0	0
3	0	0	0
	0	0	0
	0	0	0
	0	0	0

The process of simulation of M over T is displayed by set S of active states and its bit-vector representation $\mathcal{S}$.

	-	a	c	c	a	b	c	a	a	b	a
$\mathcal{S}$	1	1	1	1	1	1	1	1	1	1	1
	0	1	0	0	1	0	0	1	1	0	1
	0	0	0	0	0	1	0	0	0	1	0
	0	0	0	0	0	0	0	0	0	0	1
S	q_0	q_0 q_1	q_0	q_0	q_0 q_1	q_0 q_2	q_0	q_0 q_1	q_0 q_1	q_0 q_2	q_0 q_1 q_3

6.1.1.2 *NFA* with ε-transitions If we have an *NFA* with ε-transitions, we can transform it to an equivalent *NFA* without ε-transitions using Algorithm 6.6. There are also other algorithms for removing ε-transitions but this algorithm does not change the states of the *NFA*.

Algorithm 6.6 provides that all transitions labeled by symbols (not ε-transitions) leading from all states of $\varepsilon - CLOSURE(\{q_0\})$ lead also from q_0. Then for each state $q \in Q$ and symbol $a \in A$ all states accessible from $\delta(q, a)$ (i.e., $\varepsilon - CLOSURE(\delta(q, a))$) are inserted into $\delta'(q, a)$.

Note, that the resulting *NFA* M' can contain inaccessible states. Each state q, $q \in S \setminus \{q_0\}$, of *NFA* M' is inaccessible if the only incoming transitions into this state q in *NFA* M are the ε-transitions leading from q_0.

The ε-transitions can also be removed directly during a construction of *NFA*.

There is also other possibility of simulation of *NFA* with ε-transitions. We can implement ε-transitions by table $\mathcal{E}$ (of size $|Q|$) of bit-vectors:

Algorithm 6.6 (Removing ε-transitions from *NFA*)
Input: *NFA* $M = (Q, A, \delta, q_0, F)$ with ε-transitions.
Output: *NFA* $M' = (Q, A, \delta', q_0, F')$ without ε-transitions.
Method:

$S := \varepsilon - CLOSURE(\{q_0\})$
for each $a \in A$ **do**
  $\delta'(q_0, a) := \bigcup_{q \in S} \varepsilon - CLOSURE(\delta(q, a))$
endfor
for each $q \in Q \setminus \{q_0\}$ **do**
  for each $a \in A$ **do**
    $\delta'(q, a) := \varepsilon - CLOSURE(\delta(q, a))$
  endfor
endfor
if $S \cap F \neq \emptyset$ **then**
  $F' := F \cup \{q_0\}$
else
  $F' := F$
endif

$$\mathcal{E}[i] = \begin{bmatrix} e_0 \\ e_1 \\ \vdots \\ e_{|Q|-1} \end{bmatrix} \tag{8}$$

where bit $e_j = 1$, if $q_j \in \varepsilon - CLOSURE(\{q_i\})$, or $e_j = 0$ otherwise.

This implementation is used in Algorithm 6.7, where $\varepsilon - CLOSURE(S)$ is computed in each step of the simulation. The time and space complexities are asymptotically same as in Algorithm 6.3, therefore Theorem 6.4 holds for all *NFA*s (with as well as without ε-transitions).

Lemma 6.8
Let *NFA* $M = (Q, A, \delta, q_0, F)$ is implemented by bit-vectors as shown in the previous subsection. Then Algorithm 6.6 runs in time $\mathcal{O}(|A||Q|^2\lceil\frac{|Q|}{w}\rceil)$ and space $\mathcal{O}(|A||Q|\lceil\frac{|Q|}{w}\rceil)$, where w is a length of used computer word in bits.

Proof
Let ε-transitions be implemented by table $\mathcal{E}$ as shown above (See Formula 8).

The statement in the first **for** cycle of Algorithm 6.6 contains the cycle performed for all states of S ($\mathcal{O}(|Q|)$), in which there is nested another **for** cycle performed for all states of $\delta(q, a)$ ($\mathcal{O}(|Q|)$).

Algorithm 6.7 (Simulation of run of *NFA* with ε-transitions—bit-vector implementation of basic method)
Input: Transition tables $\mathcal{T}$ and $\mathcal{E}$, and set $\mathcal{F}$ of final states of *NFA*, input text $T = t_1 t_2 \dots t_n$.
Output: Output of run of *NFA*.
Method:

```
                    /* only ε−CLOSURE({q_0}) is active at the beginning */
S_0 := [100...0] OR E[0]
i := 1
while i ≤ n and S_{i−1} ≠ [00...0] do
  S_i := [00...0]
  for j := 0, 1, ... |Q| − 1 do
    if s_{j,i−1} = 1 then              /* q_j is active in (i − 1)-th step */
      S_i := S_i OR T[j, t_i]            /* evaluate transitions for q_j */
    endif
  endfor
  for j := 0, 1, ... |Q| − 1 do      /* construct ε−CLOSURE(S_i) */
    if s_{j,i} = 1 then
      S_i := S_i OR E[j]
    endif
  endfor
  for j := 0, 1, ... |Q| − 1 do
    if s_{j,i} = 1 and f_j = 1 then      /* if q_j is active final state */
      write(information associated with final state q_j)
    endif
  endfor
  i := i + 1
endwhile
```

The statement inside next two nested **for** cycles ($\mathcal{O}(|Q|)$ and $\mathcal{O}(A)$) also contains a cycle ($\varepsilon-CLOSURE()$ for all states of $\delta(q, a)$—$\mathcal{O}(|Q|)$).

Therefore the total time complexity is $\mathcal{O}(|A||Q|^2\lceil\frac{|Q|}{w}\rceil)$. The space complexity is given by the size of new copy M' of *NFA* and no other space is needed. □

The basic simulation method presented in this section can be used for any *NFA* and it runs in time $\mathcal{O}(n|Q|\lceil\frac{|Q|}{w}\rceil)$ and space $\mathcal{O}(|A||Q|\lceil\frac{|Q|}{w}\rceil)$, where w is a length of used computer word in bits. The other simulation methods shown below attempt to improve the time and space complexity, but they cannot be used for general *NFA*.

6.2 Dynamic programming

The dynamic programming is a general technique widely used in various branches of computer science. It was also utilized in the approximate string matching using the Hamming distance [Mel95] (and [GL89] for detection *of all permutations of pattern* with at most k errors in text) and in the approximate string matching using the Levenshtein distance [WF74, Sel80, Ukk85, LV88, Mel95].

In this Section we describe how the dynamic programming simulates the *NFA*s for the approximate string matching using the Hamming and Levenshtein distances. In the dynamic programming, the set of active states is represented by a vector of integer variables.

6.2.1 Algorithm

The dynamic programming for the string and sequence matching computes matrix D of size $(m+1) \times (n+1)$. Each element $d_{j,i}$, $0 \leq j \leq m$, $0 < i \leq n$, usually contains the edit distance between the string ending at i-th position in text T and the prefix of pattern P of length j.

6.2.2 String matching

6.2.2.1 Exact string matching The dynamic programming for the exact string matching is the same as the dynamic programming for the approximate string matching using the Hamming distance in which $k = 0$. See the paragraph below.

6.2.2.2 Approximate string matching using Hamming distance In the approximate string matching using the Hamming distance each element $d_{j,i}$, $0 < i \leq n$, $0 \leq j \leq m$, contains the Hamming distance between string $t_{i-j+1} \ldots t_i$ and string $p_1 \ldots p_j$. Elements of matrix D are computed as follows:

$$\begin{array}{lcll} d_{j,0} & := & j, & 0 \leq j \leq m \\ d_{0,i} & := & 0, & 0 \leq i \leq n \\ d_{j,i} & := & \textbf{min}(\textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1}, & \\ & & \quad d_{j-1,i-1} + 1), & 0 < i \leq n, \\ & & & 0 < j \leq m \end{array} \tag{9}$$

Formula 9 exactly represents simulation of *NFA* for approximate string matching where transition *replace* is labeled by all symbols of alphabet as shown in Fig. 6.1. However, we can optimize a little bit the formula and we get Formula 10.

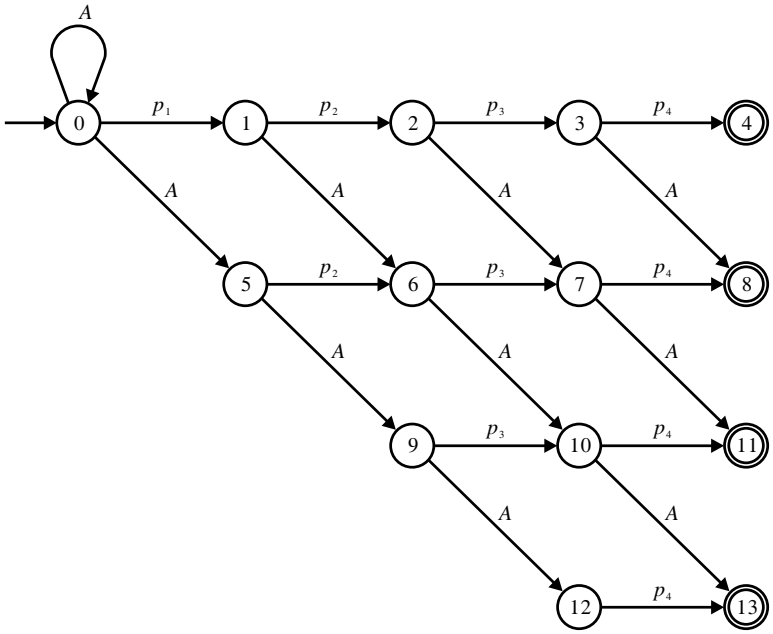


Figure 6.1: *NFA* for the approximate string matching using the Hamming distance ($m = 4$, $k = 3$)

$$\begin{array}{lcll} d_{j,0} & := & k+1, & 0 < j \leq m \\ d_{0,i} & := & 0, & 0 \leq i \leq n \\ d_{j,i} & := & \textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} \textbf{ else } d_{j-1,i-1} + 1, & 0 < i \leq n, \\ & & & 0 < j \leq m \end{array} \tag{10}$$

In Formula 10 term $d_{j-1,i-1}$ represents matching—position i in text T is increased, position j in pattern P is increased and edit distance d is the same. Term $d_{j-1,i-1} + 1$ represents edit operation *replace*—position i in text T is increased, position j in pattern P is increased and edit distance d is increased. The value of $d_{0,i}$, $0 \leq i \leq n$, is set to 0, because the Hamming distance between two empty strings (the prefix of length 0 of the pattern and string of length 0 ending at position i) is 0. The value of $d_{j,0}$, $0 \leq j \leq m$, is set to $k + 1$, where k is the maximum number of allowed errors (maximum Hamming distance). In such a way it holds $d_{j,i} > k$, $\forall i, j$, $0 \leq i < m$, $i < j \leq m$, so all the items not satisfying condition $j \leq i$ exceed maximum acceptable value k.

Algorithm 6.9 shows the use of matrix D in the approximate string matching.

An example of matrix D using Formula 10 for searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ with $k = 3$ is shown in Table 6.1.

Since we are interested in at most k errors in the found string, each value

Algorithm 6.9 (Simulation of run of *NFA*—dynamic programming)
Input: Pattern $P = p_1p_2 \dots p_m$, input text $T = t_1t_2 \dots t_n$, maximum number of errors allowed k, $k < m$.
Output: Output of run of *NFA*.
Method:

```
Compute 0-th column of matrix D
for i := 1, 2, ..., n do
  Compute i-th column of matrix D for input symbol t_i
  if d_{m,i} ≤ k then
    write('Pattern P with d_{m,i} errors ends at position i in text T.')
  endif
endfor
```

D	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	4	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
d	4	5	0	2	2	1	2	2	1	1	2	0	2	2	2	2
b	4	5	6	1	3	2	2	3	3	1	2	3	0	2	3	3
b	4	5	6	7	2	3	3	3	4	3	2	3	3	0	3	4
c	4	5	6	6	8	3	3	4	4	5	4	3	4	4	0	4
a	4	4	6	7	6	9	4	3	4	5	5	5	4	5	5	0

Table 6.1: Matrix D for pattern $P = adbbca$, text $T = adcabcaabadbbca$, and $k = 3$ using the Hamming distance

of $d_{i,j}$ greater than $k+1$ can be replaced by value $k+1$ and represents that the value is greater than k. It is useful in some implementations, since we need just $\lceil \log_2(k+2) \rceil$ bits for each number.

Theorem 6.10
The dynamic programming algorithm described by Formula 10 simulates a run of the *NFA* for the approximate string matching using the Hamming distance.

Proof
In the dynamic programming for the approximate string matching using the Hamming distance there is for each depth j, $0 < j \leq m$, of *NFA* in each step i of the run one integer variable $d_{j,i}$ that contains the Hamming distance between string $p_1 \dots p_j$ and the string in text T ending at position i. Since for each value l of the Hamming distance there is one level of states in *NFA*, integer variable $d_{j,i} = l$ and it contains level number of the topmost active state in j-th depth of *NFA*. Each value of $d_{i,j}$ greater than $k+1$ can be

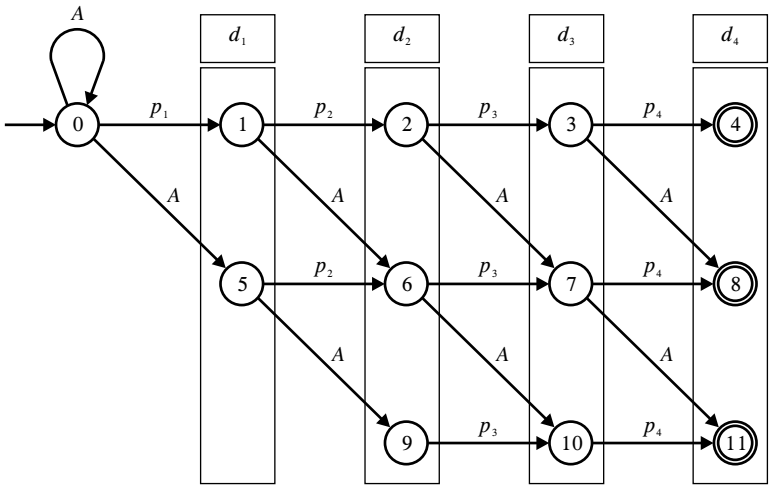


Figure 6.2: Dynamic programming uses for each depth of states of *NFA* one integer variable d

replaced by value $k + 1$ and it represents that there is no active state in j-th depth of *NFA* in i-th step of the run. So in dynamic programming, the set of active states from Section 6.1 is implemented by the vector of integer variables—each variable for one depth of *NFA*.

In Formula 10 term $d_{j-1,i-1}$ represents matching transition—active state is moved from depth $j-1$ to the next depth j within level $d_{j-1,i-1}$ and symbol t_i is read from the input. Term $d_{j-1,i-1}+1$ represents transition *replace*—active state is moved from depth $j-1$ and level $d_{j-1,i-1}$ to the next depth j and the next level $d_{j-1,i-1}+1$ and symbol t_i is read from the input.

If the condition in **if** statement of Formula 10 holds (i.e., p_j is read), only one value ($d_{j-1,i-1}$) is considered.

The selfloop of the initial state is represented by setting $d_{0,i} := 0$, $0 \leq i \leq n$.

Therefore all transitions (paths) of the *NFA* are considered.

At the beginning, only the initial state is active, therefore $d_{0,0} = 0$ and $d_{j,0} = k+1$, $0 < j \leq m$.

If $d_{m,i} \leq k$, then we report that pattern P was found with $d_{m,i}$ errors (the final state of level $d_{m,i}$ is active) ending at position i. □

6.2.2.3 Approximate string matching using Levenshtein distance

In the approximate string matching using the Levenshtein distance [Sel80, Ukk85] each element $d_{j,i}$, $0 < i \leq n$, $0 \leq j \leq m$, contains the Levenshtein distance between the string ending at position i in T and string $p_1 \ldots p_j$. Since the Levenshtein distance compares two strings of not necessary equal

length, element $d_{j,i}$ is always valid (symbols can be deleted from the pattern). It also implies that we can directly determine only the ending position of the found string in text.

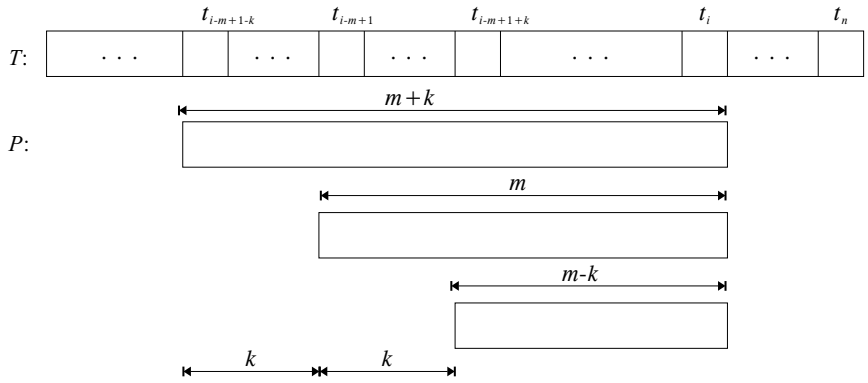


Figure 6.3: Range of beginnings of occurrence of P ending at position i in T

If the found string ends at position i, it can start in front of position $i - m + 1$ (at most k symbols inserted) or behind position $i - m + 1$ (at most k symbols deleted)—the beginning of occurrence of pattern P can be located at position $i - m + 1 + l$, $-k \leq l \leq k$, as shown in Figure 6.3.

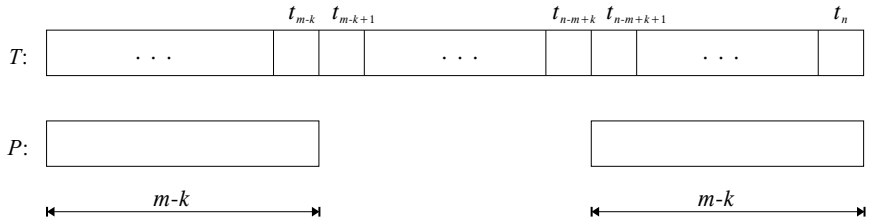


Figure 6.4: The first and the last possible occurrence of P in T

It also implies that the first possible occurrence of the pattern in the text can end at position $m-k$ and the last occurrence can start at $n-m+k+1$ as shown in Figure 6.4.

Elements of matrix D are computed as follows:

$$
\begin{array}{lcll}
d_{j,0} & := & j, & 0 \leq j \leq m \\
d_{0,i} & := & 0, & 0 \leq i \leq n \\
d_{j,i} & := & \textbf{min}(\textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} & \\
 & & \quad \textbf{else } d_{j-1,i-1} + 1, & \\
 & & \quad \textbf{if } j < m \textbf{ then } d_{j,i-1} + 1, & \\
 & & \quad d_{j-1,i} + 1), & 0 < i \leq n, \\
 & & & 0 < j \leq m
\end{array}
\tag{11}
$$

Formula 11 exactly represents simulation of *NFA* for approximate string matching where transitions *replace* and *insert* are labeled by all symbols of alphabet as shown in Fig. 6.5.

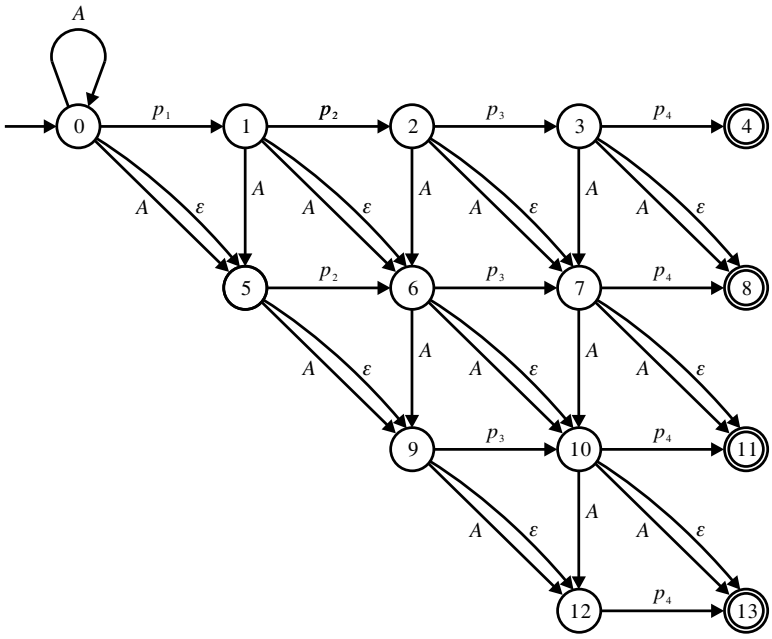


Figure 6.5: *NFA* for the approximate string matching using the Levenshtein distance ($m = 4$, $k = 3$)

In Formula 11 term $d_{j-1,i-1}$ represents matching and term $d_{j-1,i-1} + 1$ represents edit operation *replace*. Term $d_{j,i-1} + 1$ represents edit operation *insert*—position i in text T is increased, position j in pattern P is not increased and edit distance d is increased. Term $d_{j-1,i} + 1$ represents edit operation *delete*—position i in text T is not increased, position j in pattern P is increased and edit distance d is increased.

Pattern P is found with at most k differences ending at position i if $d_{m,i} \leq k$, $1 \leq i \leq n$. The maximum number of differences of the found string is $D_L(P, t_{i-l+1} \ldots t_i) = d_{m,i}$, $m - k \leq l \leq m + k$, $l < i$.

An example of matrix D for searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ is shown in Table 6.2.

Theorem 6.11
The dynamic programming algorithm described by Formula 11 simulates a run of the *NFA* for the approximate string matching using the Levenshtein distance.

Proof
The proof is similar to the proof of Theorem 6.10. We only have to show the simulation of edit operations *insert* and *delete*.

D	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
d	2	1	0	1	1	1	2	1	1	1	1	0	1	2	2	1
b	3	2	1	1	2	1	2	2	2	1	2	1	0	1	2	2
b	4	3	2	2	2	2	2	3	3	2	2	2	1	0	1	2
c	5	4	3	2	3	3	2	3	4	3	3	3	2	1	0	1
a	6	5	4	3	2	4	3	2	3	4	3	4	3	2	1	0

Table 6.2: Matrix D for pattern $P = adbbca$ and text $T = adcabcaabadbbca$ using the Levenshtein distance

In Formula 11 term $d_{j-1,i-1}$ represents matching transition and term $d_{j-1,i-1}+1$ represents transition *replace*. Term $d_{j,i-1}+1$ represents transition *insert*—active state is moved from level $d_{j,i-1}$ to the next level $d_{j,i-1}+1$ within depth j and symbol t_i is read from the input. Term **if** $j < m$ **then** provides that *insert* transition is not considered in depth m, where the *NFA* has no *insert* transition. Term $d_{j-1,i}+1$ represents transition *delete*—active state is moved from depth $j-1$ and level $d_{j-1,i}$ to the next depth j and the next level $d_{j-1,i}+1$ and no symbol is read from the input.

Since *replace* and *insert* transitions are labeled by all symbols of input alphabet A, values $d_{j-1,i-1}+1$ and $d_{j,i-1}+1$ are considered even if $t_i = p_j$. The contribution of the matching transition (i.e., $d_{j-1,i-1}$) is considered only if $t_i = p_j$.

Thus all transitions of the *NFA* are considered. □

In [GP89] they compress the matrix D. They use the property that $(d_{j,i} - d_{j-1,i-1}) \in \{0, 1\}$—the number of errors of an occurrence can only be nondecreasing when reading symbols of that occurrence. Using this property they shorten each column of the matrix D to $k+1$ entries. They represent the matrix D diagonal by diagonal in such a way that each line j, $0 \leq j \leq k$, of that new matrix contains the number of last entry of diagonal of matrix D containing j errors.

6.2.2.4 Approximate string matching using generalized Levenshtein distance For the approximate string matching using the generalized Levenshtein distance we modify Formula 11 for the Levenshtein distance such that we have added the term for edit transition *transpose*. The resulting formula is as follows:

$$
\begin{array}{lcll}
d_{j,0} & := & j, & 0 \le j \le m \\
d_{0,i} & := & 0, & 0 \le i \le n \\
d_{j,i} & := & \textbf{min}(\textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} & \\
 & & \quad \textbf{else } d_{j-1,i-1} + 1, & \\
 & & \quad \textbf{if } j < m \textbf{ then } d_{j,i-1} + 1, & \\
 & & \quad d_{j-1,i} + 1, & \\
 & & \quad \textbf{if } i > 1 \textbf{ and } j > 1 & \\
 & & \quad\quad \textbf{and } t_{i-1} = p_j \textbf{ and } t_i = p_{j-1} & \\
 & & \quad \textbf{then } d_{j-2,i-2} + 1), & 0 < i \le n, \\
 & & & 0 < j \le m
\end{array}
\tag{12}
$$

Formula 12 exactly represents simulation of *NFA* for approximate string matching shown in Fig. 6.6.

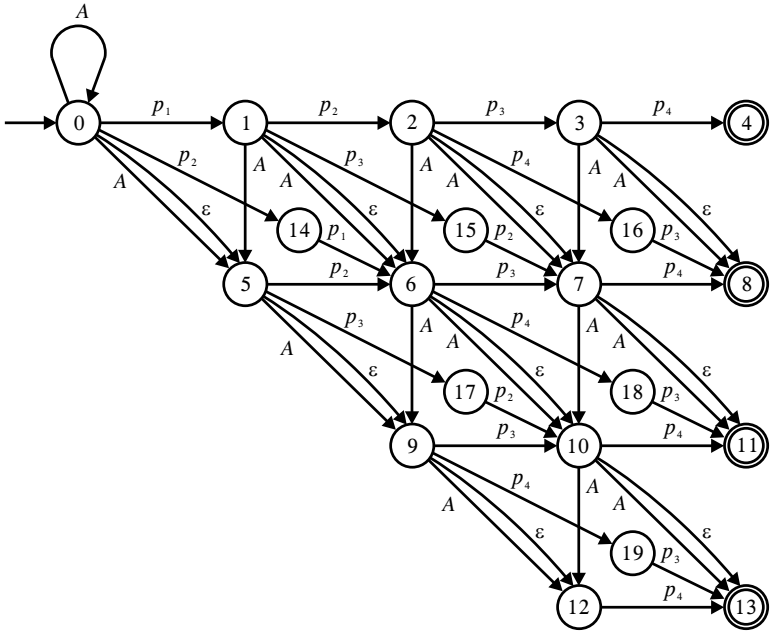


Figure 6.6: *NFA* for the approximate string matching using the generalized Levenshtein distance ($m = 4$, $k = 3$)

In Formula 12 term $d_{j-1,i-1}$ represents matching, term $d_{j-1,i-1}+1$ represents edit operation *replace*, term $d_{j,i-1}+1$ represents edit operation *insert*, term $d_{j-1,i}+1$ represents edit operation *delete*, and term $d_{j-2,i-2}+1$ represents edit operation *transpose*—position i in text T is increased by 2, position j in pattern P is increased by 2 and edit distance d is increased by 1.

An example of matrix D for searching for pattern $P = adbbca$ in text $T = adbcbaabadbbca$ is shown in Table 6.3.

D	-	a	d	b	c	b	a	a	b	a	d	b	b	c	a
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	1	1	0	0	1	0	1	1	1	1	0
d	2	1	0	1	2	2	1	1	1	1	0	1	2	2	1
b	3	2	1	0	1	2	2	2	1	2	1	0	1	2	2
b	4	3	2	1	1	1	2	3	2	2	2	1	0	1	2
c	5	4	3	2	1	1	2	3	3	3	3	2	1	0	1
a	6	5	4	3	2	2	1	2	4	3	4	3	2	1	0

Table 6.3: Matrix D for pattern $P = adbbca$ and text $T = adbcbaabadbbca$ using the generalized Levenshtein distance

Theorem 6.12
The dynamic programming algorithm described by Formula 12 simulates a run of the *NFA* for the approximate string matching using the generalized Levenshtein distance.

Proof
The proof is similar to the proof of Theorem 6.11. We only have to show the simulation of edit operation *transpose*.

In Formula 12 term $d_{j-1,i-1}$ represents matching transition, term $d_{j-1,i-1} + 1$ represents transition *replace*, term $d_{j,i-1} + 1$ represents transition *insert*, and term $d_{j-1,i} + 1$ represents transition *delete*.

Term $d_{j-2,i-2} + 1$ represents transition *transpose*—active state is moved from depth $j - 2$ and level $d_{j-2,i-2}$ to depth j and level $d_{j-2,i-2} + 1$ and symbols t_{i-1} and t_i are read from the input. When representing transition *transpose* we need not new integer variable for state on transition *transpose* in *NFA*. □

6.2.3 Time and space complexity

In the algorithms presented in this subsection, matrix D of size mostly $(m + 1) \times (n + 1)$ is computed, but in practice one needs just only one (or two for the generalized Levenshtein distance) previous columns d_{i-1} (or d_{i-1}, d_{i-2} respectively) in order to compute column d_i. In the columns we do not need to store 0-th item, since this item contains always the same value (except step 0). Therefore the space complexity of this matrix is $\mathcal{O}(m)$.

All operations shown in formulae for computing each element of matrix D can be performed in constant time, therefore the time complexity of the simulation of run of *NFA*s for the approximate string and sequence matching is $\mathcal{O}(mn)$.

In [GP89] they compacted the matrix D for the approximate string matching using the Levenshtein distance exploiting the property that $d_{j,i} -$

$d_{j-1,i-1} \in \{0,1\}$, $0 < i \leq n$, $0 < j \leq m$. For each diagonal of the matrix D they store only the positions, in which the value increases.

Let us remark that the size of the input alphabet can be reduced to reduced alphabet A', $A' \subseteq A$, $|A'| \leq m+1$ (A' contains all the symbols used in pattern P and one special symbol for all the symbols not contained in P). So A is bounded by a length m of pattern P.

The simulation of *NFA* using dynamic programming has time complexity $\mathcal{O}(mn)$ and space complexity $\mathcal{O}(m)$.

6.3 Bit parallelism

Bit parallelism is a method that uses bit vectors and benefits from the feature that the same bitwise operations (OR, AND, ADD, ... etc.) over groups of bits (or over individual bits) can be performed at once in parallel over the whole bit vector. The representatives of bit parallelism are Shift-Or, Shift-And, and Shift-Add algorithms.

Bit parallelism was used for the exact string matching (Shift-And in [Döm64]), the multiple exact string matching (Shift-And in [Shy76]), the approximate string matching using the Hamming distance (Shift-Add in [BYG92]), the approximate string matching using the Levenshtein distance (Shift-Or in [BYG92] and Shift-And in [WM92]) and for the generalized pattern matching (Shift-Or in [Abr87]), where the pattern consists not only of symbols but also of sets of symbols.

In this Section we will discuss only Shift-Or and Shift-Add algorithms. Shift-And algorithm is the same as Shift-Or algorithm but the meaning of 0 and 1 is exchanged as well as the use of bitwise operations AND and OR is exchanged.

6.3.1 Algorithm

The Shift-Or algorithm uses matrices R^l, $0 \leq l \leq k$ of size $m \times (n+1)$ and mask matrix D of size $m \times |A|$. Each element $r^l_{j,i}$, $0 < j \leq m$, $0 \leq i \leq n$, contains 0, if the edit distance between string $p_1 \ldots p_j$ and string ending at position i in text T is $\leq l$, or 1, otherwise. Each element $d_{j,x}$, $0 < j \leq m$, $x \in A$, contains 0, if $p_j = x$, or 1, otherwise. The matrices are implemented as tables of bit vectors as follows:

$$R^l_i = \begin{bmatrix} r^l_{1,i} \\ r^l_{2,i} \\ \vdots \\ r^l_{m,i} \end{bmatrix} \text{ and } D[x] = \begin{bmatrix} d_{1,x} \\ d_{2,x} \\ \vdots \\ d_{m,x} \end{bmatrix}, 0 \leq i \leq n, 0 \leq l \leq k, x \in A. \quad (13)$$

6.3.2 String matching

6.3.2.1 Exact string matching In the exact string matching, vectors R_i^0, $0 \leq i \leq n$, are computed as follows [BYG92]:

$$\begin{array}{lcll} r_{j,0}^0 & := & 1, & 0 < j \leq m \\ R_i^0 & := & \mathbf{shl}(R_{i-1}^0) \text{ OR } D[t_i], & 0 < i \leq n \end{array} \tag{14}$$

In Formula 14 operation **shl**() is the bitwise operation `left shift` that inserts 0 at the beginning of vector and operation OR is the bitwise operation `or`. Term $\mathbf{shl}(R_{i-1}^0)$ OR $D[t_i]$ represents matching—position i in text T is increased, position in pattern P is increased by operation **shl**(), and the positions corresponding to the input symbol t_i are selected by term OR $D[t_i]$. Pattern P is found at position $t_{i-m+1} \ldots t_i$ if $r_{m,i}^0 = 0$, $0 < i \leq n$.

An example of mask matrix D for pattern $P = adbbca$ is shown in Table 6.4 and an example of matrix R^0 for exact searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ is shown in Table 6.5.

D	a	b	c	d	$A \setminus \{a,b,c,d\}$
a	0	1	1	1	1
d	1	1	1	0	1
b	1	0	1	1	1
b	1	0	1	1	1
c	1	1	0	1	1
a	0	1	1	1	1

Table 6.4: Matrix D for pattern $P = adbbca$

R^0	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
d	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**

Table 6.5: Matrix R^0 for the exact string matching (for pattern $P = adbbca$ and text $T = adcabcaabadbbca$)

Theorem 6.13
Shift-Or algorithm described by Formula 14 simulates a run of the *NFA* for the exact string matching.

Proof
In Shift-Or algorithm for the exact string matching there is one bit vector R_i^0, $0 \leq i \leq n$, which represents the set of active states of *NFA*. In the vector, 0 represents active state and 1 represents non-active state of the simulated *NFA*. So in Shift-Or algorithm, the set of active states from Section 6.1 is implemented by bit vector.

In Formula 14, term $\mathbf{shl}(R_{i-1}^0)$ OR $D[t_i]$ represents matching transition—each active state is moved to the next position in the right[2] in the same level. All active states are moved at once and only the transitions corresponding to read symbol t_i are selected by mask vector $D[t_i]$, which changes 0 to 1 in each such state that its incoming matching transition is not labeled by t_i. The initial state of *NFA* is not in vector R^0 and it is implemented by inserting 0 at the beginning of the vector in operation **shl**()—initial state is always active because of its self loop.

At the beginning only the initial state is active therefore $R_0^0 = 1^{(m)}$.

If $r_{m,i}^0 = 0$, $0 < i \leq n$, then we report that the final state is active and thus the pattern is found ending at position i in text T. □

6.3.2.2 Approximate string matching using Hamming distance

In the approximate string matching using the Hamming distance, vectors R_i^l, $0 \leq l \leq k$, $0 \leq i \leq n$, are computed as follows:

$$\begin{array}{lcll} r_{j,0}^l & := & 1, & 0 < j \leq m, 0 \leq l \leq k \\ R_i^0 & := & \mathbf{shl}(R_{i-1}^0) \text{ OR } D[t_i], & 0 < i \leq n \\ R_i^l & := & (\mathbf{shl}(R_{i-1}^l) \text{ OR } D[t_i]) \text{ AND } \mathbf{shl}(R_{i-1}^{l-1}), & 0 < i \leq n, 0 < l \leq k \end{array} \quad (15)$$

In Formula 15 operation AND is bitwise operation `and`. Term $\mathbf{shl}(R_{i-1}^l)$ OR $D[t_i]$ represents matching and term $\mathbf{shl}(R_{i-1}^{l-1})$ represents edit operation *replace*—position i in text T is increased, position in pattern P is increased, and edit distance l is increased. Pattern P is found with at most k mismatches at position $t_{i-m+1} \ldots t_i$ if $r_{m,i}^k = 0$, $0 < i \leq n$. The maximum number of mismatches of the found string is $D_H(P, t_{i-m+1} \ldots t_i) = l$, where l is the minimum number such that $r_{m,i}^l = 0$.

An example of matrices R^l for searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ with at most $k = 3$ mismatches is shown in Table 6.6.

[2] In the Shift-Or algorithm, transitions from states in our figures are implemented by operation **shl**() (left shift) because of easier implementation in the case when number of states of *NFA* is greater than length of computer word and vectors R have to be divided into two or more bit vectors.

R^0	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
d	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**
R^1	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	1	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1
b	1	1	1	**0**	1	1	1	1	1	**0**	1	1	**0**	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**
R^2	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	1	1	**0**	1	**0**	**0**	1	1	**0**	**0**	1	**0**	**0**	1	1
b	1	1	1	1	**0**	1	1	1	1	1	**0**	1	1	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**
R^3	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	1	1	1	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	1
c	1	1	1	1	1	**0**	**0**	1	1	1	1	**0**	1	1	**0**	1
a	1	1	1	1	1	1	1	**0**	1	1	1	1	1	1	1	**0**

Table 6.6: Matrices R^l for the approximate string matching using the Hamming distance ($P = adbbca$, $k = 3$, and $T = adcabcaabadbbca$)

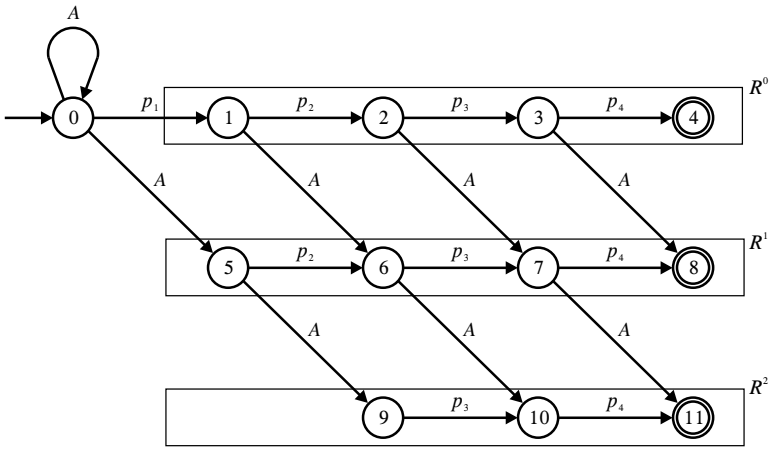


Figure 6.7: Bit parallelism uses one bit vector R for each level of states of *NFA*

Theorem 6.14
Shift-Or algorithm described by Formula 15 simulates a run of the *NFA* for the approximate string matching using the Hamming distance.

Proof
In Shift-Or algorithm for the approximate string matching there is for each level l, $0 \leq l \leq k$, of states of *NFA* one bit vector R_i^l, $0 \leq i \leq n$. So in Shift-Or algorithm, the set of active states from Chapter 6.1 is implemented by bit vectors—one vector for each level of states.

In Formula 15 term $\mathbf{shl}(R_{i-1}^l)$ OR $D[t_i]$ represents matching transition (see the proof of Theorem 6.13). Term $\mathbf{shl}(R_{i-1}^{l-1})$ represents transition *replace*—each active state of level $l-1$ is moved to the next depth in level l.

The selfloop of the initial state is implemented by inserting 0 at the beginning of vector R^0 within operation $\mathbf{shl}()$. 0 is inserted also at the beginning of each vector R^l, $0 < l \leq k$. Since the first state of l-th level is connected with the initial state by the sequence of l transitions labeled by A, each of these first states is active from l-th step of the simulation respectively till the end. The impact of 0s inserted at the beginning of vectors R^l does not appear before l-th step, therefore also vectors R^l simulate correctly the *NFA*.

If $r_{m,i}^l = 0$, $0 < i \leq n$, the final state of l-th level is active and we can report that the pattern is found with at most l errors ending at position i in the text. In fact, we report just only the minimum l in each step. □

6.3.2.3 Approximate string matching using Levenshtein distance

In the approximate string matching using the Levenshtein distance, vectors R_i^l, $0 \leq l \leq k$, $0 \leq i \leq n$, are computed by Formula 16. To prevent

insert transitions leading into final states we use auxiliary vector V defined by Formula 17.

$$\begin{array}{lcll} r^l_{j,0} & := & 0, & 0 < j \leq l, 0 < l \leq k \\ r^l_{j,0} & := & 1, & l < j \leq m, 0 \leq l \leq k \\ R^0_i & := & \mathbf{shl}(R^0_{i-1}) \text{ OR } D[t_i], & 0 < i \leq n \\ R^l_i & := & (\mathbf{shl}(R^l_{i-1}) \text{ OR } D[t_i]) & \\ & & \text{AND } \mathbf{shl}(R^{l-1}_{i-1} \text{ AND } R^{l-1}_i) & \\ & & \text{AND } (R^{l-1}_{i-1} \text{ OR } V), & 0 < i \leq n, 0 < l \leq k \end{array} \tag{16}$$

$$V = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix}, \text{ where } v_m = 1 \text{ and } v_j = 0, \forall j, 1 \leq j < m. \tag{17}$$

In Formula 16 term $\mathbf{shl}(R^l_{i-1})$ OR $D[t_i]$ represents matching, term $\mathbf{shl}(R^{l-1}_{i-1})$ represents edit operation *replace*, term $\mathbf{shl}(R^{l-1}_i)$ represents edit operation *delete*—position in pattern P is increased, position in text T is not increased, and edit distance l is increased. Term R^{l-1}_{i-1} represents edit operation *insert*—position in pattern P is not increased, position in text T is increased, and edit distance l is increased. Term *or* V provides that no *insert* transition leads from any final state.

Pattern P is found with at most k differences ending at position i if $r^k_{m,i} = 0$, $0 < i \leq n$. The maximum number of differences of the found string is l, where l is the minimum number such that $r^l_{m,i} = 0$.

An example of matrices R^l for searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ with at most $k = 3$ errors is shown in Table 6.7.

Theorem 6.15
Shift-Or algorithm described by Formula 16 simulates a run of the *NFA* for the approximate string matching using the Levenshtein distance.

Proof
The proof is similar to the proof of Theorem 6.14, we only have to add the simulation of *insert* and *delete* transitions.

In Formula 16 term $\mathbf{shl}(R^l_{i-1})$ OR $D[t_i]$ represents matching transition and term $\mathbf{shl}(R^{l-1}_{i-1})$ represents transition *replace* (see the proof of Theorem 6.14). Term R^{l-1}_{i-1} represents transition *insert*–each active state of level $l-1$ is moved into level l within the same depth. Term $\mathbf{shl}(R^{l-1}_i)$ represents transition *delete*–each active state of level $l-1$ is moved to the next depth in level l while no symbol is read from the input (position i in text T is the same).

R^0	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
d	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**
R^1	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	1	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	1	1	**0**
b	1	1	**0**	**0**	1	**0**	1	1	1	**0**	1	**0**	**0**	**0**	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	**0**	**0**	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	**0**	**0**
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	**0**
R^2	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	1	**0**	**0**	**0**	**0**	**0**	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
c	1	1	1	**0**	1	1	**0**	1	1	1	1	1	**0**	**0**	**0**	**0**
a	1	1	1	1	**0**	1	1	**0**	1	1	1	1	1	**0**	**0**	**0**
R^3	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
c	1	1	**0**	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
a	1	1	1	**0**	**0**	1	**0**	**0**	**0**	1	**0**	1	**0**	**0**	**0**	**0**

Table 6.7: Matrices R^l for the approximate string matching using the Levenshtein distance ($P = adbbca$, $k = 3$, and $T = adcabcaabadbbca$)

At the beginning, only the initial state q_0 and all states located on the same ε-diagonal like q_0 are active (i.e., all states of $\varepsilon-CLOSURE(\{q_0\})$ are active), therefore l-th bit of vector R^l, $0 < l \leq k$, is 0 in the initial setting of the vector. The states in front of l-th bit in vector R^l can also be 0, since they have no impact (l-th bit is always 0, since all states of $\varepsilon-CLOSURE(\{q_0\})$ are always active due to the selfloop of q_0). Therefore the bits behind l-th bit are set in the initial setting while the initial setting of the bits in front of l-th bit can be arbitrary.

In the case of the Levenshtein distance, the situation with inserting 0s at the beginning of vectors R^l, $0 < l \leq k$, during **shl**() operation is slightly different. Since all the states of the ε-diagonal leading from q_0 are always active, there is no impact of these 0 insertions. □

6.3.2.4 Approximate string matching using generalized Levenshtein distance In order to construct Shift-Or algorithm for the approximate string matching using the generalized Levenshtein distance [Hol97], we modify Formula 16 for the approximate string matching using the Levenshtein distance such that we add the term representing edit operation *transpose*. Since *NFA* for the approximate string matching using the generalized Levenshtein distance has on each transition *transpose* one auxiliary state, (see Figure 6.6) we have to introduce new bit vectors S_i^l, $0 \leq l < k$, $0 \leq i < n$, as follows:

$$S_i^l = \begin{bmatrix} s_{1,i}^l \\ s_{2,i}^l \\ \vdots \\ s_{m,i}^l \end{bmatrix}, 0 \leq l < k, 0 \leq i < n. \tag{18}$$

Vectors R_i^l, $0 \leq l \leq k$, $0 \leq i \leq n$, and S_i^l, $0 \leq l < k$, $0 \leq i < n$, are then computed as follows:

$$
\begin{array}{lcll}
r^l_{j,0} & := & 0, & 0 < j \leq l, \\
 & & & 0 < l \leq k \\
r^l_{j,0} & := & 1, & l < j \leq m, \\
 & & & 0 \leq l \leq k \\
R^0_i & := & \mathbf{shl}(R^0_{i-1}) \text{ OR } D[t_i], & 0 < i \leq n \\
R^l_i & := & (\mathbf{shl}(R^l_{i-1}) \text{ OR } D[t_i]) & \\
 & & \text{AND } \mathbf{shl}(R^{l-1}_{i-1} \text{ AND } R^{l-1}_i \text{ AND } (S^{l-1}_{i-1} \text{ OR } D[t_i])) & \\
 & & \text{AND } (R^{l-1}_{i-1} \text{ OR } V), & 0 < i \leq n, \\
 & & & 0 < l \leq k \\
s^l_{j,0} & := & 1, & 0 < j \leq m, \\
 & & & 0 \leq l < k \\
S^l_i & := & \mathbf{shl}(R^l_{i-1}) \text{ OR } \mathbf{shr}(D[t_i]), & 0 < i < n, \\
 & & & 0 \leq l < k
\end{array}
\tag{19}
$$

Term $\mathbf{shl}(R^l_{i-1})$ OR $D[t_i]$ represents matching, term $\mathbf{shl}(R^{l-1}_{i-1})$ represents edit operation *replace*, term $\mathbf{shl}(R^{l-1}_i)$ represents edit operation *delete*, and term R^{l-1}_{i-1} represents edit operation *insert*.

Term $(S^{l-1}_{i-1}$ OR $D[t_i])$ represents edit operation *transpose*–position in pattern P is increased by 2, position in text T is also increased by 2 but edit distance is increased just by 1. The increase of both positions by 2 is provided using vector S^l_i.

Pattern P is found with at most k differences ending at position i if $r^k_{m,i} = 0$, $0 < i \leq n$. The maximum number of differences of the found string is l, where l is the minimum number such that $r^l_{m,i} = 0$.

An example of matrices R^l and S^l for searching for pattern $P = adbbca$ in text $T = adbcbaabadbbca$ with at most $k = 3$ errors is shown in Table 6.8.

Theorem 6.16
Shift-Or algorithm described by Formula 19 simulates a run of the *NFA* for the approximate string matching using the generalized Levenshtein distance.

Proof
In Formula 19 term $\mathbf{shl}(R^l_{i-1})$ OR $D[t_i]$ represents matching transition and term $\mathbf{shl}(R^{l-1}_{i-1})$ represents transition *replace*, term R^{l-1}_{i-1} represents transition *insert*, and term $\mathbf{shl}(R^{l-1}_i)$ represents transition *delete* (see the proof of Theorem 6.15).

Term $(S^{l-1}_{i-1}$ OR $D[t_i])$ represents edit operation *transpose*. In this transition all states of level l are moved to the next position in the right $(\mathbf{shl}(R^l_{i-1}))$ of auxiliary level l'. Then only the transitions corresponding to input symbol t_i have to be selected. It is provided by mask vector $D[t_i]$. Since each transition leading to state of depth j is labeled by symbol p_{j+1}, we have to shift the mask vector in opposite direction than in which vector R^l_{i-1}

R^0		-	a	d	b	c	b	a	a	b	a	d	b	b	c	a
	a	1	**0**	1	1	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
	d	1	1	**0**	1	1	1	1	1	1	1	**0**	1	1	1	1
	b	1	1	1	**0**	1	1	1	1	1	1	1	**0**	1	1	1
	b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
	c	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
	a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**
S^0	a	1	1	**0**	1	1	1	1	1	1	1	**0**	1	1	1	1
	d	1	1	1	1	1	1	1	1	**0**	1	1	1	1	1	1
	b	1	1	1	**0**	1	1	1	1	1	1	1	**0**	1	1	1
	b	1	1	1	1	**0**	1	1	1	1	1	1	1	1	1	1
	c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R^1	a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	d	1	**0**	**0**	**0**	1	1	**0**	**0**	**0**	**0**	**0**	**0**	1	1	**0**
	b	1	1	**0**	**0**	**0**	1	1	1	**0**	1	**0**	**0**	**0**	1	1
	b	1	1	1	**0**	**0**	**0**	1	1	1	1	1	**0**	**0**	**0**	1
	c	1	1	1	1	**0**	**0**	1	1	1	1	1	1	**0**	**0**	**0**
	a	1	1	1	1	1	1	**0**	1	1	1	1	1	1	**0**	**0**
S^1	a	1	1	**0**	1	1	1	1	1	1	1	**0**	1	1	1	1
	d	1	1	1	**0**	1	**0**	1	1	**0**	1	1	**0**	**0**	1	1
	b	1	1	1	**0**	1	1	1	1	**0**	1	1	**0**	**0**	1	1
	b	1	1	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1
	c	1	1	1	1	1	1	**0**	1	1	1	1	1	1	1	**0**
	a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R^2	a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	d	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	b	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	b	1	1	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	c	1	1	1	**0**	**0**	**0**	**0**	1	1	1	1	**0**	**0**	**0**	**0**
	a	1	1	1	1	**0**	**0**	**0**	**0**	1	1	1	1	**0**	**0**	**0**
S^2	a	1	1	**0**	1	1	1	1	1	1	1	**0**	1	1	1	1
	d	1	1	1	**0**	1	**0**	1	1	**0**	1	1	**0**	**0**	1	1
	b	1	1	1	**0**	1	**0**	1	1	**0**	1	1	**0**	**0**	1	1
	b	1	1	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1
	c	1	1	1	1	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
	a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R^3	a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	d	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	b	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	b	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	c	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	a	1	1	1	**0**	**0**	**0**	**0**	**0**	1	**0**	1	**0**	**0**	**0**	**0**

Table 6.8: Matrices R^l and S^l for the approximate string matching using the generalized Levenshtein distance ($P = adbbca$, $k = 3$, and $T = adbcbaabadbbca$)

is shifted ($\mathbf{shr}(D[t_i])$). All states of auxiliary level l' are stored in vector $S_i^l := \mathbf{shl}(R_{i-1}^l)$ OR $\mathbf{shr}(D[t_i])$, $0 < i < n$, $0 \leq l < k$.

In next step $i+1$ of the computation all states of auxiliary level l' are moved to the next position in the right of level $l+1$ and only the transitions corresponding to input symbol t_{i+1} are selected by mask vector $D[t_{i+1}]$. Since each transition leading to state of depth $j+1$ is labeled by symbol p_j, we have to shift mask vector $D[t_{i+1}]$ in the same direction in which vector S_i^l is shifted. Therefore we insert term $\mathbf{shl}(S_{i-1}^{l-1}$ OR $D[t_i])$ in Formula 16 as well as we insert the formula for computation of vector S_i^l in Formula 16. □

6.3.3 Other methods of bit parallelism

In the previous Sections we described only Shift-Or algorithm. If we exchange the meaning of 0s and 1s and the usage of ANDs and ORs in the formulae presented, we get Shift-And algorithm [WM92].

Other method of bit parallelism is Shift-Add algorithm [BYG92]. In this algorithm we have one bit-vector, which contains m blocks of bits of size $b = \lceil \log_2 m \rceil$ (one block for each depth of *NFA* for the approximate string matching using the Hamming distance). The formula for computing such vector then consists of shifting the vector by b bits and adding with the mask vector for current input symbol t. This vector contains 1 in $(b.j)$-th position, if $t \neq p_{j+1}$, or 0, elsewhere. This algorithm runs in time $\mathcal{O}(\lceil \frac{mb}{w} \rceil n)$.

We can also use Shift-Add algorithm for the weighted approximate string matching using the Hamming distance. In such case each block of bits in mask vector contains binary representation of weight of the corresponding edit operation *replace*. We have also to enlarge the length of block of bits to prevent a carry to the next block of bits. Note, that Shift-Add algorithm can also be considered as an implementation of dynamic programming.

In [BYN96a] they improve the approximate string matching using the Levenshtein distance in such a way, that they search for any of the first $(k+1)$ symbols of the pattern. If they find any of them, they start *NFA* simulation and if the simulation then reaches the initial situation (i.e., only the states located in ε-diagonal leading from the initial state are active), they again start searching for any of the first $(k+1)$ symbols of the pattern, which is faster than the simulation.

Shift-Or algorithm can also be used for the exact string matching with don't care symbols, classes of symbols, and complements as shown in [BYG92]. In [WM92] they extend this method to unlimited wild cards and apply the above features for the approximate string matching. They also use bit parallelism for regular expressions, for the weighted approximate string matching, for set of patterns, and for the situations, when errors are not allowed in some parts of pattern (another mask vector is used).

All the previous cases have good time complexity if the *NFA* simulation

fits in one computer word (each vector in one computer word). If *NFA* is larger, it is necessary to partition *NFA* or pattern as described in [BYN96b, BYN96a, BYN97, BYN99, NBY98, WM92].

Shift-Or algorithm can also be used for multiple pattern matching [BYG92, BYN97] or for the distributed pattern matching [HIMM99].

6.3.4 Time and space complexity

The time and space analysis of the Shift-Or algorithm is as follows [WM92]. Denote the computer word size by w. The preprocessing requires $\mathcal{O}(m|A|)$ time plus $\mathcal{O}(k\lceil\frac{m}{w}\rceil)$ to initialize the k vectors. The running time is $\mathcal{O}(nk\lceil\frac{m}{w}\rceil)$. The space complexity is $\mathcal{O}(|A|\lceil\frac{m}{w}\rceil)$ for mask matrix D plus $\mathcal{O}(k\lceil\frac{m}{w}\rceil)$ for the k vectors.

References

[Abr87] K. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987.

[BYG92] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[BYN96a] R. A. Baeza-Yates and G. Navarro. A fast heuristic for approximate string matching. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proceedings of the 3rd South American Workshop on String Processing*, pages 47–63, Recife, Brazil, 1996. Carleton University Press.

[BYN96b] R. A. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In D. S. Hirschberg and E. W. Myers, editors, *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, number 1075 in Lecture Notes in Computer Science, pages 1–23, Laguna Beach, CA, 1996. Springer-Verlag, Berlin.

[BYN97] R. A. Baeza-Yates and G. Navarro. Multiple approximate string matching. In F. K. H. A. Dehne, A. Rau-Chaplin, J.-R. Sack, and R. Tamassia, editors, *Proceedings of the 5th Workshop on Algorithms and Data Structures*, number 1272 in Lecture Notes in Computer Science, pages 174–184, Halifax, Nova Scotia, Canada, 1997. Springer-Verlag, Berlin.

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

[CH97a] R. Cole and R. Hariharan. Tighter upper bounds on the exact complexity of string matching. *SIAM J. Comput.*, 26(3):803–856, 1997.

[CH97b] M. Crochemore and C. Hancart. Automata for matching patterns. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 2 Linear Modeling: Background and Application, chapter 9, pages 399–462. Springer-Verlag, Berlin, 1997.

[Döm64] B. Dömölki. An algorithm for syntactical analysis. *Computational Linguistics*, (3):29–46, 1964.

[GL89] R. Grossi and F. Luccio. Simple and efficient string matching with k mismatches. *Inf. Process. Lett.*, 33(3):113–120, 1989.

[GP89] Z. Galil and K. Park. An improved algorithm for approximate string matching. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, number 372 in Lecture Notes in Computer Science, pages 394–404, Stresa, Italy, 1989. Springer-Verlag, Berlin.

[HIMM99] J. Holub, C. S. Iliopoulos, B. Melichar, and L. Mouchard. Distributed string matching using finite automata. In R. Raman and J. Simpson, editors, *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms*, pages 114–128, Perth, WA, Australia, 1999.

[Hol97] J. Holub. Simulation of NFA in approximate string and sequence matching. In J. Holub, editor, *Proceedings of the Prague Stringology Club Workshop '97*, pages 39–46, Czech Technical University, Prague, Czech Republic, 1997. Collaborative Report DC–97–03.

[LV88] G. M. Landau and U. Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37(1):63–78, 1988.

[MC97] C. Moore and J. P. Crutchfield. Quantum automata and quantum grammars. 1997. `http://xxx.lanl.gov/abs/quant-ph/9707031`.

[Mel95] B. Melichar. Approximate string matching by finite automata. In V. Hlaváč and R. Šára, editors, *Computer Analysis of Images and Patterns*, number 970 in Lecture Notes in Computer Science, pages 342–349. Springer-Verlag, Berlin, 1995.

[NBY98] G. Navarro and R. Baeza-Yates. Improving an algorithm for approximate pattern matching. Technical Report TR/DCC-98-5, Dept. of Computer Science, University of Chile, 1998.

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

[Shy76] R. K. Shyamasundar. A simple string matching algorithm. Technical report, Tata Institute of Fundamental Research, 1976.

[Ukk85] E. Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1–3):132–137, 1985.

[WF74] R. A. Wagner and M. Fischer. The string-to-string correction problem. *J. Assoc. Comput. Mach.*, 21(1):168–173, 1974.

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