

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



# Automaton-based Backward Pattern Matching

Doctoral Thesis

*Jan Antoš*

PhD program: Computer Science and Engineering

*Advisor: Bořivoj Melichar*

February 2010

# AUTOMATON-BASED BACKWARD PATTERN MATCHING

*Jan Antoš*

`antosj@fel.cvut.cz`

Department of Computer Science and Engineering

Faculty of Electrical Engineering

Czech Technical University in Prague

Karlovo nám. 13, 121 35 Prague 2, CZ

## **Abstract**

The thesis proposes an universal approach to solve a large class of pattern matching problems by using the backward pattern matching technique. The motivation of the thesis is not to take a specific problem and solve it better then before, but to take a whole class of problems and propose a single approach to solve all problems from the given class. The proposed approach allows to quickly derive solutions for new pattern matching problems belonging to the same class of problems. Similar work has been already done for the forward-matching technique. Using the backward pattern matching technique for the same task has the potential to run in a sub-linear time. The proposed approach is called Universal Backward Pattern Matching Machine and it is based on the finite automata theory.

## **Keywords**

backward pattern matching, string matching, sequence matching, approximate pattern matching, sub-pattern matching, don't care symbols, finite automata, formal tool, pattern matching classification, automata construction

## Acknowledgments

I would like to thank a number of people who made this thesis possible. First of all it is my thesis supervisor, prof. Bořivoj Melichar. I thank him for his ideas, suggestions, patience, flexibility and friendliness. Without him I would not be able to come to this point, where I can present a finished doctoral thesis. Then it is my parents for their support and motivation during all those years of my doctoral studies. And last, but not least, it is my wife Zuzana for her understanding when I dedicated my free time to my studies instead of to her.

This research has been partially supported by The Ministry of Education, Youth and Sports under the research program MSMT 6840770014 and by The Czech Science Foundation as projects No. 201/09/0807, No. 201/06/1039 and No. 201/01/1433.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Historical Context . . . . .	1
1.2	Motivation . . . . .	2
1.3	Organization of the Thesis . . . . .	3
<b>2</b>	<b>Basic Definitions</b>	<b>4</b>
<b>3</b>	<b>Theoretical Background</b>	<b>9</b>
3.1	Backward Pattern Matching . . . . .	9
3.2	Classification of Pattern Matching Problems . . . . .	13
3.3	Projection of Pattern and Pattern Image . . . . .	16
<b>4</b>	<b>Universal Backward Pattern Matching Machine</b>	<b>17</b>
4.1	Principle of Universal Backward Pattern Matching Machine . . . . .	17
4.2	Selected Implementation Options . . . . .	18
4.2.1	The $f$ Function . . . . .	18
4.2.2	The Shift Function . . . . .	19
4.3	Constructor Overview . . . . .	19
4.3.1	Input of constructor . . . . .	19
4.3.2	Output of constructor . . . . .	20
4.3.3	Construction method . . . . .	20
4.4	Executor Overview . . . . .	20
4.5	Description of the Problem Instance . . . . .	21
4.6	Note on Number of Studied Pattern Matching Problems . . . . .	22
<b>5</b>	<b>Executor</b>	<b>23</b>
5.1	Algorithm . . . . .	23
5.2	Examples . . . . .	26
5.2.1	Example of Exact Pattern Matching . . . . .	26
5.2.2	Example of Approximate Pattern Matching . . . . .	26
5.3	Time and Space Complexity . . . . .	27
5.3.1	Time complexity of proposed algorithm . . . . .	27
5.3.2	Space complexity of proposed algorithm . . . . .	29
5.4	Proof of Algorithm's Correctness . . . . .	30

<b>6</b>	<b>Constructor</b>	<b>33</b>
6.1	The Constructor's Algorithm . . . . .	33
6.2	Construction of Backward Pattern Matching Automaton . . . . .	34
6.3	Construction of Automaton with Zero Input Degree of Initial State . . . . .	35
6.4	Construction of Extended Nondeterministic Finite Automaton . . . . .	36
6.4.1	Example . . . . .	37
6.4.2	Time and Space complexity . . . . .	37
6.5	Determinization of Extended Finite Automaton . . . . .	38
6.5.1	Example . . . . .	38
6.5.2	Time and Space Complexity . . . . .	39
6.6	Minimization of Extended Finite Automaton . . . . .	39
6.6.1	Time and Space Complexity . . . . .	41
6.6.2	Example . . . . .	41
6.7	Construction of Reversed Projection Automaton . . . . .	42
6.7.1	Approach to the Construction of RPA . . . . .	42
6.7.2	Reversed Projection Automaton for SFOECO Problem . . . . .	43
6.7.3	Reversed Projection Automaton for SFFECO Problem . . . . .	44
6.7.4	Reversed Projection Automaton for SFIECO Problem . . . . .	45
6.7.5	Reversed Projection Automaton for SF?EDO Family of Problems . . . . .	48
6.7.6	Reversed Projection Automaton for SF?R?O Family of Problems . . . . .	49
6.7.7	Reversed Projection Automaton for SF?D?O Family of Problems . . . . .	51
6.7.8	Reversed Projection Automaton for SF?T?O Family of Problems . . . . .	52
6.7.9	SS???O Model Construction . . . . .	54
<b>7</b>	<b>Conclusion and Future Work</b>	<b>56</b>
7.1	Conclusion . . . . .	56
7.2	Future Work . . . . .	56
7.2.1	Speed Optimizations . . . . .	56
7.2.2	Alternative Implementations . . . . .	57
7.2.3	Average Time Complexity . . . . .	58
7.2.4	Size of Backward Pattern Matching Automaton . . . . .	58

# Chapter 1

## Introduction

### 1.1 Historical Context

Pattern matching (string and sequence matching) is an essential part of many applications, including (but not limiting) text editing, word processing, image recognition, data retrieval, information mining, symbol manipulation, genetics alignment, musical processing, etc. This discipline has been intensively studied since the beginning of seventies. Exact matching of one string in a text is based on two historical articles from the year 1977: Knuth, Morris and Pratt described a forward matching algorithm [KMP77] and Boyer and Moore presented a backward pattern matching algorithm [BM77]. This backward pattern matching algorithm enables to skip parts of the text and therefore performs faster on average than the forward pattern matching algorithm.

While the initial algorithms were matching a single string in a text, more complex pattern matching problems then emerged and been extensively studied in the following decades, for example: approximate matching based on Hamming [H50], Levenshtein [L65] and Damerau [D66] distances, sequence matching [CKK72], multiple string matching [AC75, CW79b], regular expressions [K56], etc.

For the above mentioned problems many new algorithms were invented. Many of these algorithms are presented in [S09, CHL07].

A good example of the ongoing interest in pattern matching problems and the gradual design of algorithms that cover the ever larger set of problems is the extension of a matching of a single pattern to a matching of a set of patterns. The original algorithm for exact backward pattern matching of a single pattern is Boyer-Moore [BM77]. Comments-Walter [CW79a, CW79b] later developed an algorithm for exact backward pattern matching of set of patterns. That algorithm is based on Boyer-Moore approach combined with Aho-Corasick [AC75]. (A complete version can be found in [A90].) Later, Uratani [U88], Baeza-Yates and Régnier [BR90] and Crochemore et al. [CCG99] developed similar algorithms. Then the algorithms for approximate backward pattern matching of set of patterns appeared as for example Fredriksson and Navarro [FN04].

The list could then continue by the modification to do approximate pattern matching of infinite set of patterns or to do approximate pattern matching of sequences of patterns etc. This illustrates not only that there are completely different algorithms to solve

simple variants of the same pattern matching problem but also that the ongoing interest in pattern matching problems requires to find new algorithms for “a combination” of problems.

In 1996 a formalism was found, which allows principles of matching algorithms to be formally specified. This formalism is based on a finding, that all single-dimensional matching algorithms are sequential problems and thus can be solved by the utilization of finite automata [MH98].

Taxonomies and classifications have been created to organize the ever-growing set of pattern matching problems [W95]. Of these classifications, one especially interesting is the classification presented in [MH97]. That classification is not (and cannot be) complete, but it classifies 192 different pattern matching problems in a six-dimensional space. Together with the new formalism (at [MH98]) it resulted in an interesting fact: Having a finite automaton to describe the pattern matching problem of one string in a text, all the other 191 problems can be solved by simple operations applied to this one automaton [MH97]. Only a forward matching technique was explored in [MH97] leaving the question open, if the similar approach can be defined to solve all above mentioned pattern matching problems using the backward pattern matching algorithm.

The present state of knowledge is well presented for example in [CHL07].

## 1.2 Motivation

Today there are many backward pattern matching algorithms solving various pattern matching problems [CHL07]. Yet each of these algorithms is usually solving just one specific pattern matching problem.

While the usual motivation for pattern matching research is to take a specific problem and solve it better than before, our approach is different. We take a whole class of problems and propose a single approach to solve all problems from the given class with the expectation that for specific application the solution can be adapted and eventually optimized. We choose such class that is open to addition of new problems. The great benefit of designing a universal solution to all the problems of such class is that when a new problem is found and it belongs to the given class, the solution is immediately ready.

Such universal solution was already described for the forward pattern matching techniques in [MHP05], yet there is no universal solution known for the backward pattern matching technique. The aim of this thesis is to show that such universal solution is possible if the pattern matching algorithm is separated from the formal description of the pattern matching problem instance.

Both the algorithm for pattern matching and the construction of the formal description of the pattern matching problem instance is proposed. The thesis shows the construction using examples covering the key problems from the 6D classification of pattern matching problems [MH97].

The proposed approach does not present a performance improvement in comparison to existing algorithms but it allows to quickly derive solutions for new pattern matching

problems belonging to the same class of problems. If needed, it can be optimized for specific subclasses of problems as required by a particular application.

Problems covered by the proposed approach are for example: approximate pattern matching, pattern matching of finite or infinite set of patterns, don't care symbols, subwords search, searching for sequences, etc. and their combination as for example: approximate pattern matching of infinite set of patterns specified by a regular expression. For many of these combinations a backward pattern matching algorithm does not yet exist.

### 1.3 Organization of the Thesis

This dissertation thesis is organized in the following manner: After the definition of common notions in Chapter 2, the pattern matching problem specification is given in Chapter 3. The overview of the proposed approach called Universal Backward Pattern Matching Machine is given in Chapter 4. The description of the Executor part of the machine that performs the pattern matching is explained in Chapter 5. Chapter 6 proposes the approach how to realize the Constructor part of the machine that constructs the finite automaton describing the instance of pattern matching problem. Conclusion and future work chapter comes in the end of the thesis together with the references.

# Chapter 2

## Basic Definitions

**Definition 2.1** (Alphabet). An *alphabet*  $A$  is a finite non-empty set of symbols. The number of symbols will be denoted by  $|A|$ .

**Definition 2.2** (Complement of symbol). Given an alphabet  $A$  and a symbol  $a \in A$ , the *complement* of  $a$  according to  $A$  is a set of symbols  $\bar{a} = \{s : s \in A, s \neq a\}$ .

**Remark.** Because of some limitations of the graphic program used to visualize the transition diagram of an automaton, the complement of symbol  $a$  in the transition diagram is rendered as  $-a$  instead of  $\bar{a}$ .

**Definition 2.3** (String). A *string* (*word*) over the given alphabet  $A$  is a finite sequence of symbols of the alphabet  $A$ . The empty sequence  $\varepsilon$  is called an *empty string*. The *length of string*  $w$  is the number of symbols in the string  $w$  and is denoted by  $|w|$ .

**Remark.** A particular *substring* of string  $s$ , where substring starts at position  $i$  of the string  $s$  and ends at positions  $j$  (inclusive), will be denoted as  $s_i \dots s_j$ .

**Definition 2.4** (Reversed string). Having a string  $s = s_1 \dots s_n$  where  $n = |s|$  a *reversed string* is string  $s^R = s_n \dots s_1$ . Having a set of strings  $S$ , a set of all reversed strings from set  $S$  is denoted as  $S^R$ .

**Definition 2.5** (Factor, prefix, suffix, antifactor). A string  $x$  is said to be a *factor* (*substring*) of string  $y$  if  $y = uxv$  for some strings  $u, v$ , a *prefix* of  $y$  if  $u = \varepsilon$ , a *suffix* of  $y$  if  $v = \varepsilon$  and *antifactor* if  $x$  is not a factor of  $y$ . The set of all factors, prefixes, suffixes and antifactors of a string  $s$  will be denoted by  $fact(s)$ ,  $pref(s)$ ,  $suff(s)$  and  $antifact(s)$  respectively.

**Remark.** Functions  $fact$ ,  $pref$ ,  $suff$ ,  $pref^+$  and  $suff^+$  are also defined for the set of strings  $S$  in this manner:  $fact(S) = \bigcup_{s \in S} fact(s)$ .

**Definition 2.6** (Proper prefix, proper suffix). *Proper prefix* (*proper suffix*) of a string is such prefix (suffix) that is not equal to the string itself and not empty. The set of all proper prefixes and proper suffixes of string  $s$  will be denoted by  $pref^+(s)$  and  $suff^+(s)$  respectively.

**Definition 2.7** (Suffix-closed). A *suffix-closed* set of strings  $S$  is such set, that all suffixes of all strings from the set are also members of the set:  $suff(s) \subseteq S$  for  $\forall s \in S$ .

**Definition 2.8** (Edit operation replace). Edit operation *replace* is an operation which converts string  $vaw$  to string  $vbw$ , where  $v, w \in A^*$ ,  $a, b \in A$ ,  $a \neq b$  (one symbol is replaced by another).

**Definition 2.9** (Edit operation insert). Edit operation *insert* is an operation which converts string  $vw$  to string  $vaw$ , where  $v, w \in A^*$ ,  $a \in A$  (one symbol is inserted into a string).

**Definition 2.10** (Edit operation delete). Edit operation *delete* is an operation which converts string  $vaw$  to string  $vw$ , where  $v, w \in A^*$ ,  $a \in A$  (one symbol is deleted from a string).

**Definition 2.11** (Edit operation transpose). Edit operation *transpose* is an operation which converts string  $vabw$  to string  $vbaw$ , where  $v, w \in A^*$ ,  $a, b \in A$ ,  $a \neq b$  (two adjacent symbols are exchanged).

**Definition 2.12** (Hamming distance). *Hamming distance*  $D_H(v, w)$  between two strings of the same length  $v, w \in A^*$ , is a minimum number of edit operations *replace* needed to convert  $v$  to  $w$ .

**Definition 2.13** (Levenshtein distance). *Levenshtein distance*  $D_L(v, w)$  between two strings  $v, w \in A^*$ , is a minimum number of edit operations *replace*, *insert* and *delete* needed to convert  $v$  to  $w$ .

**Definition 2.14** (Damerau distance). *Damerau distance*  $D_D(v, w)$  between two strings  $v, w \in A^*$ , is a minimum number of edit operations *replace*, *insert*, *delete* and *transpose* needed to convert  $v$  to  $w$ . Each symbol  $a$  from the string  $v$  can participate at most in one edit operation *transpose*.

**Definition 2.15** (Approximate string matching). *Approximate string matching* is defined as a searching for all occurrences of pattern  $P = p_1p_2p_3 \dots p_m$  in text  $T = t_1t_2t_3 \dots t_n$  with at most  $k$  errors allowed. When  $k = 0$  then it does not matter what kind of distance we are using and we are talking about *exact string matching*.

**Definition 2.16** (Language). A *language*  $L$  over the alphabet  $A$  is an arbitrary subset of  $A^*$ , i.e.  $L \subseteq A^*$ .

**Definition 2.17** (Powerset). A *powerset* of set  $S$  is the set of all subsets of  $S$  and is denoted by  $\mathcal{P}(S)$ .

**Definition 2.18** (Nondeterministic finite automaton). A *nondeterministic finite automaton (NFA)*  $M$  is a five-tuple  $M = (Q, A, \delta, q_0, F)$ , where

- $Q$  is a finite set of states,
- $A$  is a finite input alphabet,
- $\delta$  is a mapping  $Q \times (A \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$ ,
- $q_0 \in Q$  is the initial state,
- $F \subset Q$  is the set of final states.

**Definition 2.19** (Deterministic finite automaton). A finite automaton  $M$ ,  $M = (Q, A, \delta, q_0, F)$  is *deterministic finite automaton (DFA)* if  $\forall q \in Q \quad \forall a \in A \quad |\delta(q, a)| \leq 1$ .

**Remark.** The symbols  $\delta^+(q, w)$  and  $\delta^*(q, w)$ ,  $q \in Q$ ,  $w \in A^*$ , denote a transitive and transitive reflexive closure of the mapping  $\delta$ , respectively. We will omit the  $+$  and  $*$  symbols when it is clear that  $w$  is a string.

**Definition 2.20** (Finite automaton). A *finite automaton (FA)* is nondeterministic finite automaton (NFA) or deterministic finite automaton (DFA).

**Definition 2.21** (Configuration of FA). A *configuration of finite automaton*  $M = (Q, A, \delta, q_0, F)$  is a pair  $(q, p) \in Q \times A^*$ . The *initial configuration of FA* is a pair  $(q_0, r)$  and a *final (accepting) configuration of FA* is a pair  $(q_f, \varepsilon)$ , where  $q_f \in F$ .

**Definition 2.22** (Transition of FA). A *transition of FA*  $M = (Q, A, \delta, q_0, F)$  is a relation  $\vdash_M \subset (Q \times A^*) \times (Q \times A^*)$  defined as  $(q, aw) \vdash_M (p, w)$  where  $p \in \delta(q, a)$ ,  $a \in A \cup \{\varepsilon\}$ ,  $w \in A^*$ ,  $p, q \in Q$ . The  $k$ -power of the relation  $\vdash_M$  will be denoted by  $\vdash_M^k$ . The symbols  $\vdash_M^+$  and  $\vdash_M^*$  denote a transitive and transitive reflexive closure of the relation  $\vdash_M$ , respectively.

**Definition 2.23** (Transition diagram). A *transition diagram* will be called the oriented labeled graph representing the finite automaton. The nodes of the graph are labeled by the names of the states, the edges correspond to transitions. Each edge (transition) is labeled by a symbol which has been read during a move corresponding to this edge or by  $\varepsilon$  in case of nondeterministic automaton. An initial state will be denoted by an arrow, a final state will be denoted by a double circle.

**Remark.** If there is more than one edge from the state  $q$  to the state  $p$ , these edges can be drawn as one edge labeled by the set of symbols (this is used to simplify the graphics representation, but it still means that there is a set of edges). Likewise, if there are edges for all symbols from the alphabet  $A$  going from state  $q$  to  $p$ , they can be displayed by one edge labeled by a word *all*. If the transition is labeled *other* it means that labeled edge is representing the set of transitions for all the symbols from the alphabet except those that are explicitly stated as leading from state  $q$ .

**Definition 2.24** (Move of FA). A *move of finite automaton* is such a change of configuration of the finite automaton, that exactly one symbol has been read from the automaton input.

**Definition 2.25** (String accepted by FA). We will say that an input string  $w \in A^*$  is *accepted* by a finite automaton  $M = (Q, A, \delta, q_0, F)$  if  $(q_0, w) \vdash_M^* (q, \varepsilon)$  for some  $q \in F$ .

**Definition 2.26** (Language accepted by FA). The language *accepted* by a finite automaton  $M = (Q, A, \delta, q_0, F)$  is a language  $L(M) = \{w \mid w \in A^*, (q_0, w) \vdash_M^* (q, \varepsilon) \text{ for some } q \in F\}$ . The automaton accepting the language  $L$  will be denoted by  $M(L)$ .

**Definition 2.27** (Equivalence of finite automata). Two finite automata  $M$  and  $M'$  are *equivalent* if  $L(M) = L(M')$ .

**Definition 2.28** (Minimal deterministic finite automaton). Deterministic finite automaton  $M = \{Q, A, \delta, q_0, F\}$  is *minimal* if no other deterministic automaton  $M'$  accepting the same language  $L(M') = L(M)$  has less states than  $M$ .

**Definition 2.29** (Inaccessible state). We say that a state  $q \in Q$  is *inaccessible* in a finite automaton  $M = (Q, A, \delta, q_0, F)$  if  $(q_0, vw) \not\vdash_M^* (q, w)$ ,  $v, w \in A^*$ .

**Definition 2.30** (Useless state). We say that a state  $q \in Q$  is *useless* in a finite automaton  $M = (Q, A, \delta, q_0, F)$  if  $(q, w) \not\vdash_M^* (q_F, \varepsilon)$ ,  $w \in A^*$ ,  $q_F \in F$ .

**Definition 2.31** ( $\varepsilon$ -transition). The transitions labelled by  $\varepsilon$  will be called  $\varepsilon$ -transitions.

**Remark.** Because of some limitations of the graphic program used to visualize the transition diagram of an automaton, the label of  $\varepsilon$ -transition in the transition diagram is rendered as *eps* instead of  $\varepsilon$ .

**Definition 2.32** (Active state). An *active state* of FA  $M = (Q, A, \delta, q_0, F)$  after reading input string  $w \in A^*$  is each state  $q \in Q$  such that  $(q_0, w) \vdash_M^* (q, \varepsilon)$ .

**Definition 2.33** (Intersection of finite automata). Let's have a finite automaton  $M_1(L_1)$  accepting language  $L_1(A)$  and a finite automaton  $M_2(L_2)$  accepting  $L_2(A)$ . Then automaton  $M(L_1 \cap L_2)$  is called an *intersection of automata*  $M_1$  and  $M_2$ .

**Definition 2.34** (Outgoing transition, Output degree). An *outgoing transition* of state  $q$  is such transition  $t$  where  $\exists a \in A$ ;  $\delta(q, a) \neq \emptyset$ . Number of such transitions is called *output degree of state  $q$*  and denoted  $out\_deg(q)$ .

**Definition 2.35** (Incoming transition, Input degree, Target state). An *incoming transition* to state  $q$  is such transition  $t$  where  $\exists a \in A, \exists s \in Q$ ;  $\delta(s, a) = q$ . Number of such transitions is called *input degree of state  $q$*  and denoted  $in\_deg(q)$ . State  $q$  is called *target state of transition  $t$* .

**Definition 2.36** (Regular expression). A *regular expression*  $V$  over an 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 2.37** (Value of regular expression). A *value*  $h(x)$  of a regular expression  $x$  is defined as follows:

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

**Definition 2.38** (Pattern Matching Problem). *Pattern matching problem* is the type of pattern matching that we want to perform. Examples are: Exact matching of one string, exact matching of a finite set of strings, approximate matching, matching of all substrings, etc. Pattern matching problem will be denoted as  $\theta$ .

**Definition 2.39** (Pattern). *Pattern* is the string that we would like to match. Set of patterns is either finite or infinite set of patterns specified by an enumeration or by regular expression. We will denote pattern by  $p$  and set of patterns by  $P$ .

**Remark.** To make the explanation more simple we presume that no pattern set can contain an empty string ( $\varepsilon$ ). This also applies for the subpattern matching:  $\varepsilon$  is not considered a subpattern.

**Remark.** *Instance of a pattern matching problem* is the combination of the problem and the set of patterns. For example: instance of pattern matching problem “exact matching of one string” and pattern “banana” is “exact matching of string banana”.

**Definition 2.40** (Projection of Pattern, Image of Pattern). *Projection of pattern* is such function that for given pattern matching problem  $\theta$  and pattern  $p$  has a value of the set of all strings in alphabet  $A$  that are considered match to the given pattern. We will denote this function  $proj_{\theta,A}(p)$  or simply  $proj(p)$ . For set of patterns  $P$  the function is defined as  $proj(P) = \bigcup_{p \in P} proj(p)$ . The strings produced by projection are called *pattern images*. We will denote a pattern image by  $g$  and the set of pattern images by  $G$ . It holds  $proj(p) = g$  and  $proj(P) = G$ .

# Chapter 3

## Theoretical Background

### 3.1 Backward Pattern Matching

Pattern matching algorithms have evolved during past decades. Naive forward pattern matching algorithm is easy to understand and implement. Its principle is given by Algorithm 3.1 and example is given by Figure 3.2.

Please note: variable *position* in the algorithm represents a position in the text. Positions are starting from 1. Variable *offset* represents an offset between the position in the text and the actually compared symbol.

Algorithm 3.1: NAIVE FORWARD PATTERN MATCHING ALGORITHM

Input: Text  $T$ , pattern  $p$

Output: Set of numbers, each number represents a position in text  $T$   
where pattern  $p$  occurs

Method:

```
1   for  $\forall position \in \langle 1, |T| - |p| \rangle$  do
2        $offset \leftarrow 0$ 
3       while  $offset < |p|$  and  $T_{position+offset} = p_{offset+1}$  do
4            $offset \leftarrow offset + 1$ 
5       end while
6       if  $offset = |p|$  then  $output(position)$ 
7   end for
```

The main disadvantage is the time complexity of this algorithm. It has a complexity of  $O(m.n)$ , where  $m$  is the length of pattern and  $n$  is the length of text. Reason for this high complexity is the fact, that the same text is compared over and over again.

The forward pattern matching algorithm has been improved in the past up to its maximum effectiveness: Each symbol of input text is compared exactly once, thus giving the time complexity of  $O(n)$ . Such algorithm is outlined in Algorithm 3.3 and example is shown in Figure 3.4. Another approach for fastest forward pattern matching algorithm is the use of deterministic finite automaton [MHP05].

The introduction of backward pattern matching has further decreased the time complexity. The main principle of the backward pattern matching is that the text is compared

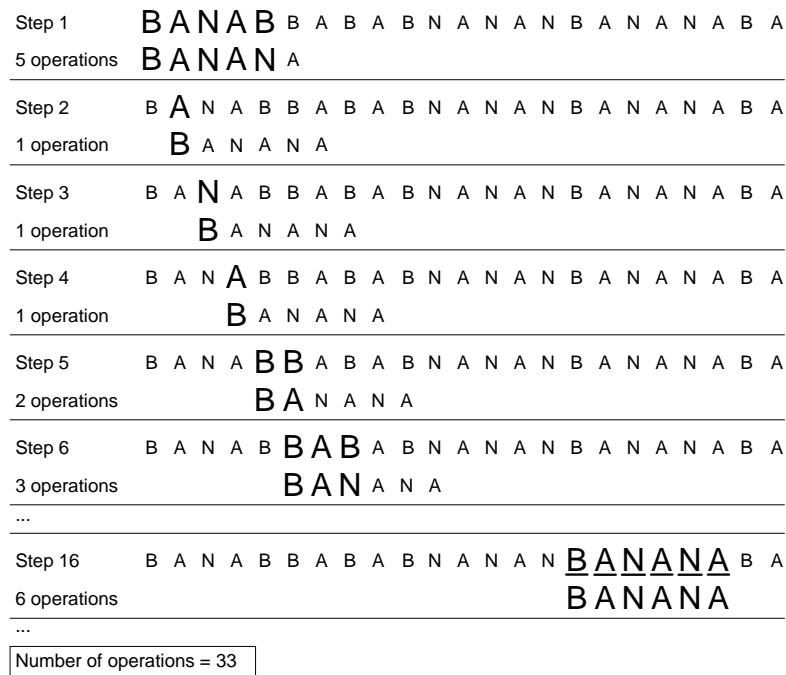


Figure 3.2: Naive forward pattern matching algorithm: matching of pattern BANANA

from left to right, but the pattern is compared from the right to left. See Algorithm 3.5 for description (algorithm is adapted from [CWZ04]; function  $f$  will be explained further in the text) and Figure 3.6 for an example of a particular algorithm.

Backward pattern matching is able to skip parts of text (line 11 of Algorithm 3.5). If we examine the time complexity of this algorithm, it still would be  $O(n.m)$  in the worst case, but it will get close to  $O(n/m)$  on average. Especially when using alphabets with large number of symbols (for example: ascii or unicode files, ...) this average complexity is easily met. An optimization for some pattern matching problems exists that has the upper bound of time complexity  $O(n)$  [MHP05].

The Algorithm 3.5 has several implementation options available.

First such implementation option of Algorithm 3.5 is the selection of  $f$  function mentioned on line 4 ( $f$  function is citation from [CWZ04]).  $f$  function is such a function

$$f \in \mathcal{P}(A^*) \rightarrow \mathcal{P}(A^*)$$

satisfying

$$p \in f(p) \wedge \text{su}ff(f(p)) \subseteq f(p).$$

I.e.  $f$  is such that  $p$  is included in  $f(p)$  and  $f(p)$  is suffix-closed. For this function then holds (for all  $w, x \in A^*$ )

$$w \notin f(p) \Rightarrow w \neq p \quad \text{and} \quad w \notin f(p) \Rightarrow xw \neq p$$

where  $p$  is a pattern. The  $f$  function can be realized by a range of functions, for example by:  $\text{su}ff(p)$ ,  $\text{fact}(p)$ ,  $\text{factoracle}(p)$  [ACR99], ...

Algorithm 3.3: THE FASTEST FORWARD PATTERN MATCHING ALGORITHM

Input: Text  $T$ , pattern  $p$

Output: Set of numbers, each number represents a position in text  $T$  where pattern  $p$  occurs

Method:

```

1  offset ← 0
2  for  $\forall position \in \langle 1, |T| - |p| \rangle$  do
3      if  $T_{position} \neq p_{offset+1}$  then
4          get shift (compute or lookup in table)
5          offset ← offset - shift + 1
6      else
7          offset ← offset + 1
8      end if
9      if offset =  $|p|$  then
10         output(position - offset + 1)
11         get shift (compute or lookup in table)
12         offset ← offset - shift + 1
13     end if
14 end for

```

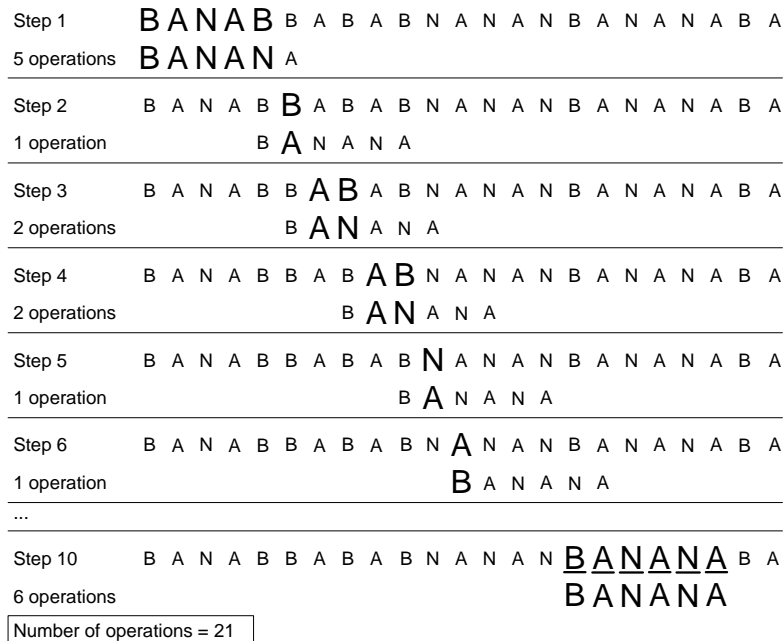


Figure 3.4: The fastest forward pattern matching algorithm: matching of pattern BANANA

Algorithm 3.5: BACKWARD PATTERN MATCHING ALGORITHM

Input: Text  $T$ , pattern  $p$

Output: Set of numbers, each number represents a position in text  $T$  where pattern  $p$  occurs

Method:

```

1  position ← |p|
2  while position ∈ ⟨|p|, |T|⟩ do
3      offset ← |p|
4      while  $T_{position-(|p|-offset)} \dots T_{position} \in f(p)$  do
5          offset ← offset - 1
6      end while
7      if offset = 0 then
8          output(position - |p|)
9      end if
10     compute shift
11     position ← position + shift
12 end while

```

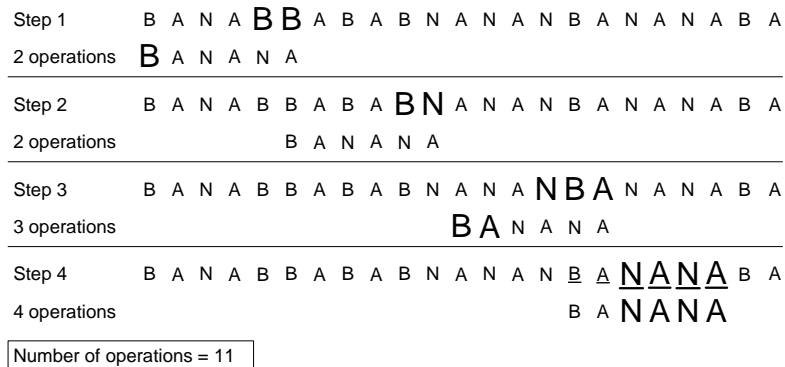


Figure 3.6: Backward pattern matching algorithm: matching of pattern BANANA

The second implementation option is the implementation of the

$$T_{position-(|p|-offset)} \cdots T_{position} \in f(p)$$

condition. For algorithm to work efficiently it needs to be done in constant time.

The third implementation option is the selection of shift function (line 10 of Algorithm 3.5). The shift function can be either precomputed prior to the pattern matching or it can be dynamically computed on the fly. The shift function computes the safe shift that the algorithm can perform in the text. The safe shift is such advance of the position in the text that guarantees that no occurrence of pattern is skipped. There are several suboptimal shift functions known today. The selection of the shift function directly affects the average time complexity of the algorithm.

Examples of three different approaches to the shift function are presented in Figure 3.7 [M05]. They differ in what part of the pattern is used for the shift computation. These shift functions are also known as good prefix shift function [L92], repeated suffix shift function [CCG91] and antifactor shift function [ACR99].

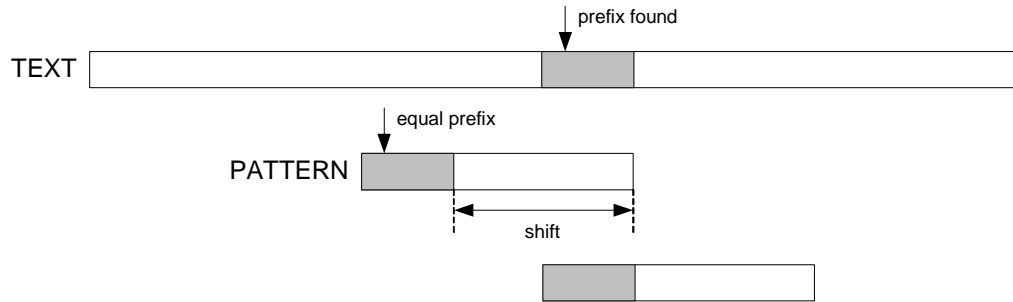
## 3.2 Classification of Pattern Matching Problems

Classification of pattern matching problems has been described in [MH97]. This section presents a brief extract of the main ideas. See [MH97] for full details.

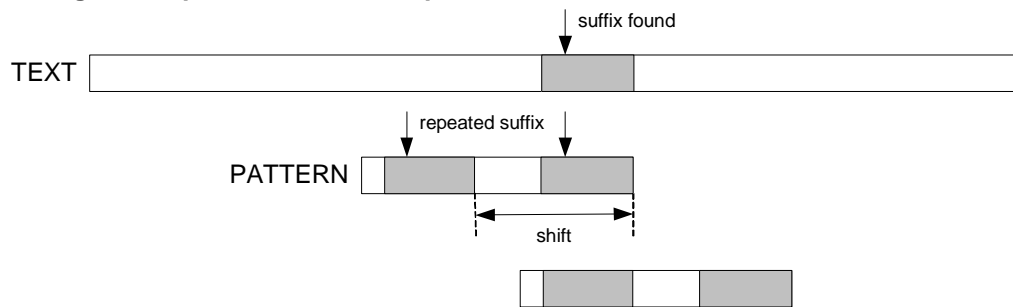
Pattern matching problems for finite size alphabets can be classified according to several criteria. We will use six criteria for classification leading to six-dimensional space in which one point corresponds to particular pattern matching problem. Let us make a list of all dimensions including possible values in each dimension:

1. Nature of the pattern:
  - (a) string,
  - (b) sequence.
2. Integrity of the pattern:
  - (a) full pattern,
  - (b) subpattern.
3. Number of patterns:
  - (a) one,
  - (b) finite number,
  - (c) infinite number.
4. The way of matching:
  - (a) exact,
  - (b) approximate matching with Hamming distance (R-matching),

**Looking for a prefix of the pattern:**



**Looking for a repeated suffix of the pattern:**



**Looking for an antifactor of the pattern:**

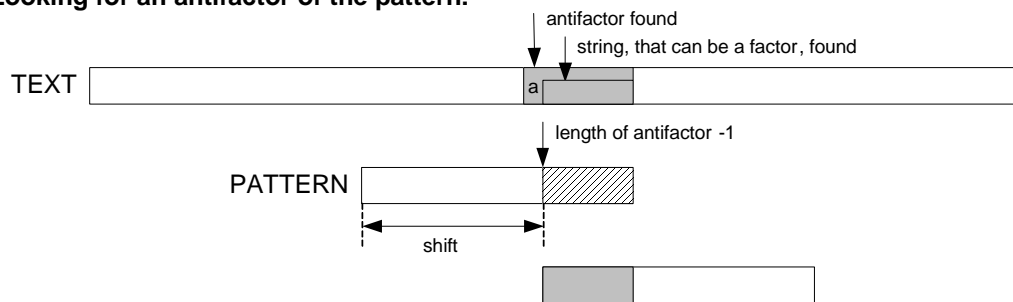


Figure 3.7: Various options of backward pattern matching shift computations [M05]

- (c) approximate matching with Levenshtein distance (DIR-matching),
  - (d) approximate matching with Damerau distance (DIRT-matching).
5. Importance of symbols in pattern:
    - (a) take care of all symbols,
    - (b) don't care of some symbols.
  6. Number of instances of pattern:
    - (a) one,
    - (b) finite sequence.

The above classification is visualized in Figure 3.8. If we count the number of possible pattern matching problems, we obtain

$$N = 2.2.3.4.2.2 = 192.$$

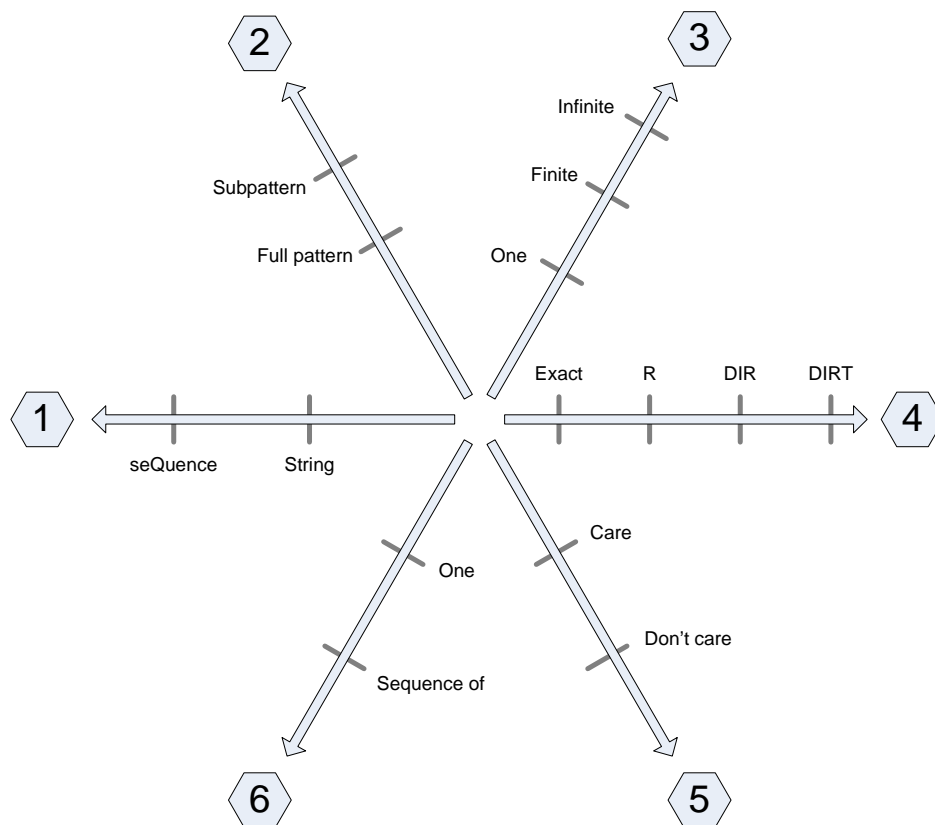


Figure 3.8: 6D Classification of pattern matching problems

In order to make references to particular pattern matching problem easy, we will use abbreviations for all problems. These abbreviations are summarized in Table 3.1. Using this method, we can, for example, refer to exact string matching of one string as SFOECO problem.

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

We will denote a pattern matching problem by symbol  $\theta$  and a set of pattern matching problems by symbol  $\Theta$ . A single pattern matching problem can be then written, for example, as  $\theta = SFOECO$ . Similar to this a statement  $\Theta = SFO???$  denotes a family of pattern matching problems.  $\theta \in SFO???$  means any single pattern matching problem, which is a member of given family of problems.

Dimension	1	2	3	4	5	6
	S	S	O	E	C	O
	Q	F	F	R	D	S
			I	D		
				T		

Table 3.1: Abbreviations of dimension values

### 3.3 Projection of Pattern and Pattern Image

Let us have a pattern matching problem and a set of patterns. If solving the exact string matching of one (or many) full string then the set of patterns specified is also the set of patterns that we match in the text. But if choose for example approximate pattern matching, then the strings matched in the text can differ from the patterns by the specified approximate distance. In that case we call such string that is considered to be a match for the given pattern a *pattern image*. The function computing for a pattern its image is called *projection* function. See Definitions 2.38, 2.39, 2.40.

An example: Let us have pattern matching problem SFORCO (R stands for approximate pattern matching according to Hamming distance), maximum distance of matching strings  $k_{max}$  and set of patterns  $\{banana\}$ . The the projection function is  $proj_{SFORCO,A}(s) = \forall w \in A^+ : D_H(s, w) \leq k_{max}$ . The set of pattern images is then  $G = proj_{SFORCO,A}(banana)$ . In similar way:

$$\begin{aligned} \theta = SFOECO, P = \{banana\} &\Rightarrow G = \{banana\}, \\ \theta = SFFRCO, P = \{apple, pear\} &\Rightarrow \\ G = \{w : w \in A^*, D_H(w, apple) < k_{max}\} \cup \{w : w \in A^*, D_H(w, pear) < k_{max}\} \\ \theta = SSOECO, P = \{banana\} &\Rightarrow G = \{w : w \in fact(banana)\}. \end{aligned}$$

# Chapter 4

## Universal Backward Pattern Matching Machine

### 4.1 Principle of Universal Backward Pattern Matching Machine

Usually a pattern matching problem is hard-coded into the algorithm. That limits the use of the algorithm to solve that specific pattern matching problem. In order to achieve true universal algorithm we need to provide way how to separate two aspects of the algorithm: the description of the instance of pattern matching problem and the actual pattern matching.

The mechanism that we propose, a *Universal Backward Pattern Matching Machine*, has two parts as can be seen in Figure 4.1. The first part is *Constructor* that takes the specification of the pattern matching problem and the set of patterns and produces a formalized description of instance of the problem in the form of extended deterministic finite automaton. The second part is *Executor* that takes the automaton produced by the constructor and the text and using the appropriate method (based on the selected  $f$  function and selected shift function) it performs the pattern matching. The executor is repeatedly using the automaton and a set of internal variables to match the patterns in the text.

In this way the executor part of the algorithm is completely universal and can solve a

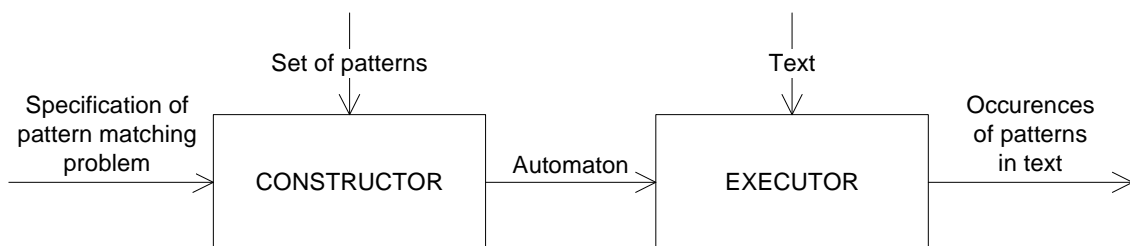


Figure 4.1: The schema of Universal Backward Pattern Matching Machine with Constructor and Executor parts

vast number of problems. Further text of this dissertation thesis shows how to solve at least 48 different pattern matching problems by this approach.

The constructor is briefly described in Section 4.3 and in detail in Chapter 6. The executor is briefly described in Section 4.4 and in detail in Chapter 5.

## 4.2 Selected Implementation Options

The Universal Backward Pattern Matching Machine is based on Algorithm 3.5. There are several implementation options possible. In this thesis we have explored Universal Backward Pattern Matching Machine for just one possible implementation option. This option is described in this section.

Please note, that by selecting different implementation options (i.e. different  $f$  and shift functions) the resulting Universal Backward Pattern Matching Machine will work in different way. It will still have The Constructor and The Executor parts but they will implement different algorithms and the automaton produced by The Constructor will also accept different languages.

### 4.2.1 The $f$ Function

Various  $f$  functions (see Section 3.1) result in different Universal Backward Pattern Matching Machines. The selection of  $f$  function has impact on the way how the problem instance is described by the finite automaton and what is the algorithm in executor part.

In this thesis we present the Universal Backward Pattern Matching Machine realizing  $f$  function as  $fact(G)$  where  $G$  is a set of pattern images. It is used in  $T_i \dots T_j \in f(P)$  condition.

The executor will use extended deterministic finite automaton  $M$  produced by constructor to implement the  $f$  function and test the condition. Since we perform backward pattern matching, we will put the text backwards on the automaton input. This means that the automaton will accept reversed strings.

Reversed function  $f$  is  $f^R = (fact(G))^R = fact(G^R)$ .

The automaton produced by constructor will be suffix automaton accepting the language of  $L(M) = suff(G^R)$ .

We use the following relation between functions:

$$suff(G^R) \subset fact(G^R) \quad \text{and} \quad antifact(G^R) \cap suff(G^R) = \emptyset$$

to recognize all factors of  $G^R$ . The automaton itself will not accept language of all factors but the transition function for the reversed factor will always be defined:

$$w \in fact(P) \Rightarrow |\delta(q_0, w^R)| \geq 1,$$

even if none of the  $\delta(q_0, w^R)$  states are final. In the opposite the transition function for antifactor will never be defined:

$$w \notin fact(P) \Rightarrow \delta(q_0, w^R) = \emptyset.$$

Thus we can use even suffix automaton as a tool for detecting the reversed factors in the text.

## 4.2.2 The Shift Function

The proposed approach will compute the shift function by the good prefix shift approach [L92] (see Figure 3.7). The method is looking for the longest prefix that ends at the given position and then computes the shift based on the length of prefix. We will use the extended deterministic finite automaton constructed by the constructor to look for the prefixes of pattern images. Because we read the text backwards we will put the reversed substrings of text on the automaton input. The automaton will accept the language of reversed suffixes of pattern images:

$$L(M) = (\text{pref}(G))^R = \text{suff}(G^R).$$

The shift function is computed using the longest prefix found and the length of shortest pattern image from  $G$ .

## 4.3 Constructor Overview

At this section we will describe The Constructor part of The Universal Backward Pattern Matching Machine. We will first explain inputs and output of the constructor and then we will describe how it works.

### 4.3.1 Input of constructor

An input to the constructor is:

- specification of backward pattern matching problem,
- set of patterns.

We need some formalism how to specify the pattern matching problem. There can be many ways how to formally specify it. We will use the formalism based on six-dimensional classification of the pattern matching problems presented in [MH97]. This classification is described in Section 3.2 and it describes 192 different pattern matching problems as points in six dimensional space. These points are represented as six-letter long abbreviation representing values of each dimension. An example of the specification of the pattern matching problem is SFFECO problem that represents [String matching, Full pattern, Finite number of patterns, Exact matching, only Care symbols, One string].

### 4.3.2 Output of constructor

Output of the constructor is a formal description of the instance of the pattern matching problem. We choose the formalism based on the finite automata theory because all single-dimensional pattern matching problems can be described by the finite automata [MH98]. The output of the constructor is the extended deterministic finite automaton defined in Section 4.5. This automaton represents the instance of pattern matching problem by accepting the language of all reversed suffixes of all pattern images.

The pattern matching problem is expressed in the way how the set of patterns is projected to pattern images. The constructor itself does not take the projection function as an input but it composes the resulting projection function according to the 6-D specification of the pattern matching instance. How exactly is the projection function composed is explained in Chapter 6.

### 4.3.3 Construction method

At first an automaton accepting the language of reversed pattern images is constructed. The language contains all possible strings that are considered a match with any of given patterns according to the pattern matching problem. To construct this automaton only a limited number of algorithms is needed. These algorithms perform simple automaton operations to derive the resulting automaton. Example of similar approach for forward pattern matching problems can be found in [MH98], [MHP05].

Next the automaton representing the  $f$  function is constructed. The resulting automaton is extended – it has three types of finite states as described in Section 4.5. The resulting automaton is deterministic and minimal. The construction and subsequent determinization and minimization of the extended automaton is described in Section 6.

## 4.4 Executor Overview

The executor takes two inputs:

- the formalized description of the instance of the pattern matching problem in the form of extended deterministic finite automaton,
- the text upon which we would like to apply the instance of the problem.

It produces as the output a list of indices of the text where there is the first symbol of a matched pattern.

The executor is using a small number of scalar variables to compute the shift function on the fly. The executors' algorithm is the same for all solved pattern matching problems. The algorithm is studied in detail in Chapter 5. Its short summary is given here:

1. It starts on the  $|P|_{min}$  position.

2. For each position it reads the text backwards and puts the symbols on the automaton input.
3. By reaching final states in the automaton the algorithm is able to decide if it found the pattern in the text or if it found the prefix of a pattern.
4. It uses the scalar variables to keep the track of the position in the text and also to compute the length of the longest prefix found at the respective position.
5. When there is no transition in the automaton for the given input symbol it is clear that the antifactor has been found. The shift function is computed at that time based on the longest prefix found so far.
6. The position is shifted according to the shift function and the automaton starts again in its initial state.

## 4.5 Description of the Problem Instance

The instance of the pattern matching problem is formally described as the extended deterministic finite automaton that we call *Backward Pattern Matching Automaton* (Definition 4.1). This automaton is used in the executor to perform the pattern matching. Chapter 6 explains in detail how to construct these automata for different pattern matching problems.

**Definition 4.1** (Backward Pattern Matching Automaton). *Backward Pattern Matching Automaton (BPMA)* is a seven-tuple  $M = (Q, A, \delta, q_0, F_p, F_s, F_{ps})$ , where

$Q$  is a finite set of states,

$A$  is a finite input alphabet,

$\delta$  is a mapping  $Q \times A \rightarrow Q$ ,

$q_0 \in Q$  is the initial state,

$F_p \subset Q, F_s \subset Q, F_{ps} \subset Q$  are mutually disjoint sets of final states, i.e.  $(F_p \cap F_s = \emptyset) \wedge (F_p \cap F_{ps} = \emptyset) \wedge (F_s \cap F_{ps} = \emptyset)$ .

Sets  $F_p, F_s$  and  $F_{ps}$  are such, that if automaton  $M_p = (Q, A, \delta, q_0, F_p)$  is accepting  $L(M_p)$  then automaton  $M_s = (Q, A, \delta, q_0, F_s)$  is accepting  $L(M_s) = \text{suffix}^+(L(M_p)) \setminus L(M_p)$  and automaton  $M_{ps} = (Q, A, \delta, q_0, F_{ps})$  is accepting  $L(M_{ps}) = \text{suffix}^+(L(M_p)) \cap L(M_p)$ .

**Remark.** In transition diagrams we will denote  $q \in F_p$  by full inner circle,  $q \in F_s$  by dotted inner circle and  $q \in F_{ps}$  by dashed inner circle.

By having three disjoint sets of final states Backward Pattern Matching Automaton is accepting three languages  $L_p, L_s$  and  $L_{ps}$  that together represent the language of all reversed pattern images  $G^R$  and of all suffixes of all reversed pattern images  $\text{suffix}^+(G^R)$ :

$$L_p = G^R \setminus \text{suffix}^+(G^R),$$

$$L_s = \text{suffix}^+(G^R) \setminus G^R,$$

$$L_{ps} = G^R \cap \text{suffix}^+(G^R).$$

In less formal way the explanation of three disjunct sets of final states is simple: we will construct our BPMA automaton in such way, that by reaching some state from  $F_p$  we know, that we have put the reversed pattern image on the automaton input and thus we can report match. By reaching some state from  $F_s$  we know that we have put the reversed proper prefix of some pattern image (more precisely a proper suffix of reversed pattern image) on the automaton input and thus we can compute the shift function according to it. By reaching some state from  $F_{ps}$  we know, that the string on the automaton input is both a reversed pattern image and also a reversed proper prefix of another pattern image and thus we can report match and also compute the shift function according to it.

The reason why we would like to differentiate between these three sets of final states is explained in the Chapter 5 in the description of the algorithm.

An example of such BPMA for exact pattern matching of two patterns *bana* and *ban* is given in Figure 4.2.

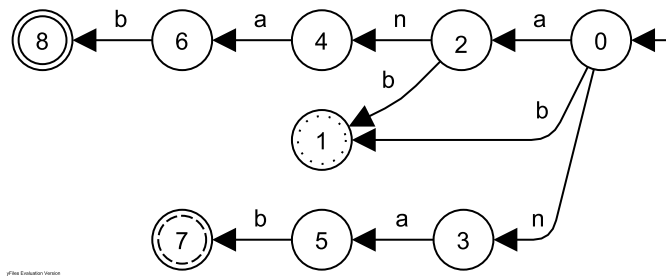


Figure 4.2: Example of BPMA for SFFECO problem and set of patterns  $\{bana, ban\}$

## 4.6 Note on Number of Studied Pattern Matching Problems

The number of problems solved by this thesis is limited to the family of S????O problems. This family includes exact or approximate, string or substring matching problems of one, finite or infinite set of problems with optional don't care symbols. It includes 48 pattern matching problems.

The S????O family does not include sequence matching and matching of sequence of patterns. The presented approach can solve these problems as well but the average time complexity for Q????? and ?????S problems will in most cases reach upper bound of executor's algorithm  $O(n \cdot |G|_{max})$  (see Section 5.3.1). Together with  $|G|_{max}$  being in these cases equal to the length of text  $|G|_{max} = |T| = n$  the algorithm will have time complexity of  $O(n^2)$ . Further optimizations are required to solve these problems efficiently. See Section 7.2.1.

# Chapter 5

## Executor

This chapter describes the executor part of the Universal Backward Pattern Matching Machine described in Chapter 4. The Executor is using the selected implementation option of  $f = fact(G)$  function and good-prefix shift function (see Section 4.2).

The concept of separation between constructor and executor is the key idea that enables to create truly universal algorithm for large number of problems. The description of the problem instance is only one of the input parameters of the algorithm. Algorithm is thus completely abstracted from the pattern matching problem. The examples of problems solved by the algorithm described in this thesis are only illustrations what the algorithm is capable of. The algorithm is universal in such way that if some presently unknown pattern matching problem will be introduced in the future it will be solvable by the proposed algorithm given the condition that a description of the problem instance is constructed in the form of backward pattern matching automaton.

The executor has as an input the description of the instance of pattern matching problem and the text in which we want to perform the actual pattern matching. The description of the problem instance is the extended deterministic finite automaton called Backward Pattern Matching Automaton that we will denote by  $M$  in this chapter. The text will be denoted by  $T$ .

### 5.1 Algorithm

The executor's algorithm is described in Algorithm 5.1.

The input variable  $shift_{max}$  is the maximum safe shift. The maximum safe shift is equal to the shortest pattern image  $|G|_{min}$ . The maximum safe shift for the given instance of problem described by BPMA is

$$shift_{max} = |L(M_{p,ps})|_{min}$$

where

$$M_{p,ps} = (Q, A, \delta, q_0, F_p, \emptyset, F_{ps}).$$

The breadth-first graph traversal algorithm [K97] can be used to compute the  $shift_{max}$  value. Such computation can be done by the constructor but to keep the Figure 4.1 simple we have not mentioned it.

To keep the algorithm simple, it requires the text  $T$  to begin with a special symbol *Start Of Text*:  $SOT \notin A$  which will work as efficient guard to protect the algorithm of running out of the text scope.

Algorithm 5.1: THE EXECUTOR ALGORITHM OF UNIVERSAL BACKWARD PATTERN MATCHING MACHINE

Input: Description of problem instance  $M$  in the form of Backward Pattern Matching Automaton  $M = (Q, A, \delta, q_0, F_p, F_s, F_{ps})$ , maximum safe shift  $shift_{max}$  and text  $T$

Output: Set of numbers, each number represents a position in text  $T$  where the first symbol of pattern  $p \in P$  occurs

Method:

```

1  position ← shiftmax
2  offset ← 0
3  plc ← 0      (note: plc = prefix length counter)
4  tc ← 0      (note: tc = transition counter)
5  q ← q0
6  while position ≤ |T| do
7      if δ(q, Tposition-offset) ≠ ∅ then
8          q ← δ(q, Tposition-offset)
9          tc ← tc + 1
10         if q ∈ Fs ∪ Fps then plc ← tc end if
11         if q ∈ Fp ∪ Fps then output(position - offset) end if
12         offset ← offset + 1
13     else
14         shift ← max{1, shiftmax - plc}
15         position ← position + shift
16         offset ← 0
17         plc ← 0
18         tc ← 0
19         q ← q0
20     end if
21 end while

```

The algorithm is using following variables:

- *Position* stores the actual position in the text from where the pattern is compared (backward). Text is indexed from 1, so if position equals 1 it points at the first symbol of text.
- *Offset* stores the offset in the text from the position. *Offset* stores positive numbers and thus the actual position of the symbol that is put to automaton input is  $position - offset$ . *Offset* can also be interpreted as number of symbols that were successfully compared – i.e. how many symbols were put on the automaton input and were accepted since the last initialization of the automaton.
- *Prefix length counter* (abbreviated *plc*) stores the longest proper prefix found from the last initialization of the automaton.

- *Transition counter* (abbreviated *tc*) stores the number of transitions that the automaton performed from its last initialization. This number is used for computing the length of the string that has been at that moment put on the input. (Variable *tc* can be avoided and offset could be used instead. Variable *tc* is used in the algorithm for the better readability.)
- *Shift* stores the number of symbols, that the algorithm will move forward after there is no transition function defined for the given input symbol.
- Variables *q* and *q'* store the states of the automaton.

The algorithm repeatedly uses the BPMA automaton to match the pattern images (and their prefixes) ending at given position. This is done by reading the symbols of text backward from the position in text, putting these symbols on the automaton input and evaluating the automaton states. After the given position is processed, algorithm computes the shift, moves to the next position in the text, resets all its variables and does the matching again. The following steps explain this process in more detail.

1. The algorithm sets the initial position to the length of the shortest pattern image that is represented by the  $shift_{max}$  constant (line 1). All the other variables are set to initial values (lines 2-5).
2. For given position in text  $T$  the algorithm outputs the match if there is some pattern image ending at that position, i.e. if exists number  $x$  such, that

$$T_x \dots T_{position} \in G.$$

For given  $w = T_x \dots T_{position}$  this situation occurs if

$$\delta(q_0, w^R) \in F_p \cup F_{ps}.$$

In this case, the value of  $x$  is put to algorithm output (line 11).

3. Simultaneously with Step 2 the algorithm also has to determine the longest proper prefix ending at the given position. It computes  $|w|_{max}$  where

$$w \in pref^+(G) \wedge w = T_{position-|w|} \dots T_{position}.$$

The algorithm evaluates if  $w$  is a proper prefix of a pattern image by examining whether

$$\delta(q_0, w^R) \in F_s \cup F_{ps}.$$

If  $w$  is a proper prefix of some pattern image, its length is stored in the variable *plc* (line 10). The length of previously found proper prefix is overwritten. The algorithm reads symbols  $T_{position}, T_{position-1}, \dots$  until it reads the first symbol  $T_{position-n}$  such, that for string  $v = T_{position-n} \dots T_{position}$  is

$$\delta(q_0, v^R) = \emptyset.$$

After that situation (line 7), the variable *plc* stores the length of the longest proper prefix ending at that position.

4. After computing the longest proper prefix  $plc_{max}$  for the given position in the text, the algorithm computes the longest safe shift (line 14). Safe shift means how much a position in a text can be advanced in order not to skip any occurrence of a pattern. Trivial safe shift is 1. The longest safe shift can never be longer than the shortest pattern image which is  $|G|_{min}$ . Since we know, that there is a possibility of pattern occurring at position  $position - plc_{max}$ , and we know  $|G|_{min}$ , the shift of  $|G|_{min} - plc_{max}$  will be safe. So, summarized, the shift function is

$$shift = \max\{1, |G|_{min} - plc_{max}\}.$$

In the algorithm  $|G|_{min}$  is represented by  $shift_{max}$  and  $plc_{max}$  by  $plc$ .

Note at this point of time, why we are using the proper prefixes instead of prefixes. The reason is each string is also a prefix of itself but not a proper prefix of itself:

$$\forall s \in A^+ : s = pref(s), s \neq pref^+(p).$$

The shift function based on prefixes (rather than proper prefixes) would equal 1 after each match, which is inefficient.

5. The algorithm advances its position by the  $shift$  value (line 15):

$$position \leftarrow position + shift$$

and resets its variables (lines 16-19). This reset includes also new initialization of the automaton (line 19). The algorithm then repeats Steps 2 through 5 of this explanation until the end of the text is reached (line 6).

## 5.2 Examples

### 5.2.1 Example of Exact Pattern Matching

At first let us demonstrate the algorithm on a simple problem of exact string pattern matching. For pattern matching problem SFOECO and pattern *banana* the set of all pattern images is  $G = \{banana\}$ . Backward Pattern Matching Automaton is shown in Figure 5.2. The execution of the executor's algorithm is visualized in Figure 5.3. As you can see, the number of operations (in the means of number of states processed and also the number of symbols read) is 14 while with the fastest forward matching algorithm must read at least 23 symbols.

### 5.2.2 Example of Approximate Pattern Matching

The second example shows the power of the algorithm. With the same algorithm we will solve approximate pattern matching problem. The executor's algorithm does not change but only Backward Pattern Matching Automaton looks different for the same pattern *banana*. For problem  $\theta = SFORCO$  with a maximum Hamming distance  $k_{max} = 1$  and pattern *banana* the set of all pattern images is  $G = \{\forall w \in A^+ : D_H(s, w) \leq 1\}$ . To explain the set of pattern images less formally we can define symbol  $\circ$  which represents

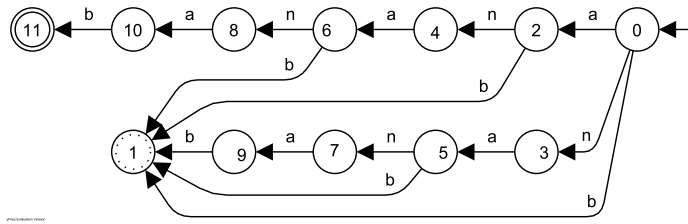


Figure 5.2: Backward Pattern Matching Automaton for SFOECO problem and pattern *banana*

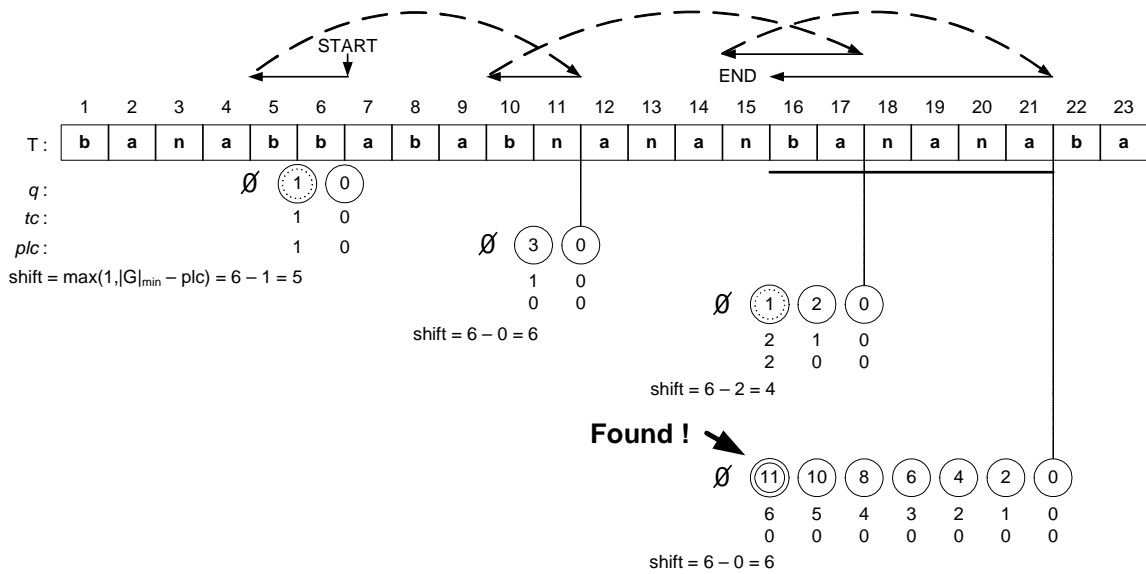


Figure 5.3: Visualization of executor algorithm for SFOECO problem and pattern *banana*

any symbol of the alphabet. The set of all pattern images is then  $G = \{\circ anana, b \circ nana, ba \circ ana, ban \circ na, bana \circ a, banano\}$ . Backward Pattern Matching Automaton  $M$  is shown in Figure 5.4. The execution of the executor's algorithm is visualized in Figure 5.5.

The number of compared symbols is 31 which is not that efficient as the fastest forward-matching algorithm which would compare only 23 symbols. Future work is needed to optimize the algorithm for the better efficiency (see Section 7.2.1). Nevertheless the power of the algorithm is its universality which has been shown on these two examples – same algorithm solved exact and also approximate pattern matching problems.

### 5.3 Time and Space Complexity

#### 5.3.1 Time complexity of proposed algorithm

**Theorem 5.1.** Algorithm 5.1 will output positions of all pattern occurrences in  $O(\frac{n}{|G|_{min}})$  time in the best case and in  $O(n \cdot |G|_{max})$  in the worst case.  $G$  is the set of all pattern images,  $n$  is the length of text.

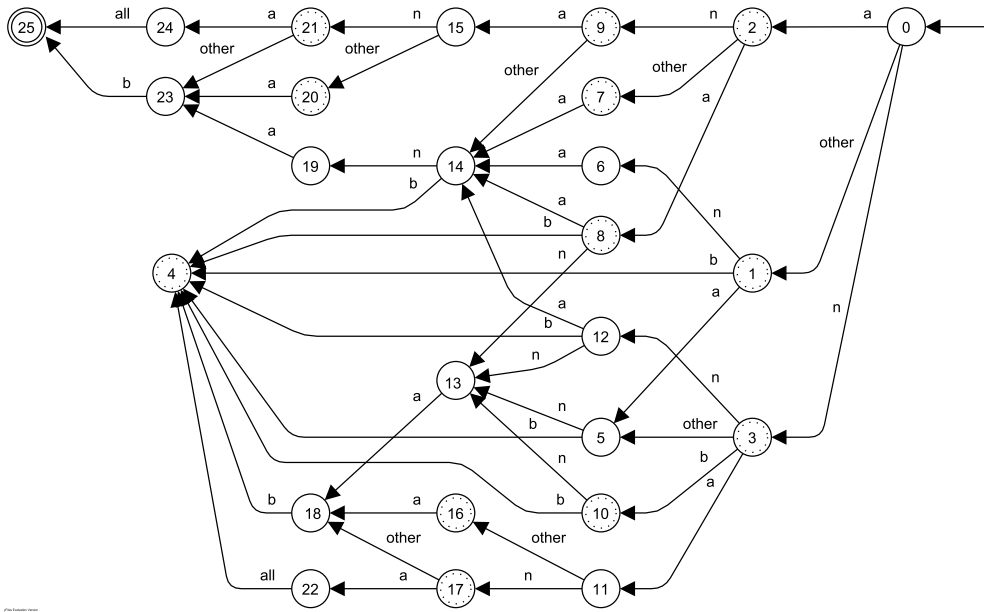


Figure 5.4: Backward Pattern Matching Automaton for SFORCO problem and pattern *banana*

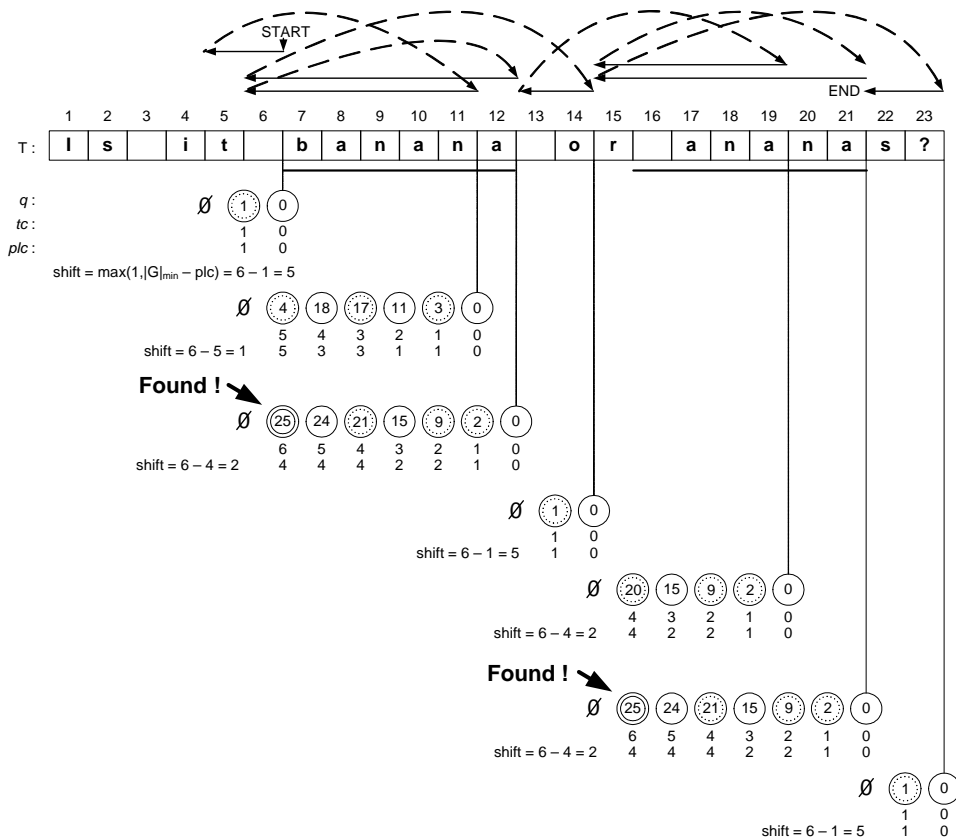


Figure 5.5: Visualization of The Executor's Algorithm for SFORCO problem and pattern *banana*

*Proof.* The time complexity of the proposed algorithm is dependent on the number of executions of its while loop. It is guarded by the content of the *position* variable with respect to the length of text which is  $n = |T|$ . The *position* variable is incremented in the else part of the if statement. It can never be decremented. The largest increment is  $shift_{max} = |G|_{min}$ . Thus, the lower bound of the time complexity is  $O(\frac{n}{|G|_{min}})$ .

The while loop can iterate even without the incrementation of *position* variable if the condition in the if statement is true. In that case *offset* is incremented by one. The maximum number of consequent iterations when the if condition is true is at most the longest path in the backward pattern matching automaton. This path is equivalent to the longest pattern image, i.e.  $|G|_{max}$ . After that many iterations, the condition cannot be satisfied and *position* is incremented by at least one. Thus, the upper bound of the time complexity is  $O(n \cdot |G|_{max})$ .  $\square$

Please note that the upper bound of the time complexity is equal to the time complexity of the naive pattern matching algorithm. This might seem as not very efficient, but we have to take in account that at first: the here proposed algorithm does the pattern matching for more complex problems then just exact matching of one string and at second: the average time complexity in typical cases (for example matching words in ASCII or Unicode files) is usually lower then  $O(n)$ . For the exact matching of one string the proposed algorithm is very similar to the Backward DAWG Matching [L92] but the BPMM approach can solve the class of problems. It is hard to set more restrictive bounds for algorithm solving the complete set of problems from 6D space. Indicative bounds for some problems are given for example in [Y79, CM94].

Two optimizations are suggested that might accelerate the algorithm. See Section 7.2.1.

For a specific subclasses of the problem space specific optimizations might exist. For a real-world application an optimization is usually needed. Such optimization can build on the general algorithm proposed by this thesis and can tune the algorithm to perform the best for a specific subset of the problem space. Duplicate processing of the same symbol is easy to avoid where the set of pattern images has a finite number of members. Another subset of problem space might be solved by a simulation of nondeterministic BPMA by a bit-parallelism as has been presented for example in [NR00, NR04, FN04]. Or by other simulation methods [HK09, H02].

### 5.3.2 Space complexity of proposed algorithm

**Theorem 5.2.** Algorithm 5.1 uses constant memory independent of the length of the text.

*Proof.* The algorithm does not contain any memory structure that can grow with the length of the text. It holds in memory only few scalar variables and the automaton. Thus the space complexity with respect to the length of text is constant, i.e.  $O(1)$ .  $\square$

The space complexity of Backward Pattern Matching Automaton that is created in the constructor does not depend on the length of text but on several other criteria (pattern matching problem, number of patterns and their length and structure etc.) The time

and space complexity of its construction and the space complexity of the BPMA is given in Chapter 6.

## 5.4 Proof of Algorithm's Correctness

**Theorem 5.3.** Let us have text  $T$ , pattern set  $P$ , set of pattern images  $G$  and backward pattern matching automaton  $M = (Q, A, \delta, q_0, F_p, F_s, F_{ps})$  such that  $L_p(M) = G^R \setminus \text{suff}^+(G^R)$ ,  $L_s(M) = \text{suff}^+(G^R) \setminus G^R$ ,  $L_{ps}(M) = G^R \cap \text{suff}^+(G^R)$ . BPMA realizes  $\text{proj}(P) = G$ . Algorithm 5.1 then outputs all positions in text  $T$  where first symbol of pattern  $p \in P$  occurs.

*Proof.* To prove Theorem 5.3 we need to prove following statements:

1. For each position  $m$  in the text the algorithm will find all patterns ending at position  $m$ . (Theorem 5.4)
2. If there is a pattern ending at position  $m$  and the algorithm's position in text is  $m - n$  then in at most  $n$  iterations the algorithm will move to position  $m$ . (Theorem 5.5)
3. Algorithm will process the complete text in finite time and then it will stop. (Theorem 5.6)

By Theorem 5.4 we ensure that the algorithm will not output more or less pattern matches if properly set to the given position. By statement 2 we ensure that the algorithm will not skip any position where some pattern occurs due to a too-long shift. By statement 2 and 3 we ensure that the algorithm will finish in finite time. Thus by combination of these we prove Theorem 5.3.  $\square$

**Theorem 5.4.** Let us have  $T, P, G$  and  $M$  from Theorem 5.3. For each position  $m$  in the text  $T$  Algorithm 5.1 finds all patterns  $p \in P$  ending at position  $m$ .

*Proof.* Let us prove that BPMA will decide whether string  $w^R$  given on its input is  $w = \text{proj}(p)$  where  $p \in P$ . This proof is easy: BPMA is able to decide if

$$\begin{aligned} & (w^R \in G^R \setminus \text{suff}^+(G^R)) \vee (w^R \in G^R \cap \text{suff}^+(G^R)) \\ & \Rightarrow w^R \in (G^R \setminus \text{suff}^+(G^R)) \cup (G^R \cap \text{suff}^+(G^R)) \\ & \Rightarrow w^R \in G^R \Rightarrow w \in G. \end{aligned}$$

Because  $G = \text{proj}(P)$  we know that by finding  $w$  we have found some pattern  $p$ .

Next, let us prove that the algorithm implements the BPMA automaton correctly. Processing of each position in text starts with the  $q$  being set to  $q_0$  (lines 5, 19) and it ends with  $\delta(q, T_{\text{position}-\text{offset}}) \neq \emptyset$  condition being false (line 7). This happens in the while loop. In the iterations  $q$  variable is updated only by  $q \leftarrow \delta(q, T_{\text{position}-\text{offset}})$  (line 8). Between these operations  $\text{offset}$  is always incremented by 1 (line 12). Algorithm thus strictly follows the transition function of BPMA and it reads symbols of text in sequence.

Output is only produced when  $q \in F_p \cup F_{ps}$  (line 11) which is in accordance with the previous paragraph. Algorithm thus implements the automaton correctly.

Because *offset* variable is incremented in each iteration and because text is preceded by the SOT (Start of Text) character, the algorithm can never end in endless-loop and the processing time to decide if  $w \in G$  is finite.  $\square$

**Theorem 5.5.** Let us have  $T, P, G$  and  $M$  from Theorem 5.3. If there is a pattern  $p \in P$  ending at position  $m$  in text  $T$  and the position of Algorithm 5.1 in text is  $m - n$ , where  $n \geq 0$ , then the algorithm will in at most  $n$  position increments move to position  $m$ .

*Proof.* Let us denote by  $u$  the longest prefix of any pattern image that ends at position  $m - n$  in text  $T$ :  $u \in A^+$ ,  $u \in \text{pref}(G)$ ,  $u = T_{m-n-|u|} \dots T_{m-n}$ . Now let us prove that for any position  $m - n$ , where  $n \geq 0$ , the following expression is true:  $|u| + n \geq |g|$ . There are 4 possible situations:

- if position  $m - n$  is further from  $m$  than is the length of pattern image  $g$ , i.e.  $n \geq |g|$ , and because  $|u|$  is always positive, i.e.  $|u| \geq 0$ , the expression  $|u| + n \geq |g|$  is always true,
- if  $n < |g|$  and  $u$  is the prefix of  $g$  occurring at position  $m$  then the expression  $|u| + n \geq |g|$  is always true because  $|u| + n = |g|$  in this case,
- if  $n < |p|$  and  $u$  is not prefix of  $g$  occurring at position  $m$  then  $u$  has to be longer than any possible prefix of  $g$  that would end at position  $m - n$ . This is proved by contradiction: If  $u$ , which is the longest prefix ending at position  $m - n$ , were shorter than the prefix of  $g$  then there would be some string  $T_{m-|g|+k} \dots T_{m-n}$  where  $k > 0$  of length  $(m - n) - (m - |g| + k) = |g| - n - k$  that is longer than the prefix of pattern image  $g$  which is  $|T_{m-|g|} \dots T_{m-n}| = (m - n) - (m - |g|) = |g| - n$ . I.e.  $|g| - n - k > |g| - n \Rightarrow k < 0$  which is contradiction with  $k > 0$ . So it implies that  $|u| + n > |g|$ .

Now let us proof that algorithm will not skip the position  $m$  by performing number of subsequent shift operations. We prove that statement by showing that *shift* will never be larger than  $n$ :

$$\begin{aligned}
 \text{shift} &= |G|_{\min} - |u| \quad \wedge \quad |u| \geq |g| - n \\
 &\Rightarrow \text{shift} \leq |G|_{\min} - (|g| - n) \\
 &\Rightarrow \text{shift} \leq |G|_{\min} - |g| + n \\
 &\Rightarrow \text{shift} \leq n + (|G|_{\min} - |g|) \quad \wedge \quad |G|_{\min} - |g| \leq 0 \\
 &\Rightarrow \text{shift} \leq n.
 \end{aligned}$$

So for any  $m - n$  position preceding position  $m$  in text the shift will be always at most  $n$ . Shift will be also always at least 1 because  $\text{shift} = \max(1, |G|_{\min} - |u|)$ . So the algorithm will in at most  $n$  steps position itself to position  $m$ .

To prove that the algorithm operates as described in this proof we can apply the same logic as we did in the proof of Theorem 5.4 to show that the algorithm will for

given position decide what is the longest prefix of pattern image ending at that position. Because it is complete analogy it is omitted here.

Shift is computed by  $shift \leftarrow \max\{1, shift_{max} - plc\}$  (line 14). Value of variable  $shift_{max}$  equals  $|G|_{min}$  and it is the input to the algorithm (see Section 5.1). Value of variable  $plc$  equals  $|u|$ . It is stored by  $plc \leftarrow tc$  (line 10) if prefix is detected by condition  $q \in F_s \cup F_{ps}$  (line 10). Variable  $tc$  counts the number of symbols read from the last shift operation. The length of each newly detected prefix rewrites the old length in variable  $plc$  until after the shift operation. Thus at the moment when there no more transitions possible ( $\delta(q, T_{position-offset}) \neq \emptyset$  is false, line 7), variable  $plc$  stores the length of the longest prefix ending at given position.  $\square$

**Theorem 5.6.** Algorithm 5.1 processes the complete text in finite time and then it stops.

*Proof.* If we add virtually at the end of text some pattern image  $g \in G$ , then Theorem 5.5 proves that the algorithm will reach the position of the last symbol of pattern  $g$  in at most  $|T| + |g|$  shift operations, i.e. in finite time. If we remove the virtually added pattern image then the algorithm will stop processing after reaching position  $|T| + 1$  (line 6) which will happen also in finite time.  $\square$

# Chapter 6

## Constructor

This chapter describes The Constructor of The Universal Backward Pattern Matching Machine described in Chapter 4. The Constructor is described for the selected implementation option of  $f = fact(G)$  function and good-prefix shift function (see Section 4.2).

The purpose of The Constructor is to take the specification of the pattern matching problem and the set of patterns and to produce a formalized description of instance of the problem in the form of extended deterministic finite automaton. The Constructor abstracts The Executor from the solved pattern matching problem by creating a formalized description of the instance of pattern matching problem in an universal way.

The inputs to The Constructor are set of patterns and specification of backward pattern matching problem. The specification is in the form of points in six-dimensional space as defined in Section 3.2. An example of the specification is SFFECO problem that represents [String matching, Full pattern, Finite number of patterns, Exact matching, only Care symbols, One string].

The output of The Constructor is formal description of the instance of the pattern matching problem. The output has the form of the extended deterministic finite automaton defined in Section 4.5.

### 6.1 The Constructor's Algorithm

The Constructor's algorithm is specified in Algorithm 6.1.

Algorithm 6.1: THE CONSTRUCTOR'S ALGORITHM

Input: Pattern matching problem  $\theta$ , set of patterns  $P$

Output: Backward Pattern Matching Automatin  $M_{BPMA}$

Method:

- 1 By Algorithm 6.13 construct Reversed Projection Automaton  $M_{RPA}$  such that  $L(M_{RPA}) = (proj_{\theta}(P))^R$
- 2 By Algorithm 6.2 construct Backward Pattern Matching Automaton  $M_{BPMA}$  from  $M_{RPA}$

The Algorithm 6.1 has two steps:

1. Given the problem  $\theta$  and the set of patterns  $P$  determine the projection of patterns  $proj(P)$  and construct nondeterministic finite automaton  $M_{RPA}$  accepting the set of all reversed images of patterns  $L(M_{RPA}) = (proj(P))^R$ . This step is described in Section 6.7.
2. Construct a minimal extended deterministic finite automaton  $M_{BPMA}$  corresponding to  $M_{RPA}$  that accepts the set of all reversed pattern images and proper suffixes of these reversed images. This step is described in Section 6.2.

**Definition 6.1** (Reversed Projection Automaton). Given pattern set  $P$  and pattern matching problem  $\theta$ , we will call *Reversed Projection Automaton* such automaton  $M$  that accepts  $L(M) = (proj_{\theta}(P))^R$ . We will denote it by  $M_{RPA}$ .

We first introduce how to construct Backward Pattern Matching Automaton from Reversed Projection Automaton. Then we show how RPA for 48 different pattern matching problems can be constructed.

## 6.2 Construction of Backward Pattern Matching Automaton

Construction of The Backward Pattern Matching Automaton from the given Reversed Projection Automaton is not dependent on the actual pattern matching problem. It is a universal approach that can be applied not only to problems studied in this thesis but also to any other pattern matching problems if there is a corresponding RPA constructed for the given problem.

The construction is described in Algorithm 6.2.

Algorithm 6.2: CONSTRUCTION OF BPMA FROM RPA

Input: Nondeterministic finite automaton  $M_{RPA}$   
 Output: Backward Pattern Matching Automatin  $M_{BPMA}$   
 Method:

- 1 By Algorithm 6.3 construct  $M'_{RPA} = (Q, A, \delta, q_0, F)$  with  $in\_deg(q_0) = 0$  equivalent to  $M_{RPA}$
- 2 By Algorithm 6.4 construct extended nondeterministic finite automaton  $M_{ENFA}$  such that  $L(M_{ENFA}) = L(M'_{RPA}) \cup suff(L(M'_{RPA}))$
- 3 By Algorithm 6.7 construct extended deterministic finite automaton  $M_{EDFA}$  equivalent to  $M_{ENFA}$
- 4 By Algorithm 6.10 construct minimal extended deterministic finite automaton  $M_{BPMA}$  equivalent to  $M_{EDFA}$

The algorithm takes as an input Reversed Projection Automaton. RPA accepts the language of all reversed pattern images and thus represents a projection function over reversed patterns. It has a form of nondeterministic finite automaton.

As a first step the algorithm transforms the  $M_{RPA}$  automaton to have initial state with zero input degree (i.e. no transitions leading to the initial state). This is to simplify the construction of  $M_{ENFA}$ .

The algorithm then constructs extended nondeterministic finite automaton  $M_{ENFA}$  (see Definition 4.1). Automaton  $M_{ENFA}$  is accepting the three languages  $L_p$ ,  $L_s$  and  $L_{ps}$  that together represent the language of all reversed pattern images  $G^R$  and of all suffixes of all reversed pattern images  $suff(G^R)$ . The reason for constructing this automaton is the use of the good prefix shift method (see Chapter 3.1 and Figure 3.7) and the fact that  $(pref(G))^R = suff(G^R)$ . If the Universal Backward Pattern Matching Machine was based on different shift function then the language accepted by the ENFA would be different.

The algorithm then constructs in two steps the minimal extended deterministic finite automaton that is equivalent to  $M_{ENFA}$ . Using the minimal deterministic automaton is the optimization for The Executors's Algorithm. Nondeterministic automaton would need to be simulated while deterministic automaton can be used as is. The steps needed to determinize the automaton present additional computational complexity and also in some cases might increase the number of states in the resulting BPMA but this construction is performed only once. It is then rewarded by efficient Executor's algorithm where the automaton is used repeatedly.

### 6.3 Construction of Automaton with Zero Input Degree of Initial State

Algorithm 6.3 constructs from given nondeterministic finite automaton  $M$  equivalent nondeterministic finite automaton  $M'$  where the initial state has zero input degree.

Algorithm 6.3: CONSTRUCTION OF EQUIVALENT NFA WITH ZERO INPUT DEGREE OF INITIAL STATE

Input: Nondeterministic finite automaton  $M = (Q, A, \delta, q_0, F)$   
 Output: Nondeterministic finite automaton  $M' = (Q', A, \delta', q'_0, F')$   
 with  $in\_deg(q'_0) = 0$

Method:

```

1    $Q' \leftarrow Q, F' \leftarrow F$ 
2    $\delta'(q, a) \leftarrow \delta(q, a)$  for all  $q \in Q, a \in A$ 
3   if  $\exists q \in Q, \exists a \in A : q_0 \in \delta(q, a)$  then
4        $Q' \leftarrow Q' \cup \{q_{00}\}, q'_0 \leftarrow q_{00}$ 
5        $\delta'(q_{00}, a) \leftarrow \delta(q_0, a)$  for all  $a \in A$ 
6   else
7        $q'_0 \leftarrow q_0$ 
8   end if
9    $M' \leftarrow (Q', A, \delta', q'_0, F')$ 
    
```

The principle of the algorithm is simple: it duplicates the initial state and all transitions leading from the initial states. The former initial state and its incoming and

outgoing transitions are not modified at all, only it is no longer marked as initial state. The duplicate state is then marked as initial state.

## 6.4 Construction of Extended Nondeterministic Finite Automaton

Let us have Reversed Projection Automaton  $M_{RPA}$  with zero input degree of initial state. Algorithm 6.4 constructs from  $M_{RPA}$  extended nondeterministic finite automaton  $M_{ENFA}$  accepting  $L_p(M_{ENFA}) = G^R$  and  $L_s(M_{ENFA}) = suff^+(G^R)$ .

Algorithm 6.4: CONSTRUCTION OF EXTENDED NFA FROM RPA  
 Input: Nondeterministic finite automaton  $M_{RPA} = (Q, A, \delta, q_0, F)$ ,  
 $Q = \{q_0, q_1, \dots, q_n\}$  with  $in\_deg(q_0) = 0$   
 Output: Extended nondeterministic finite automaton  $M_{ENFA}$   
 Method:

```

1   $Q' \leftarrow \{q_{i,p}, q_{i,s} : i \in \langle 0, n \rangle\}$ 
2  for  $\forall i, j \in \langle 0, n \rangle, \forall a \in A$  do
3      if  $\delta(q_i, a) \ni q_j$  then
4          if  $i \neq 0$  then
5               $\delta'(q_{i,p}, a) \leftarrow \delta'(q_{i,p}, a) \cup q_{j,p}$ 
6               $\delta'(q_{i,s}, a) \leftarrow \delta'(q_{i,s}, a) \cup q_{j,s}$ 
7          else
8               $\delta'(q_{0,p}, a) \leftarrow \delta'(q_{0,p}, a) \cup q_{j,p}$ 
9               $\delta'(q_{0,p}, a) \leftarrow \delta'(q_{0,p}, a) \cup \bigcup_{q_k \in Q \setminus \{q_0\}} \delta(q_k, a)$ 
10         end if
11     end if
12 end for
13  $F_p \leftarrow \{q_{i,p} : q_i \in F, i \in \langle 0, n \rangle\}$ 
14  $F_s \leftarrow \{q_{i,s} : q_i \in F, i \in \langle 0, n \rangle\}$ 
15  $F_{ps} \leftarrow \emptyset$ 
16  $M_{ENFA} \leftarrow (Q', A, \delta', q_{0,p}, F_p, F_s, F_{ps})$ 
    
```

The algorithm creates a new automaton with two sets of states recognizing two languages. The states with  $p$  index accept reversed pattern images, the states with  $s$  index accept proper suffixes of reversed pattern images. The third set of states having  $ps$  indices is empty. Resulting automaton thus looks like union of two automata: the pattern automaton and the suffix automaton. The suffix automaton is created by adding the  $\varepsilon$ -transitions from the initial state to all  $s$ -indexed states.  $\varepsilon$ -transitions are at the same time replaced by the equivalent non-epsilon transitions.

The resulting automaton  $M_{ENFA} = (Q, A, \delta, q_0, F_p, F_s, F_{ps})$  is able to decide, if the accepted word  $w$  is reversed pattern image

$$\delta(q_0, w) \ni F_p \Leftrightarrow w^R \in G$$

or reversed proper prefix of the pattern image

$$\delta(q_0, w) \ni F_s \Leftrightarrow w^R \in pref^+(G)$$

or both

$$(\delta(q_0, w) \ni F_p \wedge \delta(q_0, w) \ni F_s) \Leftrightarrow (w^R \in G \wedge w^R \in \text{pref}^+(G)).$$

Please note that the automaton  $M_{ENFA}$  constructed by Algorithm 6.4 does not produce The Backward Pattern Matching Automaton.  $M_{ENFA}$  is recognizing following two languages:  $L_p(M_{ENFA}) = G^R$ ,  $L_s(M_{ENFA}) = \text{suff}^+(G^R)$ . This violates Definition 4.1 of Backward Pattern Matching Automaton because there are strings  $w$  for which  $\delta(q_0, w) \ni F_p$  and at the same time  $\delta(q_0, w) \ni F_s$ . In BPMA for such string  $w$  it should be  $\delta(q_0, w) \ni F_{ps}$ . BPMA will be created from  $M_{ENFA}$  by determinization. In deterministic automaton  $F_{ps}$  set will not be empty and the automaton will adhere to Definition 4.1.

### 6.4.1 Example

Examples of a simple  $M_{RPA}$  for SFOECO problem and corresponding  $M_{ENFA}$  are given in Figures 6.5 and 6.6.

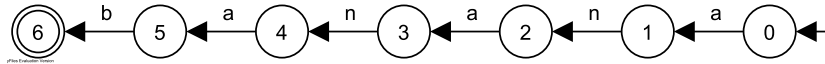


Figure 6.5: Example RPA accepting the reversed projection of pattern *banana* for SFOECO problem

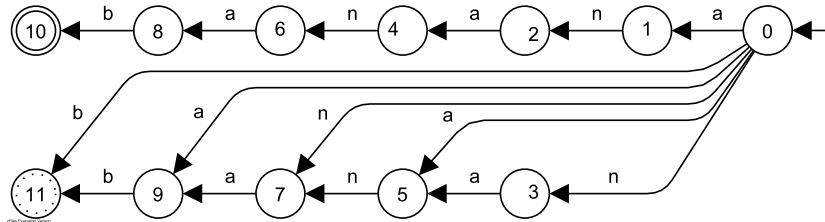


Figure 6.6: Example ENFA constructed from RPA given in Figure 6.5 by Algorithm 6.4

### 6.4.2 Time and Space complexity

The time complexity is given by the need to duplicate all transitions between states with  $p$  index to the same transitions between states with  $s$  index. This is done in  $n^2$  steps where  $n$  is the number of states of input automaton. Another  $n - 1$  steps are needed to create transitions between initial state and states with  $s$  index. The time complexity is thus  $n^2 + n - 1$  resulting in  $O(n^2)$ . Implementation of this algorithm can easily reach lower complexity if tree-traversal algorithm [K97] is used instead of examining the whole  $n \times n$  space of states.

The memory complexity is given by the size of the output automaton which is  $2n - 1$  because all states except the initial state are duplicated. This results in  $O(n)$  space complexity.

For specific subclasses of problems we could adapt existing linear algorithms for construction of suffix automata [BBH85].

## 6.5 Determinization of Extended Finite Automaton

Algorithm 6.7 is well known algorithm for construction of deterministic finite automaton that is equivalent to nondeterministic finite automaton (see [HMU01] or [C84]). The algorithm is only slightly modified on the last lines where the computation of additional sets of final states happens. This modification has no impact on the standard proof of correctness. Also, it has no impact on the time and space complexity of the algorithm.

Algorithm 6.7: CONSTRUCTION OF EDFA EQUIVALENT TO ENFA

Input: Extended nondeterministic finite automaton  
 $M_{ENFA} = (Q, A, \delta, q_0, F_p, F_s, F_{ps}), Q = \{q_0, q_1, \dots, q_n\}$   
Output: Extended deterministic finite automaton  $M_{EDFA}$  such that  
 $L(M_{EDFA}) = L(M_{ENFA})$   
Method:

```

1   $R \leftarrow \{\{q_0\}\}$ 
2   $Q', F'_p, F'_s, F'_{ps} \leftarrow \emptyset$ 
3
4  while  $R \neq \emptyset$  do
5      for  $\forall q \in R$  do
6           $R \leftarrow R \setminus \{q\}$ 
7          for  $\forall a \in A$  do
8               $q' \leftarrow \bigcup_{p \in q} \varepsilon\text{Closure}(\delta(p, a))$ 
9              if  $q' \notin Q'$  then  $R \leftarrow R \cup \{q'\}, Q' \leftarrow Q' \cup \{q'\}$ 
10             if  $F_p \cap q' \neq \emptyset$  then  $F'_p \leftarrow F'_p \cup \{q'\}$ 
11             if  $F_s \cap q' \neq \emptyset$  then  $F'_s \leftarrow F'_s \cup \{q'\}$ 
12              $\delta'(q, a) \leftarrow \delta'(q, a) \cup q'$ 
13         end
14     end
15 end
16
17  $F''_{ps} \leftarrow F'_p \cap F'_s$ 
18  $F''_p \leftarrow F'_p \setminus F''_{ps}, F''_s \leftarrow F'_s \setminus F''_{ps}$ 
19
20  $M_{ENFA} \leftarrow (Q', A, \delta', \{q_0\}, F''_p, F''_s, F''_{ps})$ 

```

### 6.5.1 Example

Examples of a simple ENFA and corresponding EDFA are given in Figures 6.8 and 6.9.

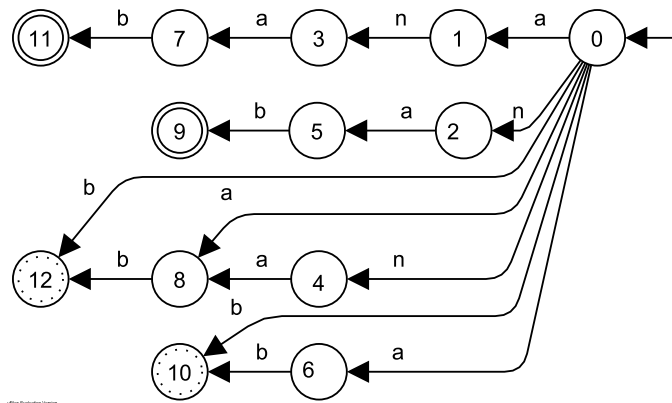


Figure 6.8: Example ENFA for pattern set  $\{bana, ban\}$  and problem SFECO

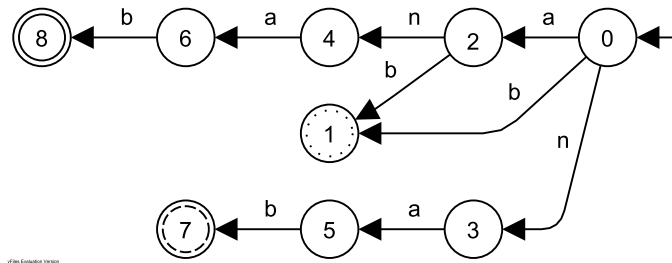


Figure 6.9: Example EDFA constructed from ENFA given in Figure 6.8 by Algorithm 6.7

### 6.5.2 Time and Space Complexity

Generally a determinization of a finite automaton can lead to exponential growth of automaton states. See [HMU01]. Yet in typical cases the deterministic automaton has a reasonable number of states. In the example shown above the resulting deterministic automaton has even less states than the nondeterministic automaton. The size of the deterministic automaton for various types of reverse projection automata is left for future studies (see Section 7.2).

## 6.6 Minimization of Extended Finite Automaton

Algorithm 6.10 is based on the algorithm described in [HMU01] or [C84]. It is based on iterative construction of partitions of set of states in order to find the equivalent states. Two states  $p, q \in Q$  of automaton  $M = (Q, A, \delta, q_0, F)$  are equivalent if for all  $w \in A^*$  is  $\delta(p, w) \in F \Leftrightarrow \delta(q, w) \in F$ .

**Remark.** We use symbol  $\mathcal{S}$  for a set of sets. It is used to distinguish set of strings  $S = \{s_1, \dots, s_n\}$  from a set of sets of strings  $\mathcal{S} = \{S_1, \dots, S_m\}$ .

In its first half, Algorithm 6.10 constructs sets of equivalence in iterative fashion (the length of  $w$  from previous paragraph is incremented by one in each iteration) until two

Algorithm 6.10: CONSTRUCTION OF MINIMAL EDFA

Input: Extended deterministic finite automaton  $M = (Q, A, \delta, q_0, F_p, F_s, F_{ps})$  where  $Q = \{q_0, q_1, \dots, q_n\}$

Output: Minimal extended deterministic finite automaton  $M_{min}$  equivalent to  $M$

Method:

```

1   $\mathbf{R}_0 \leftarrow \{F_p, F_s, F_{ps}, Q \setminus (F_p \cup F_s \cup F_{ps})\}$ 
2   $i \leftarrow 0$ 
3  do
4       $\mathbf{R}_{i+1} \leftarrow \emptyset$ 
5      for  $\forall R \in \mathbf{R}_i$  do
6           $\mathbf{R}_{i+1} \leftarrow \mathbf{R}_{i+1} \cup \{R' \in R : \text{for each } a \in A \text{ holds, that for all}$ 
7               $q \in R' \text{ exists } X \in \mathbf{R}_i \text{ such that } \delta(q, a) \subset X\}$ 
8          end
9           $i \text{ gets } i + 1$ 
10 while  $\mathbf{R}_i \neq \mathbf{R}_{i-1}$ 
11
12  $\mathbf{F}_p, \mathbf{F}_s, \mathbf{F}_{ps} \leftarrow \emptyset$ 
13 for  $\forall R \in \mathbf{R}_i, \forall a \in A$  do
14     if  $\delta(r, a) = \{r'\}, r \in R, r' \in R'$  then  $\delta'(R, a) \leftarrow \{R'\}$ 
15     if  $q_0 \in R$  then  $R_0 \leftarrow R$ 
16     if  $F_p \cap R \neq \emptyset$  then  $\mathbf{F}_p \leftarrow \mathbf{F}_p \cup \{R\}$ 
17     if  $F_s \cap R \neq \emptyset$  then  $\mathbf{F}_s \leftarrow \mathbf{F}_s \cup \{R\}$ 
18     if  $F_{ps} \cap R \neq \emptyset$  then  $\mathbf{F}_{ps} \leftarrow \mathbf{F}_{ps} \cup \{R\}$ 
19 end
20  $M_{min} \leftarrow (\mathbf{R}_i, A, \delta', R_0, \mathbf{F}_p, \mathbf{F}_s, \mathbf{F}_{ps})$ 
    
```

iterations yield the same partitioning. The second half of the algorithm then decides which partitions are final states in the output automaton and defines the transition function.

Compared to the algorithm for automaton reduction from [HMU01] or [C84] is Algorithm 6.10 modified only in the initial partitioning - instead of creating two partitions in the beginning (final and non-final states), Algorithm 6.10 creates four partitions (non-final states, final states  $F_p$  accepting  $L_p = G^R \setminus \text{suff}^+(G^R)$ , final states  $F_s$  accepting  $L = \text{suff}^+(G^R) \setminus G^R$  and final states  $F_{ps}$  accepting  $L = G^R \cap \text{suff}^+(G^R)$ ).

The above-mentioned modification is not significant for the formal proof of the correctness of the algorithm that is given for example in [HMU01]. The proof of modified algorithm is omitted here.

The minimal extended deterministic finite automaton constructed by Algorithm 6.10 is also an output of Algorithm 6.2 and is called Backward Pattern Matching Automaton. It describes the instance of pattern matching problem and it is a direct input to The Exector.

### 6.6.1 Time and Space Complexity

The modification of the algorithm has also no impact on its time and space complexity. It has been proved in [HMU01] that the upper bound of time complexity is  $O(n \cdot \log n)$  where  $n$  is the number of states of input automaton. The memory complexity is given by the queue of states but it can never be larger than the number of states of input automaton. And each state has to be processed, thus the memory complexity of the algorithm is  $O(n)$ .

### 6.6.2 Example

Examples of an extended deterministic automaton and its reduction to the minimal extended deterministic automaton are given in Figures 6.11 and 6.12.

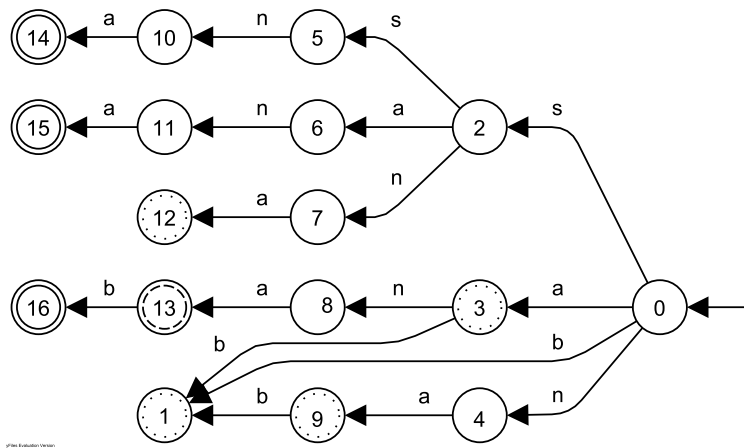


Figure 6.11: Example EDFA for the pattern set  $\{bana, anas, anss, ana\}$  and problem SFOECO

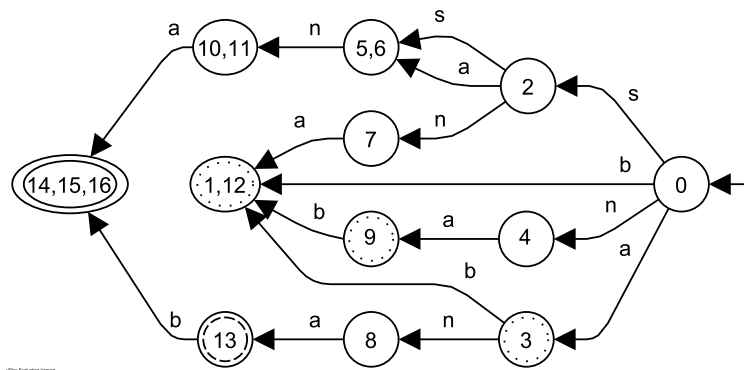


Figure 6.12: Minimal EDFA equivalent to EDFA given in Figure 6.11 constructed by Algorithm 6.10

## 6.7 Construction of Reversed Projection Automaton

This chapter describes how to construct the Reversed Projection Automaton (RPA) for the given pattern matching problem and the given pattern set. This construction is the first step of Algorithm 6.1. This RPA is used as a starting point for the following Algorithm 6.2 that constructs BPMA (Backward Pattern Matching Automaton). BPMA is then used directly in the pattern matching in The Executor's Algorithm 5.1.

The Reversed Projection Automaton is nondeterministic finite automaton accepting the language of all reversed pattern images  $L(RPA) = G^R$ . The set of pattern images  $G$  for the given set of patterns  $P$  is produced by the output of projection function  $proj_{\theta, A}(P) = G$ . In our approach we will not determine the projection function for each pattern matching problem. Instead we will construct the RPA directly by the approach described in Section 6.7.1. The value of projection function is the same as the language accepted by the automaton. Projection of set of patterns are all reversed strings that are accepted by Reversed Projection Automaton:

$$g^R \in L(RPA) \Leftrightarrow g \in proj(P).$$

### 6.7.1 Approach to the Construction of RPA

The projection automaton for specific problem is not constructed by a problem-specific algorithm. Instead we use the 6-D classification (see Section 3.2) to construct the automaton by the sequence of intermediate algorithms, each representing one move on the specific axis of 6-D space by the following approach:

1. Construct automaton for Dimension 3: One - Finite - Infinite
2. Construct automaton for Dimension 5 (Care - Don't care)
3. Construct automaton for Dimension 4: Exact - R - DIR - DIRT
4. Construct automaton for Dimension 2: Full pattern - Subpattern
5. Construct automaton for Dimension 1: seQuence - String
6. Construct automaton for Dimension 6: One - Sequence of

This approach up to step 4 is implemented in Algorithm 6.13. This thesis does not describe the algorithms for steps 5 and 6 because the reasons specified in Section 4.6.

We will start by constructing automaton for one of three points of 6-D space located on the 3<sup>rd</sup> axis: SFOECO, SF FECO, SFIECO representing pattern matching of one, finite number of and infinite number of patterns.

Then we take the resulting automaton and continue to move along the 5<sup>th</sup> axis: I.e. if the problem includes don't care symbols we will construct RPA for example for SFOEDO problem from SFOECO problem by a specific algorithm for 5<sup>th</sup> axis.

Then we continue with 4<sup>th</sup> axis etc. The complete sequence of all steps is given in Algorithm 6.13.

Algorithm 6.13: CONSTRUCTION OF REVERSED PROJECTION AUTOMATON FOR SFOECO FAMILY OF PATTERN MATCHING PROBLEMS

Input: Pattern matching problem  $\theta \in SFOECO$ , set of patterns  $P$

Output: Reversed Projection Automaton  $M_4$

Method:

```

1   if  $\theta \in SFOECO$  then construct automaton  $M_1$  from  $P$  by Algorithm 6.14
2   else if  $\theta \in SFOECO$  then construct automaton  $M_1$  from  $P$  by Algorithm 6.16
3   else construct automaton  $M_1$  from  $P$  by Algorithm from Section 6.7.4
4
5   if  $\theta \in SFOECO$  then construct automaton  $M_2$  from  $M_1$  by Algorithm 6.23
6   else  $M_2 \leftarrow M_1$ 
7
8   if  $\theta \in SFOECO$  then construct automaton  $M_3$  from  $M_2$  by Algorithm 6.27
9   else if  $\theta \in SFOECO$  then construct automaton  $M_3$  from  $M_2$  by Algorithm 6.29
10  else if  $\theta \in SFOECO$  then construct automaton  $M_3$  from  $M_2$  by Algorithm 6.31
11  else  $M_3 \leftarrow M_2$ 
12
13  if  $\theta \in SFOECO$  then construct automaton  $M_4$  from  $M_3$  by Algorithm 6.33
14  else  $M_4 \leftarrow M_3$ 

```

By this principle we can construct the Reversed Projection Automaton for all 192 pattern matching problems described by 6-D classification from [MH97] in six steps. If there is a different classification we could build RPA for problems classified by it in the same fashion.

## 6.7.2 Reversed Projection Automaton for SFOECO Problem

Algorithm 6.14 constructs Reversed Projection Automaton for problem SFOECO. It takes as an input a single pattern  $p$ . The projection function for SFOECO problem is defined as  $proj(p) = \{p^R\}$ .

The base NFA automaton  $M$  is simple one. It accepts the language  $L(M) = \{p^R\}$ . Algorithm 6.14 constructs this automaton. It is very straightforward. Also, please notice that the output of this algorithm is minimal and deterministic.

Algorithm 6.14: CONSTRUCTION OF RPA FOR SFOECO PROBLEM

Input: Pattern  $p \in A^*$ ,  $p = a_1, a_2, \dots, a_m$

Output: Deterministic finite automaton  $M$

Method:

```

1    $Q \leftarrow \{q_0, q_1, \dots, q_m\}$ 
2    $\delta(q_i, a_{m-i}) \leftarrow \{q_{i+1}\}$  for all  $i = 0, 1, \dots, m-1$ 
3    $F \leftarrow \{q_m\}$ 
4    $M \leftarrow (Q, A, \delta, q_0, F)$ 

```

### 6.7.2.1 Time and Space Complexity

The time complexity is  $O(m)$  where  $m$  is the length of pattern. The space complexity is the same. Both is given by line 2.

### 6.7.2.2 Example

Example of the Reversed Projection Automaton for pattern *banana* and SFOECO problem is given in Figure 6.15. Corresponding BPMA constructed by Algorithm 6.2 is shown in Figure 5.2. The execution of The Executor's algorithm is visualized in Figure 5.3.

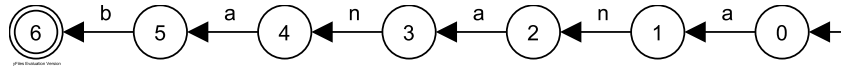


Figure 6.15: Example Reversed Projection Automaton for pattern *banana* and problem SFOECO

## 6.7.3 Reversed Projection Automaton for SFECO Problem

Algorithm 6.14 constructs Reversed Projection Automaton for problem SFECO. It takes as an input a finite set of patterns  $P$ . The projection function for SFECO problem is defined as  $proj(P) = P^R$ .

The RPA accepts the language  $L(M) = P^R$ . Algorithm 6.14 constructs this automaton. The output of this automaton is nondeterministic acyclic finite automaton that is also called trie.

Algorithm 6.16: CONSTRUCTION OF RPA FOR SFECO PROBLEM

Input: Set of patterns  $P \subset A^+$ ,  $P = \{p_1, \dots, p_n\}$ ,  $p = a_1, a_2, \dots, a_m$

Output: Nondeterministic finite automaton  $M$

Method:

```

1    $Q \leftarrow \{q_0\}$ ,  $j \leftarrow 1$ 
2   for  $\forall p \in P$  do
3      $m \leftarrow |p|$ 
4      $Q \leftarrow Q \cup \{q_{j1}, q_{j2}, \dots, q_{jm}\}$ 
5      $\delta(q_{ji}, a_{m-i}) \leftarrow \{q_{ji+1}\}$  for all  $i = 1, \dots, m$ 
6      $\delta(q_0, a_m) \leftarrow \delta(q_0, a_m) \cup \{q_{j1}\}$ 
7      $F \leftarrow F \cup \{q_{jm}\}$ 
8      $j \leftarrow j + 1$ 
9   end for
10   $M \leftarrow (Q, A, \delta, q_0, F)$ 
  
```

### 6.7.3.1 Time and Space Complexity

The time complexity is given by lines 3 and 6. It is  $\sum_{i=1}^n |p_i|$ . The size of the resulting automaton is  $1 + \sum_{i=1}^n |p_i|$  and that is also the space complexity of the algorithm.

6.7.3.2 Example

Example of the Reversed Projection Automaton for patterns *banana* and *Havana* and SFFECO problem is given in Figure 6.17. Corresponding BPMA constructed by Algorithm 6.2 is shown in Figure 6.18. The execution of The Executor’s algorithm is visualized in Figure 6.19.

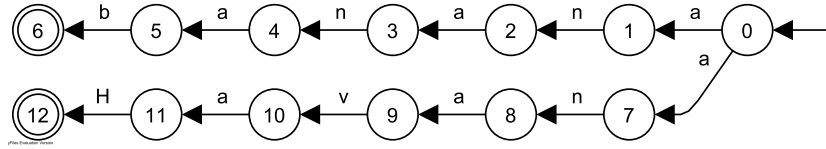


Figure 6.17: Example Reversed Projection Automaton for patterns *banana* and *Havana* and problem SFFECO

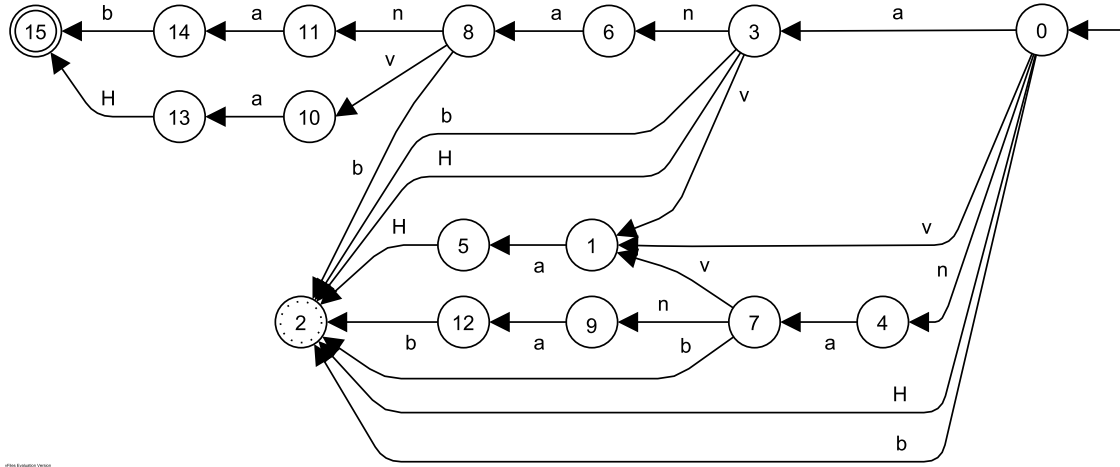


Figure 6.18: Backward Pattern Matching Automaton for RPA given in Figure 6.17 constructed by Algorithm 6.2

6.7.4 Reversed Projection Automaton for SFIECO Problem

In SFIECO pattern matching problem the input set of patterns is infinite and it is defined by the regular expression  $P = h(V)$ . The projection function for SFFECO problem is defined as  $proj(P) = P^R = (h(V))^{-1} = h(V^{-1})$ .  $V^{-1}$  is derived from  $V$  by following set of rules, where  $V_1$  and  $V_2$  are regular expressions and  $a$  is symbol:

1.  $(V_1.V_2)^{-1} \Leftrightarrow V_2^{-1}.V_1^{-1}$ ,
2.  $(V_1 + V_2)^{-1} \Leftrightarrow V_1^{-1} + V_2^{-1}$ ,
3.  $(V_1^*)^{-1} \Leftrightarrow (V_1^{-1})^*$ ,
4.  $a^{-1} \Leftrightarrow a$ ,

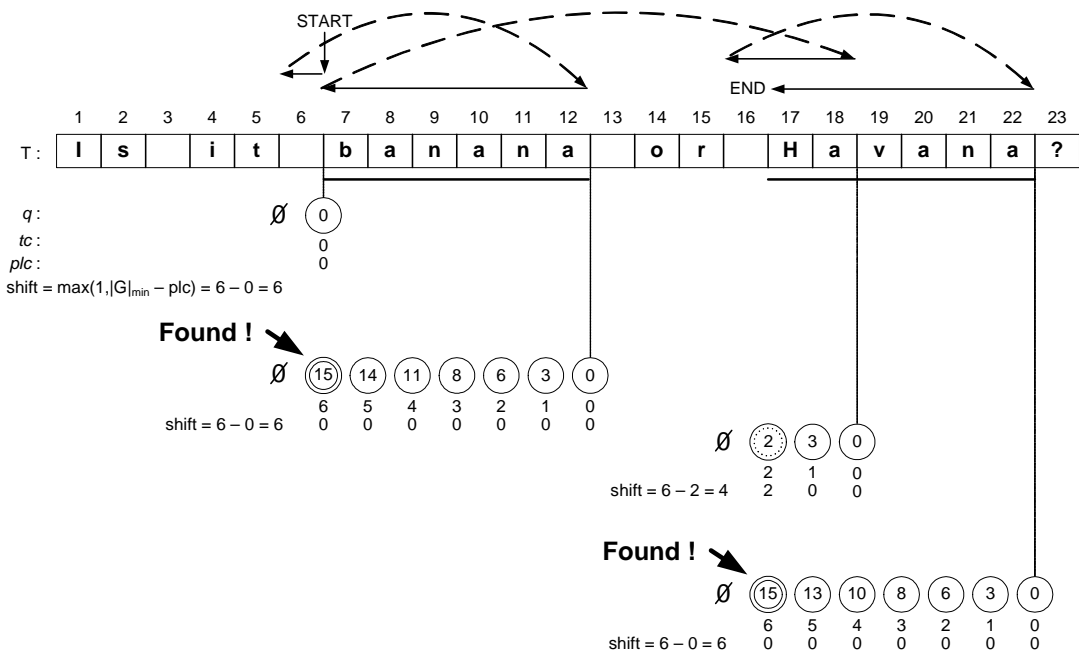


Figure 6.19: Visualization of the executor's algorithm for SFFECO problem and patterns *banana* and *Havana*

5.  $\varepsilon^{-1} \Leftrightarrow \varepsilon$ ,
6.  $\emptyset^{-1} \Leftrightarrow \emptyset$ .

We can use well known algorithms to construct automaton accepting the language specified by regular expression: see [NY60, T68] or [HMU01].

### 6.7.4.1 Example

Example of the Reversed Projection Automaton for set of patterns defined by regular expression as  $P = h(aa(bb)^*cc)$  and SFIECO problem is given in Figure 6.20. Corresponding BPMA constructed by Algorithm 6.2 is shown in Figure 6.21. The execution of the executor's algorithm is visualized in Figure 6.22.

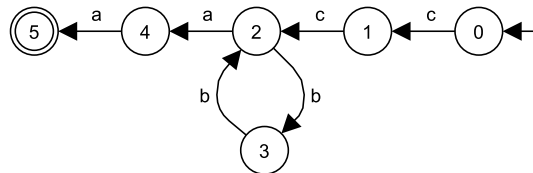


Figure 6.20: Example Reversed Projection Automaton for set of patterns  $P = h(aa(bb)^*cc)$  and problem SFIECO

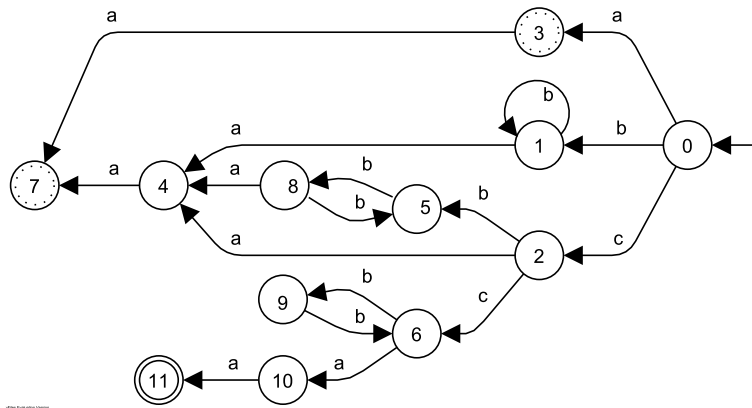


Figure 6.21: Backward Pattern Matching Automaton for RPA given in Figure 6.20 constructed by Algorithm 6.2

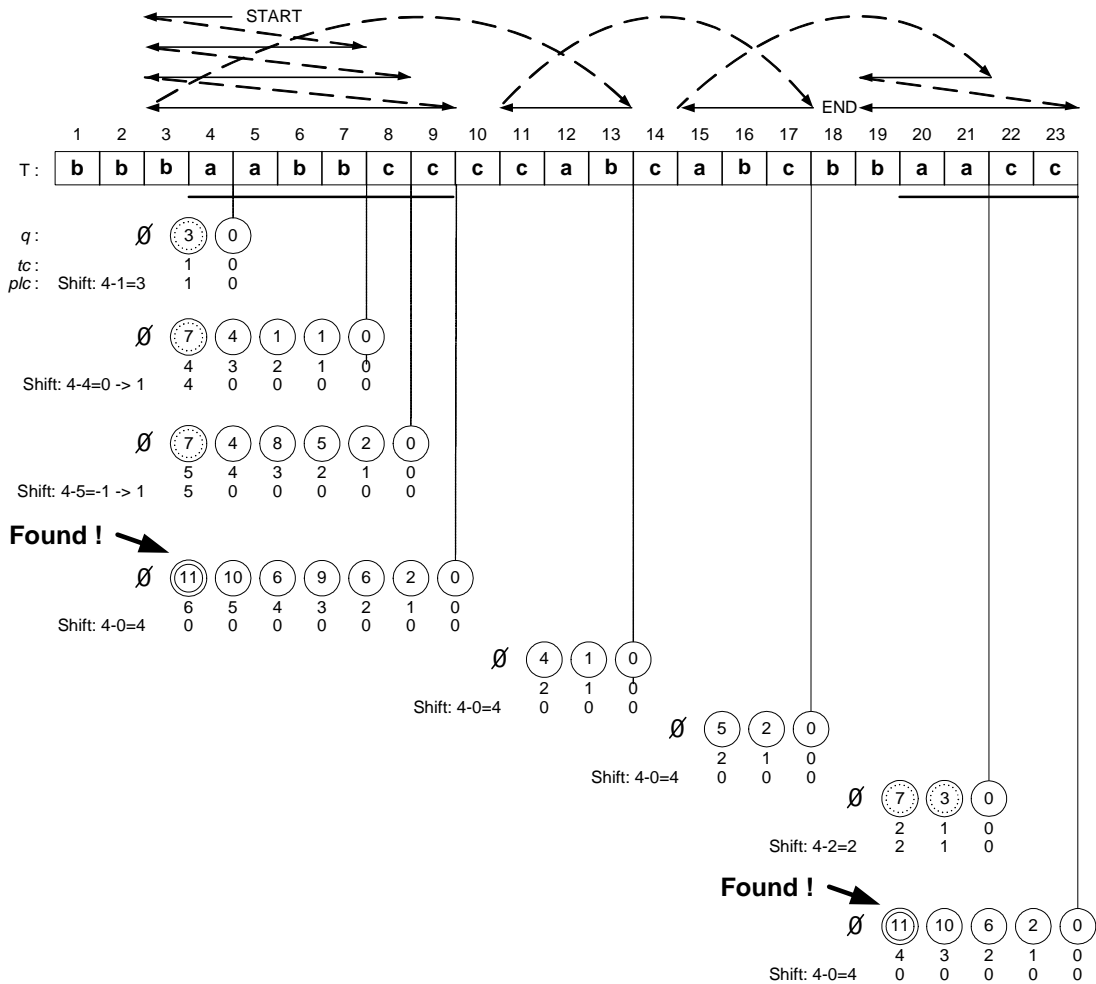


Figure 6.22: Visualization of the executor's algorithm for SFIECO problem and set of patterns  $P = h(aa(bb)^*cc)$

### 6.7.5 Reversed Projection Automaton for SF?EDO Family of Problems

This section describes how to construct the backward pattern matching automaton for the family of matching problems that involve the don't care symbols. We will use symbol  $\circ$  for the don't care symbol in the pattern. This don't care symbol means, that any symbol is considered a match. So for example pattern "boll" will match any of the following: "ball, bell, bill, bull, ...". Projection function for set of patterns  $P \subset (A \cup \{\circ\})^*$ ,  $\circ \notin A$  is  $proj(P) = \{uav : u \circ v \in P, a \in A, u, v \in A^*\}$ .

To construct RPA  $M_{SF?EDO} = (Q, A, \delta, q_0, F)$  we take as input the RPA for SF?ECO problem  $M_{SF?ECO} = (Q, A', \delta', q_0, F)$ , where  $A' = A \cup \{\circ\}$  (assuming  $\circ \notin A$ ) and we use Algorithm 6.23.

Algorithm 6.23: CONSTRUCTION OF RPA FOR SF?EDO FAMILY OF PROBLEMS  
 Input: Nondeterministic finite automaton  $M_{SF?ECO} = (Q, A \cup \{\circ\}, \delta, q_0, F)$   
 Output: Nondeterministic finite automaton  $M_{SF?EDO}$   
 Method:

- 1  $\delta'(q, a) \leftarrow \delta(q, a)$  for all  $q \in Q, a \in A$
- 2  $\delta'(q, \circ) \leftarrow \delta(q, \circ)$  for all  $q \in Q, a \in A$
- 3  $M_{SF?EDO} \leftarrow (Q, A, \delta', q_0, F)$

#### 6.7.5.1 Time and Space Complexity

If the algorithm is implemented exactly as prescribed then the time complexity will be  $2n \cdot |A|$ . But it is easy to implement the algorithm in the way that the time complexity is equal to the number of transitions.

The memory size of the output automaton is exactly the same as the input automaton thus the memory complexity is  $O(n)$ .

#### 6.7.5.2 Example

Example of the Reversed Projection Automaton for pattern  $b*ll$  and SFOECO problem (assuming  $A' = \{\circ\} \cup A$ ) is given in Figure 6.24. Reversed Projection Automaton for SFOEDO problem constructed by Algorithm 6.23 where  $\circ$  don't care symbol is replaced by  $all$  label representing  $\delta(q, all) = \bigcup_{a \in A} \delta(q, a)$  is given in Figure 6.25. Corresponding BPMA constructed by Algorithm 6.2 is shown in Figure 6.26.

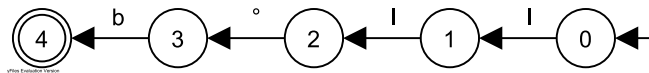


Figure 6.24: Example Reversed Projection Automaton for pattern  $b*ll$  and problem SFFECO ( $A' = \{\circ\} \cup A$ )

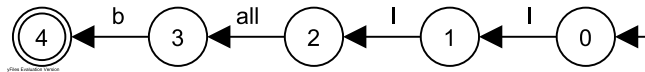


Figure 6.25: Reversed Projection Automaton constructed from RPA given in Figure 6.24 by Algorithm 6.23

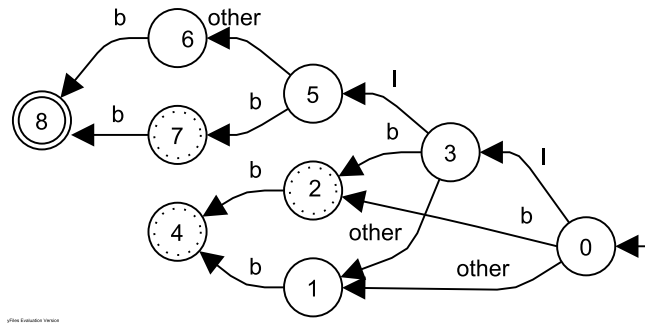


Figure 6.26: Backward Pattern Matching Automaton for RPA given in Figure 6.23 constructed by Algorithm 6.2

### 6.7.6 Reversed Projection Automaton for SF?R?O Family of Problems

This section describes how to construct the backward pattern matching automaton for the family of full string R-matching problems. R-matching means approximate matching where the operation "replace" is allowed. This kind of approximate matching was first described by Hamming in [H50]. By Hamming distance we call the distance of two patterns  $u$  and  $v$  produced by one or more replace operations.

An example of Hamming distance is:

$$D_H(\textit{banana}, \textit{Havana}) = 2.$$

The projection function for SFORCO pattern matching problem is defined as  $proj(p) = \{w : D_H(p, w) \leq k, w \in A^+\}$ , where  $k$  denotes the maximum distance between two patterns that we consider being equal (and thus representing a match in the text). In combination with projection functions defined for 3<sup>rd</sup> and 5<sup>th</sup> dimension of 6-D space we can define in similar way projection functions for SFFRCO, SFIRCO and also for SFORDO, SFFRDO and SFIRDO pattern matching problems.

To construct RPA for SF?R?O family of problems, we use Algorithms 6.27 that takes as input RPA for SF?E?O problem and constant  $k_{max}$ .

Lines 1 to 6 of Algorithm 6.27 construct union of  $k_{max} + 1$  copies of the input automaton. This new automaton has states  $q_{k,i}$  which correspond to states  $q_i$  of input automaton. Transitions between these states are copied as well as final states. Note that these copies are not "connected" to each other by any transitions.

New transitions are then added on lines 7 to 9. These transitions represent the operation replace.

Algorithm 6.27: CONSTRUCTION OF RPA FOR SF?R?O FAMILY OF PROBLEMS

Input: Nondeterministic finite automaton  $M_{SF?E?O} = (Q, A, \delta, q_0, F)$

Output: Nondeterministic finite automaton  $M_{SF?R?O}$

Method:

```

1    $Q' \leftarrow \emptyset, F' \leftarrow \emptyset$ 
2   for  $\forall k \in \langle 0, k_{max} \rangle$  do
3        $Q' \leftarrow Q' \cup \{q_{k,i} : q_i \in Q\}$ 
4        $\delta'(q_{k,i}, a) \leftarrow \delta'(q_{k,i}, a) \cup \{q_{k,j} : q_j \in \delta(q_i, a)\}$  for all  $a \in A, q_i \in Q$ 
5        $F' \leftarrow F' \cup \{q_{k,i} : q_i \in F\}$ 
6   end for
7   for  $\forall k \in \langle 0, k_{max} - 1 \rangle$  do
8        $\delta'(q_{k,i}, \bar{a}) \leftarrow \delta'(q_{k,i}, \bar{a}) \cup \{q_{k+1,j} : q_j \in \delta(q_i, a)\}$  for all  $a \in A, q_i \in Q$ 
9   end for
10   $M_{SF?R?O} \leftarrow (Q', A, \delta', q_{0,0}, F')$ 

```

### 6.7.6.1 Time and Space Complexity

Time complexity is determined by lines 4 and 8. These lines if implemented in efficient way will present the traversal of all automaton states and transitions. This traversal will happen  $2k_{max} - 1$  times (lines 1 and 7). The resulting time complexity is  $O(n)$  where  $n$  is the number of transitions.

The space complexity is given by the size of the output automaton which  $(k_{max} + 1) \cdot n$ .

### 6.7.6.2 Example

Example of the Reversed Projection Automaton for pattern *banana* and SFOECO problem is given in Figure 6.15. Reversed Projection Automaton for SFORCO problem and  $k_{max} = 1$  constructed by Algorithm 6.27 is given in Figure 6.28. Corresponding BPMA constructed by Algorithm 6.2 is shown in Figure 5.4 and the execution of The Executor's algorithm is visualized in Figure 5.5.

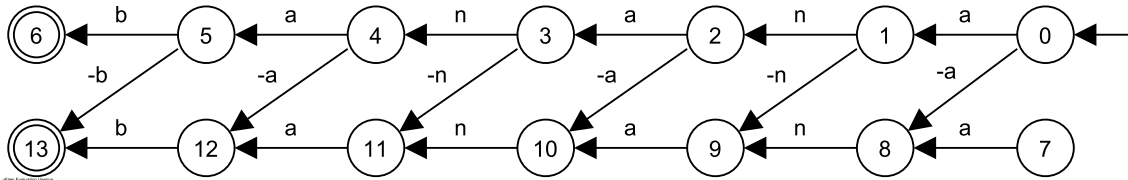


Figure 6.28: Reversed Projection Automaton constructed from RPA given in Figure 6.15 by Algorithm 6.27

### 6.7.7 Reversed Projection Automaton for SF?D?O Family of Problems

This section describes how to construct the backward pattern matching automaton for the family of full string DIR-matching problems. DIR-matching means approximate matching where the operations "replace", "delete" and "insert" are allowed. This kind of approximate matching was first described by Levenshtein in [L65]. By Levenshtein distance we call the distance of two patterns  $u$  and  $v$  produced by one or more replace, delete or edit operations.

An example of Levenshtein distance is:

$$D_L(\text{small}, \text{ball}) = 2.$$

(An explanation of distance 2 can be: symbol  $s$  was replaced by symbol  $b$  and symbol  $m$  was deleted.)

The projection function for SFODCO pattern matching problem is defined as  $proj(p) = \{w : D_L(p, w) \leq k_{max}, w \in A^+\}$ , where  $k_{max}$  denotes the maximum distance between two patterns that we consider being equal (and thus representing a match in the text). In combination with projection functions defined for 3<sup>rd</sup> and 5<sup>th</sup> dimension of 6-D space we can define in similar way projection functions for SFFDCO, SFIDCO and also for SFODDO, SFFDDO and SFIDDO pattern matching problems.

To construct RPA for SF?D?O family of problems, we use Algorithms 6.29 that takes as input RPA for SF?E?O problem and constant  $k_{max}$ .

Algorithm 6.29: CONSTRUCTION OF RPA FOR SF?D?O FAMILY OF PROBLEMS

Input: Nondeterministic finite automaton  $M_{SF?E?O} = (Q, A, \delta, q_0, F)$

Output: Nondeterministic finite automaton  $M_{SF?D?O}$

Method:

```

1    $Q' \leftarrow \emptyset, F' \leftarrow \emptyset$ 
2   for  $\forall k \in \langle 0, k_{max} \rangle$  do
3        $Q' \leftarrow Q' \cup \{q_{k,i} : q_i \in Q\}$ 
4        $\delta'(q_{k,i}, a) \leftarrow \delta'(q_{k,i}, a) \cup \{q_{k,j} : q_j \in \delta(q_i, a)\}$  for all  $a \in A, q_i \in Q$ 
5        $F' \leftarrow F' \cup \{q_{k,i} : q_i \in F\}$ 
6   end for
7
8   for  $\forall k \in \langle 0, k_{max} - 1 \rangle$  do
9        $\delta'(q_{k,i}, \bar{a}) \leftarrow \delta'(q_{k,i}, \bar{a}) \cup \{q_{k+1,j} : q_j \in \delta(q_i, a)\}$  for all  $a \in A, q_i \in Q$ 
10       $\delta'(q_{k,i}, \varepsilon) \leftarrow \delta'(q_{k,i}, \varepsilon) \cup \{q_{k+1,j} : q_j \in \delta(q_i, a)\}$  for all  $a \in A, q_i \in Q$ 
11       $\delta'(q_{k,i}, \bar{a}) \leftarrow \delta'(q_{k,i}, \bar{a}) \cup \{q_{k+1,i}\}$  for all  $a \in A, q_i \in Q$ 
                                         such that  $\delta(q_i, a) \neq \emptyset$ 
12  end for
13
14   $M_{SF?D?O} \leftarrow (Q', A, \delta', q_{0,0}, F')$ 

```

Lines 1 to 6 of Algorithm 6.29 construct union of  $k_{max}+1$  copies of the input automaton. This new automaton has states  $q_{k,i}$  which are equivalent to states  $q_i$  of input automaton.

Transitions between these states are copied as well as final states. Note that these copies are not "connected" to each other by any transitions.

New transitions are then added on lines 8 to 12. These transitions represent the operation replace (line 9), operation delete (line 10) and operation insert (line 11).

### 6.7.7.1 Time and Space Complexity

Time complexity is determined by lines 4 and 9 to 11. These lines if implemented in efficient way will present the traversal of all automaton states and transitions. This traversal will happen  $2k_{max} - 1$  times (lines 2 and 8). The resulting time complexity is  $O(n)$  where  $n$  is the number of transitions.

The space complexity is given by the size of the output automaton which  $(k_{max} + 1) \cdot n$ .

### 6.7.7.2 Example

Example of the Reversed Projection Automaton for pattern *banana* and SFOECO problem is given in Figure 6.15. Reversed Projection Automaton for SFODCO problem and  $k_{max} = 1$  constructed by Algorithm 6.29 is given in Figure 6.30.

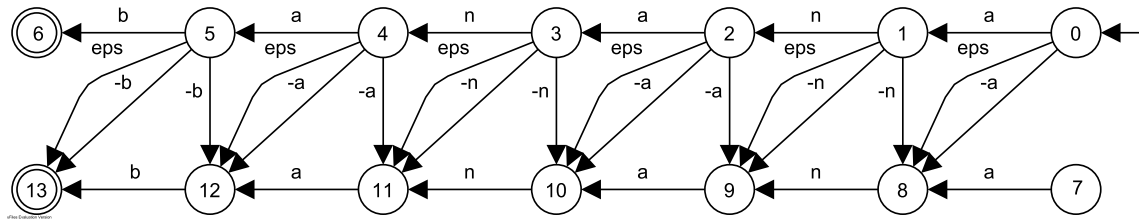


Figure 6.30: Reversed Projection Automaton constructed from RPA given in Figure 6.15 by Algorithm 6.29

## 6.7.8 Reversed Projection Automaton for SF?T?O Family of Problems

This section describes how to construct the backward pattern matching automaton for the family of full string DIRT-matching problems. DIRT-matching means approximate matching where the operations "replace", "delete", "insert" and "transpose" are allowed. This kind of approximate matching was first described by Damerau in [D66]. By Damerau distance we call the distance of two patterns  $u$  and  $v$  produced by one or more replace, delete, edit or transpose operations.

An example of Damerau distance is:

$$D_D(\textit{banana}, \textit{baanna}) = 1.$$

(An explanation of distance 1 is: the middle symbols  $na$  are transposed to  $an$ . The Levenshtein distance of these two strings would be 2 (two replacements) but it is equal to 1 in Damerau distance because transposition is counted as one operation only.)

The projection function for SFOTCO pattern matching problem is defined as  $proj(p) = \{w : D_D(p, w) \leq k_{max}, w \in A^+\}$ , where  $k$  denotes the maximum distance between two patterns that we consider being equal (and thus representing a match in the text). In combination with projection functions defined for 3<sup>rd</sup> and 5<sup>th</sup> dimension of 6-D space we can define in similar way projection functions for SFFTCO, SFITCO and also for SFOTDO, SFFTDO and SFITDO pattern matching problems.

To construct RPA for SF?T?O family of problems, we use Algorithms 6.31 that takes as input RPA for SF?E?O problem and constant  $k_{max}$ .

Algorithm 6.31: CONSTRUCTION OF RPA FOR SF?T?O FAMILY OF PROBLEMS

Input: Nondeterministic finite automaton  $M_{SF?E?O} = (Q, A, \delta, q_0, F)$

Output: Nondeterministic finite automaton  $M_{SF?T?O}$

Method:

```

1    $Q' \leftarrow \emptyset, F' \leftarrow \emptyset$ 
2   for  $\forall k \in \langle 0, k_{max} \rangle$  do
3        $Q' \leftarrow Q' \cup \{q_{k,i} : q_i \in Q\} \cup \{q_{k,i_T} : q_i \in Q\}$ 
4        $\delta'(q_{k,i}, a) \leftarrow \delta'(q_{k,i}, a) \cup \{q_{k,j} : q_j \in \delta(q_i, a)\}$  for all  $a \in A, q_i \in Q$ 
5        $F' \leftarrow F' \cup \{q_{k,i} : q_i \in F\}$ 
6   end for
7
8   for  $\forall k \in \langle 0, k_{max} - 1 \rangle$  do
9        $\delta'(q_{k,i}, \bar{a}) \leftarrow \delta'(q_{k,i}, \bar{a}) \cup \{q_{k+1,j} : q_j \in \delta(q_i, a)\}$  for all  $a \in A, q_i \in Q$ 
10       $\delta'(q_{k,i}, \varepsilon) \leftarrow \delta'(q_{k,i}, \varepsilon) \cup \{q_{k+1,j} : q_j \in \delta(q_i, a)\}$  for all  $a \in A, q_i \in Q$ 
11       $\delta'(q_{k,i}, \bar{a}) \leftarrow \delta'(q_{k,i}, \bar{a}) \cup \{q_{k+1,i}\}$  for all  $a \in A, q_i \in Q$ 
                                                such that  $\delta(q_i, a) \neq \emptyset$ 
12       $\delta'(q_{k,i}, b) \leftarrow \delta'(q_{k,i}, b) \cup \{q_{k+1,j_T} : q_j \in \delta(q_i, ab)\}$  for all  $a, b \in A, q_i \in Q$ 
13       $\delta'(q_{k+1,i_T}, a) \leftarrow \delta'(q_{k+1,i_T}, a) \cup \{q_{k+1,j} : q_j \in \delta(q_i, ab)\}$ 
                                                for all  $a, b \in A, q_i \in Q$ 
14   end for
15
16    $M_{SF?T?O} \leftarrow (Q', A, \delta', q_{0,0}, F')$ 

```

Lines 1 to 6 of Algorithm 6.31 construct union of  $k_{max}+1$  copies of the input automaton. This new automaton has states  $q_{k,i}$  which are equivalent to states  $q_i$  of input automaton. Transitions between these states are copied as well as final states. Note that these copies are not "connected" to each other by any transitions.

New transitions are then added on lines 8 to 14. These transitions represent the operation replace (line 9), operation delete (line 10), operation insert (line 11) and operation transpose (lines 12 to 13).

### 6.7.8.1 Time and Space Complexity

Time complexity is determined by lines 4 and 9 to 13. These lines if implemented in efficient way will present the traversal of all automaton states and transitions. This traversal will happen  $2k_{max} - 1$  times (lines 2 and 8). The resulting time complexity is  $O(n)$  where  $n$  is the number of transitions.

The space complexity is given by the size of the output automaton which  $(k_{max} + 1) \cdot (2n - 1)$ .

### 6.7.8.2 Example

Example of the Reversed Projection Automaton for pattern *banana* and SFOECO problem is given in Figure 6.15. Reversed Projection Automaton for SFOTCO problem and  $k_{max} = 1$  constructed by Algorithm 6.31 is given in Figure 6.32.

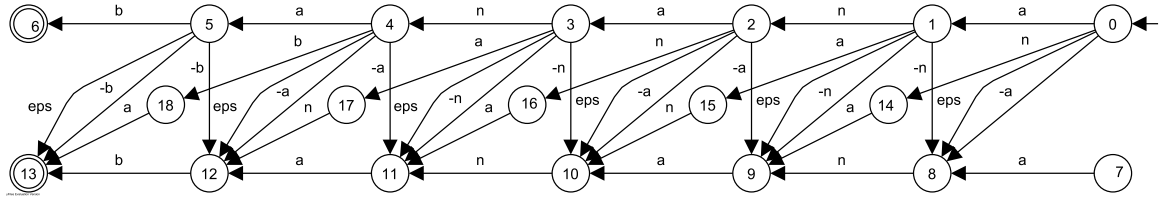


Figure 6.32: Reversed Projection Automaton constructed from RPA given in Figure 6.15 by Algorithm 6.31

### 6.7.9 SS???O Model Construction

This section describes how to construct the backward pattern matching automaton for the family of substring matching problems. The projection function for SSOECO pattern matching problem is defined as  $proj(p) = fact(p)$ . Projection function for SS???O pattern matching problems is the combination of projection functions for individual axis of 6-D space.

To construct RPA for SS???O family of problems, we use Algorithms 6.33 that takes as input RPA for SF???O problem. The algorithm assumes the input automaton to have zero input degree of initial state. If the input RPA does not satisfy this assumption we can use Algorithm 6.3 to construct equivalent automaton with zero input degree of initial state.

Algorithm 6.33: CONSTRUCTION OF RPA FOR SS???O FAMILY OF PROBLEMS

Input: Nondeterministic finite automaton  $M_{SF???O} = (Q, A, \delta, q_0, F)$ ,  
 $in\_deg(q_0) = 0$

Output: Nondeterministic finite automaton  $M_{SS???O}$

Method:

- 1  $\delta' \leftarrow \delta$
- 2 **for**  $\forall q \in Q, q \neq q_0, \forall a \in A$  **do**
- 3      $\delta'(q_0, a) \leftarrow \delta'(q_0, a) \cup \delta(q, a)$
- 4 **end**
- 5  $F' \leftarrow Q \setminus \{q_0\}$
- 6  $M_{SS???O} \leftarrow (Q, A, \delta', q_0, F')$

Algorithm 6.33 constructs a factor automaton from the input automaton. It does so by virtually adding  $\varepsilon$ -transitions from initial state to all states. Technically the algorithm is adding non- $\varepsilon$ -transitions equivalent with  $\varepsilon$ -transitions on line 3. By adding these transitions the output automaton will accept all suffixes of the language of input automaton.

Then the algorithm sets all states (except the initial state) to be final. After this modification the output automaton will also receive all prefixes of the language of suffix automaton. Together these two modifications construct factor automaton.

### 6.7.9.1 Time and Space Complexity

Time complexity is given by line 2 that presents the iteration of all transitions in the automaton. Time complexity is thus  $O(m)$  where  $m$  is number of transitions.

Space complexity of the output automaton is the same in the mean of states and doubles in the mean of transitions. We can thus set also space complexity to be  $O(m)$ .

### 6.7.9.2 Example

Example of the Reversed Projection Automaton for pattern *banana* and SFOECO problem is given in Figure 6.15. Reversed Projection Automaton for SSOECO problem constructed by Algorithm 6.33 is given in Figure 6.34.

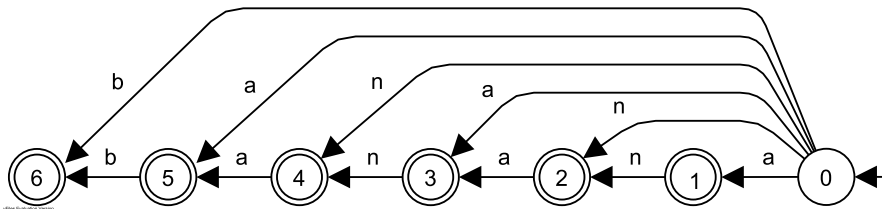


Figure 6.34: Reversed Projection Automaton constructed from RPA given in Figure 6.15 by Algorithm 6.33

# Chapter 7

## Conclusion and Future Work

### 7.1 Conclusion

This thesis proposes new approach for backward pattern matching problem called Universal Backward Pattern Matching Machine. The main innovation presented by this new approach is strict separation of description of problem instance and the pattern matching algorithm. Such separation enabled to construct new and truly universal algorithm that can solve vast number of pattern matching problems. This algorithm is presented in this thesis and the proof of its correctness is given. 48 different pattern matching problems are studied in detail and it is shown how to construct the description of problem instance for them and how these problems are solved by this new approach. Several implementation options are listed to realize the Universal Backward Pattern Matching Machine. This thesis studies in detail one implementation using good-prefix shift method.

The proposed approach does not present a performance improvement in comparison to existing algorithms but it allows to quickly derive solutions for new pattern matching problems belonging to the same class of problems. If needed, it can be optimized for specific subclasses of problems as required by a particular application.

Moreover, by using the same abstraction: the constructor and the executor and the formalized description of a problem instance, the resulting machine can describe inner working of many of existing algorithms. So the approach presented in this thesis can be also used to study existing algorithms from a new perspective, to compare them and to derive their new variants.

Some questions presented in this thesis still remain open and some optimizations of the algorithm are suggested but not studied. These leave space for future research.

### 7.2 Future Work

#### 7.2.1 Speed Optimizations

Algorithm 5.1 has the upper bound of time complexity  $O(n \cdot |G|_{max})$ . We suggest two optimizations that might speed-up the algorithm.

### 7.2.1.1 Longer Safe Shift

The longest safe shift can be longer than the one specified in Algorithm 5.1. The idea of a longer safe shift is to select the shortest pattern from the subset  $G' \subset G$  of all pattern images that start with the prefix (or prefixes) ending at the current position ( $w$  from step 3 of Section 5.1). In most cases this number can be higher than  $|G|_{min}$ . Lets compute  $G'$  based on that finding:

$$G' = \{g; g \in G, w \in pref(g)\}.$$

The longest safe shift is then

$$shift = max(1, min(|G|_{min}, |G'|_{min} - plc_{max})).$$

Such optimization would not lower the upper bound of the time complexity but it will lead to lower average time complexity.

### 7.2.1.2 Elimination of Duplicate Processing of Same Text

Very important for the algorithm's efficiency would be not to process the same parts of the text twice. Suggested optimization is to remember the text compared before the shift function is computed. If the same text is reached from the next position then instead of comparing the same text twice we would use the memory to decide whether we have detected a match and what is the longest prefix for that position.

If this optimization proves to be possible than the while loop of Algorithm 5.1 would execute at most  $n$  times because every symbol of the text will be compared at most once. That would result in very fast pattern matching algorithm with  $O(n)$  worst-case time complexity and with lowest time complexity of  $O(n/|G|_{min})$ .

### 7.2.1.3 Optimizations for Specific Subclasses

For a specific subclasses of the problem space a specific optimizations might exist. For a real-world application an optimization is usually needed. Such optimization can build on the general algorithm proposed by this thesis and can tune the algorithm to perform the best for a specific subset of the problem space.

## 7.2.2 Alternative Implementations

The Universal Backward Pattern Matching Machine presented in this thesis is based on certain selection of implementation options as described in Section 4.2. Future work should study other implementation options of  $f$  function and shift function and should compare the results with the machine described here. It is possible that with other implementation options the resulting algorithm might be faster or might have smaller memory complexity.

### 7.2.3 Average Time Complexity

The performance of the approach presented in this thesis should be compared to the existing pattern matching algorithm. Even if no universal pattern matching algorithm exists, we can still compare our universal approach to specific algorithms to see how they compare in worst-cases but especially how they compare for typical situations.

### 7.2.4 Size of Backward Pattern Matching Automaton

The pessimistic upper bound on the size of Backward Pattern Matching Automaton (BPMA) given by this thesis is  $O(2^n)$  where  $n$  is the number of states of Reversed Projection Automaton. This is due to the determinization step in the automaton construction. While this upper bound is true for very specific cases, the vast majority of Backward Pattern Matching Automata constructed will be much smaller. For example in [M96] it was shown that this bound does not take place for approximate pattern matching. Future work should study the upper bound of the BPMA size for specific pattern matching problems because great reduction in the upper bound is expected for most of the problems.

# Bibliography

- [A90] A.V. Aho: *Algorithms for Finding Patterns in Strings*. In: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, vol. A, Algorithms and Complexity, pp 255–300, Elsevier, Amsterdam, 1990.
- [AC75] A.V. Aho, M.J. Corasick: *Efficient String Matching: An Aid to Bibliographic Research*. *Communications of ACM*, Vol. 18, No. 6, pp. 333-340, 1975.
- [ACR99] C. Allauzen, M. Crochemore, M. Raffinot: *Factor Oracle: A New Structure for Pattern Matching*. In: *Proceedings of SOFSEM'99*. LNCS, vol. 1725, pp. 291–306, Springer Verlag, 1999.
- [BR90] R. A. Baeza-Yates, M. Régnier: *Fast Algorithms for Two-dimensional and Multiple Pattern Matching*. In: *Proceedings of the 2<sup>nd</sup> Scandinavian Workshop in Algorithmic Theory*, Lecture Notes in Computer Science 477, pp. 332–347, Springer-Verlag, Berlin, 1990.
- [BBH85] Blumer, A., Blumer, J., Haussler, D. et al.: *The Smallest Automaton Recognizing the Subwords of a Text*, *Theoretical Computer Science* 40, pp. 31—55, 1985.
- [BM77] R. S. Boyer, J. S. Moore: *A Fast String Searching Algorithm*. *Communications of the ACM*, Vol. 20, No. 10, pp. 762-772, October 1977.
- [CM94] W. Chang, T. Marr: *Approximate string matching and local similarity*. In *Proceedings of 5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pp. 259–273, 1994.
- [CKK72] V. Chvatal, D.A. Klarner, D.E. Knuth: *Selected Combinatorial Research Problems*. STAN-CS-72-292, Stanford University, 1972.
- [C84] M. Chytil: *Automaty a gramatiky*. Matemarický seminář SNTL Praha, 1984.
- [CW79a] B. Commentz-Walter: *A String Matching Algorithm Fast on the Average*. Technical Report 79.09.007, I.B.M. Heidelberg Scientific Center, Germany, 1979.
- [CW79b] B. Commentz-Walter: *A String Matching Algorithm Fast on the Average*. In *Proceedings of the 6th International Colloquium on Automata*,

- Languages and Programming, number 71 in Lecture Notes in Computer Science, pages 118–132, Graz, Austria, Springer-Verlag, Berlin, 1979.
- [CCG91] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, W. Rytter: *Deux méthodes pour accélérer l'algorithme de Boyer-Moore*. In: Actes des 2e Journées franco-belges: Théories des Automates et Applications, pp. 45–63. Publications de l'Université de Rouen, France, number 176, 1991.
- [CCG99] M. Crochemore, A. Czumaj, L. Gąsieniec, T. Lecroq, W. Plandowski, W. Rytter: *Fast Practical Multi-pattern Matching*. Information Processing Letters 71(3-4), pp. 107–113, 1999.
- [CHL07] M. Crochemore, C. Hancart, T. Lecroq: *Algorithms on Strings*. Cambridge University Press, 2007.
- [CWZ04] L. Cleophas, B. W. Watson, G. Zwaan: *Automaton-based sublinear keyword pattern matching*. In Proceedings of the 11th Symposium on String Processing and Information REtrieval (SPIRE), Padova, Italy, October 2004.
- [D66] F. J. Damerau: *A technique for computer detection and correction of spelling errors*. Communications of the ACM, Vol. 7, No. 3, pp. 171–176, March 1966.
- [FN04] K. Fredriksson, G. Navarro: *Average-optimal single and multiple approximate string matching*. ACM Journal of Experimental Algorithmics 9, 2004.
- [H50] R. W. Hamming: *Error-detecting and error-correcting codes*. Bell System Technical Journal 29:2, 1950, pp. 147–160.
- [H02] J. Holub: *Dynamic Programming - NFA Simulation*. Proceedings of the 7th Conference on Implementation and Application of Automata, University of Tours, Tours, France, July 2002, LNCS 2608, Springer-Verlag, pp. 295–300.
- [HK09] J. Holub, T. Kadlec: *NFA Simulation Using Deterministic State Cache*. London Algorithmics 2008, Theory and Practice, College Publications, pp. 152–166, 2009.
- [HMU01] J. E. Hopcroft, R. Motwani, J. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, 2001.
- [K56] S.C. Kleene: *Representation of Events in Nerve Nets and Finite Automata*. In Automata Studies, Princeton University Press, pp. 3–42, 1956.
- [K97] D.E. Knuth: *The Art Of Computer Programming Vol 1. 3rd ed.* Addison-Wesley, 1997.
- [KMP77] D. E. Knuth, J. H. Morris, V. R. Pratt: *Fast Pattern Matching in Strings*. SIAM J. Comput., Vol. 6, No. 2, June 1977.

- [L65] V. I. Levenshtein: *Binary codes capable of correcting deletions, insertions, and reversals*. Doklady Akademii Nauk SSSR, 163:4, 1965, pp.845-848.
- [L92] Lecroq, T.: *A variation on the Boyer-Moore algorithm*. Theoretical Computer Science, vol. 92, num. 1, pp. 119–144, 1992.
- [M96] Melichar, B.: *String Matching with  $k$  Differences by Finite Automata*. Proceedings of the 13<sup>th</sup> ICPR, Vol. II, pp. 256–260, 1996.
- [M03] B. Melichar: *Jazyky a překlady*. Vydavatelství ČVUT Praha, 2003.
- [MH97] B. Melichar, J. Holub: *6D Classification of Pattern Matching Problems*. Proceedings of the Prague Stringology Club Workshop '97, July 1997, pp. 24-32.
- [MH98] B. Melichar, J. Holub: *Pattern Matching and Finite Automata*. In Proceedings of Summer School of Information Systems and Their Applications 1998, Ruprechtov, Czech Republic, September 1998, pp. 154-183.
- [MHP05] B. Melichar, J. Holub, T. Polcar: *Text Searching Algorithms - Volume I: Forward String Matching*. <http://psc.felk.cvut.cz/athens/>, 2005.
- [M05] B. Melichar: *Text Searching Algorithms - Volume II: Backward String Matching*. <http://psc.felk.cvut.cz/athens/>, 2005.
- [NR00] G. Navarro, M. Raffinot: *Fast and Flexible String Matching by Combining Bit-Parallelism and Suffix Automata*. ACM Journal of Experimental Algorithmics 5: 4, 2000.
- [NR04] G. Navarro, M. Raffinot: *New Techniques for Regular Expression Searching*. Algorithmica 41(2), pp. 89–116, 2004.
- [NY60] R. McNaughton, H. Yamada: *Regular Expressions and State Graphs for Automata*. IEEE Trans. Electronic Computers 9:1, January 1960, pp. 39-47.
- [S09] D. Salomon: *Handbook of Data Compression*. Fifth edition, Springer, 2009.
- [T68] Thompson, K.: *Programming Techniques: Regular expression search algorithm*. C. ACM, Vol. 11, No. 6, pp. 419-422, 1968.
- [U88] N. Uratani: *A Fast Algorithm for Matching Patterns*. Internal report of IECEJ, Japan, 1988.
- [W95] B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*. Ph. D. Thesis, Eindhoven University of Technology, 1995.
- [Y79] A. C. Yao: *The complexity of pattern matching for a random string*. SIAM Journal of Computing, 8(3), pp. 368–387, 1979.