Dissertation Thesis

Simulation of Nondeterministic Finite Automata in Pattern Matching

by
Jan Holub

Submitted to
the Faculty of Electrical Engineering, Czech Technical University, Prague,
in partial fulfillment of the requirements for the degree of Doctor.

February, 2000
Prague

Thesis Supervisor:
Prof. Bořivoj Melichar
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Prague, Czech Republic

Abstract

We present various nondeterministic finite automata (*NFA*s) for the approximate string and sequence matching. *NFA*s are usually easy to construct but their equivalent deterministic finite automata (*DFA*s) can have up to $2^{|Q|}$ states, where $|Q|$ is a number of states of considered *NFA*. This is only a theoretical upper bound, which is reached in rare cases in pattern matching. We also discuss sizes of equivalent *DFA*s for the approximate string and sequence matching *NFA*s. In case, that the equivalent *DFA* cannot be used because of its enormous space requirements or the construction of the *DFA* takes too much time with respect to the length of input text, one can use a method that simulates a run of *NFA*.

We present basic simulation method that simulates a run of any general *NFA* in time $\mathcal{O}(n|Q|\lceil\frac{|Q|}{w}\rceil)$ and space $\mathcal{O}(|\Sigma||Q|\lceil\frac{|Q|}{w}\rceil)$, where n is a length of input text, Q is a set of states of *NFA*, Σ is an input alphabet, and w is bit-length of machine word. We also discuss a reduction of alphabet that can decrease the space complexity.

Then we present two other simulation techniques called dynamic programming and bit parallelism that improve the complexities of the basic simulation method. Dynamic programming uses arithmetic operations and bit parallelism uses bitwise operations. Both these algorithms were designed in order to solve directly some problems of the approximate string matching. We show in detail how these algorithms simulate *NFA*s and having this knowledge we modify them in order to solve other problems of the approximate string and sequence matching.

In that way we present directions for use of these simulation techniques for other *NFA*s. Finally, we try to formulate some rules for using these simulation methods for general *NFA*s.

Acknowledgements

First of all, I would like to express my gratitude to my thesis supervisor, prof. Bořivoj Melichar. He deserves credit for many suggestions during the course of this work. He also provided me with numerous opportunities for professional advancements. His efforts as thesis supervisor contributed substantially to the quality and completeness of the thesis.

The staff of our department has provided me a pleasant environment for my research. I would like to thank to the head of our department for taking care of my financial support. My work has been partially supported by grants 0892/97 and 0840/99 from FRVŠ agency, grant 201/98/1155 from GAČR grant agency, and internal grant 3098098/336 from CTU.

I would like also to thank to prof. Maxime Crochemore and his staff for providing me a quiet and peaceful place at his department, where I wrote some parts of this thesis. My stay at this department was partially supported by grant Socrates of EU.

Finally, my greatest thanks to my parents whose support was very important.

Dedication

To my parents

Contents

Chapter 1

Introduction

1.1 Motivation

There are many tasks in Computer Science. Some of them can be solved using finite automata. The finite automata theory is well developed formal system that is used in various areas. The original formal study of finite state systems (neural nets) is from 1943 by McCulloch and Pitts [MP43]. In 1956 Kleene [Kle56] modeled the neural nets of McCulloch and Pitts by finite automata. In that time similar models were presented by Huffman [Huf74], Moore [Moo56], and Mealy [Mea55]. In 1959, Rabin and Scott introduced nondeterministic finite automata (*NFA*s) in [RS59].

Some of pattern matching problems can be solved by finite automata. Some of algorithms solving these pattern matching problems can be then considered as simulators of *NFA*s, which can be constructed to solve these problems. If we construct *NFA*s solving other pattern matching problems, then we can use the same principle of *NFA* simulation and we get algorithms solving these other pattern matching problems.

1.2 Problem Statement

The nondeterministic finite automaton (*NFA*) accepts some input string if and only if there exists path from the initial to final state for this input string. In some states of *NFA*, there is more than one possibility of move. In such cases, the *NFA* predicts the correct move and uses it. In practice it is impossible to choose the correct way knowing just the current state and the current input symbol.

If we construct *NFA* for solving a problem, we cannot use it directly because of nondeterminism of *NFA*. One way of using this *NFA* is to transform it to the equivalent deterministic finite automaton (*DFA*) but such *DFA* can have $\mathcal{O}(2^m)$ states, where m is a number of states of *NFA*. This fact discourages from using the finite automata but in the reality the increase of states is not always exponential. We show another way of usage of *NFA*. This way is the simulation of *NFA* and it has the space complexity as well as the preprocessing time linear with respect to m. We show the simulation of *NFA* in the area of the approximate string and sequence matching.

1.3 Related work

The exact string matching has been studied from the second half of 1970's. The developed algorithms compare pattern with text in the same direction [AC75, KMP77] or in the opposite direction [BM77, CW79]. A classification of the exact string matching algorithms can be found in [Wat95].

The approximate string matching has also been studied since the second half of 1970's. It has begun by the papers [WF74, LW75] computing the edit distance between two strings. Several algorithms have been developed for the approximate string matching (See surveys [GG88, JTU96]), among them the most important are "dynamic programming" [Sel80, Ukk85] and "Shift-Or" based algorithms [BYG92, WM92]. These two algorithms have been developed in an intuitive way. We show that these two algorithms simulate *NFA*. This view make these algorithms easier to understand.

The approximate pattern matching algorithms are utilized in various areas: In text information systems it is used for example for finding misspelled words (caused by OCR processing [TKY+96]) or similar words. In computer graphics and computer vision one can describe image by Freeman chains [Fre74, Sam90] or Solomonoff codes [Sol64] and then retrieve images by content [GP] or compare images [FL96]. Other usage is in molecular biology, genetics, speech recognition, ... etc.

1.4 Contribution of the thesis

In this thesis we present *NFA*s for the exact and approximate pattern (string and sequence) matching[1] and their variants. We show upper bounds for numbers of states of *DFA*s equivalent to these *NFA*s. Then we show how to use these *NFA*s by using a deterministic simulation of their run.

We present basic simulation method that can be used for arbitrary *NFA*. Then we present uniform treatment of already existing approximate string matching algorithms [BYG92, Ukk85, WM92] by the way of simulation of *NFA*. Each of these algorithms was used for one of approximate string matching problems.

We show how these algorithms can be used for simulation of all other *NFA*s presented in this thesis (*NFA*s for the approximate string and sequence matching using the Hamming, Levenshtein, and generalized Levenshtein distance, *NFA*s for the pattern verification and the weighted approximate pattern matching, and reduced *NFA*s for these problems). Then we formulate rules for use of these simulation techniques for general *NFA*s. These simulation methods lead from the basic simulation method and they improve its space and time complexities.

1.5 Organization of the thesis

Chapter 2 introduces the basic definitions and the terminology used in this thesis. In Chapter 3 we show all the nondeterministic finite automata discussed in the thesis. We also estimate upper bounds for numbers of states of their equivalent *DFA*s. Chapter 4 presents the basic simulation method for general nondeterministic finite automata that is the base for the other simulation methods presented in this thesis. Chapter 5 describes the simulation method based on *dynamic programming*. In this section we also discuss the application of this simulation method to all the *NFA*s introduced in Chapter 3. In Chapter 6 we show next simulation method called *bit parallelism* and we also discuss its application to all the *NFA*s introduced in Chapter 3. Chapter 7 concludes the thesis with slight comparison of presented simulation methods and outlines for future work.

[1] In this thesis we consider only pattern matching problems, in which input text is not preprocessed. We also do not consider algorithms that can skip some part of input text like [BM77].

Chapter 2

Basic Definitions

In this chapter we define the basic notions that are used in this work.

Definition 2.1 (Alphabet)
An *alphabet* Σ is a finite non-empty set of *symbols*.

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

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

Definition 2.4 (Set of all strings)
The *set of all strings* over Σ is denoted Σ^*.

Definition 2.5 (Substring)
String x is a *substring* of string y, if $y = uxv$, where $x, y, u, v \in \Sigma^*$.

Definition 2.6 (Length of string)
The *length of string* w is the number of symbols in string $w \in \Sigma^*$ and is denoted $|w|$.

Definition 2.7 (Empty string)
An *empty string* is a string of length 0 and is denoted ε.

Definition 2.8 (Powerset)
For any set A the set of all subsets of A is called *powerset* of A and is denoted $\mathcal{P}(A)$.

Definition 2.9 (Replace)
Edit operation *replace* is an operation which converts string wav to string wbv, where $w, v \in \Sigma^*$, $a, b \in \Sigma$, $a \neq b$ (one symbol is replaced by another).

Definition 2.10 (Insert)
Edit operation *insert* is an operation which converts string wv to string wav, where $w, v \in \Sigma^*$, $a \in \Sigma$ (a symbol is inserted into a string).

Definition 2.11 (Delete)
Edit operation *delete* is an operation which converts string wav to string wv, where $w, v \in \Sigma^*$, $a \in \Sigma$ (a symbol is removed from a string).

Definition 2.12 (Transpose)
Edit operation *transpose* is an operation which converts $wabv$ to string $wbav$, where $w, v \in \Sigma^*$, $a, b \in \Sigma$ (two adjacent symbols are exchanged).

Definition 2.13 (Hamming distance)
Hamming distance $D_H(v, w)$ between two strings $v, w \in \Sigma^*$, $|v| = |w|$ is a minimum number of edit operations *replace* needed to convert v to w.

Definition 2.14 (Levenshtein distance)
Levenshtein distance $D_L(v, w)$ between two strings $v, w \in \Sigma^*$, is a minimum number of edit operations *replace*, *insert*, and *delete* needed to convert v to w.

Definition 2.15 (Generalized Levenshtein distance)
Generalized Levenshtein distance $D_G(v, w)$ between two strings $v, w \in \Sigma^*$, is a minimum number of edit operations *replace*, *insert*, *delete*, and *transpose* needed to convert v to w. Each symbol of v can participate at most in one edit operation *transpose*.

Definition 2.16 ((Non)overlapping occurrence)
Let $U = \{u_1, u_2, \ldots, u_m\}$ and $V = \{v_1, v_2, \ldots, v_l\}$, $U \neq V$, be two occurrences of pattern $P = p_1p_2 \ldots p_m$ in text $T = t_1t_2 \ldots t_n$ (i.e., $\forall j, 1 \leq j < m, 1 \leq u_j < u_{j+1} \leq n, 1 \leq v_j < v_{j+1} \leq n$ and $\forall j, 1 \leq j \leq m, p_j = t_{u_j}, p_j = t_{v_j}$). The occurrences U and V of the pattern P are *overlapping* if $v_1 \leq u_1 \leq v_m$ or $u_1 \leq v_1 \leq u_m$, or *non-overlapping*, otherwise.

Definition 2.17 (Approximate string matching)
Approximate string matching is defined as a searching for all the occurrences of pattern $P = p_1p_2 \ldots p_m$ in text $T = t_1t_2 \ldots t_n$ with at most k errors allowed. The number of errors allowed in a found substring is determined by the Hamming, Levenshtein, or generalized Levenshtein distances. The approximate string matching using the Hamming distance is called *approximate string matching with k mismatches* and the approximate string matching using the Levenshtein distance is called *approximate string matching with k differences*.

Remark 2.18 (Trivial Occurrence)
In this thesis we are not interested in such occurrence (e.g., with j errors) that is caused just only by appending a symbol behind the end of the previous occurrence, which has one error less ($j - 1$ errors)—appending a symbol just increases number of errors. We call such occurrence a *trivial occurrence*.

Definition 2.19 (Approximate sequence matching)
Approximate sequence matching is defined as a searching for all the occurrences of pattern $P = p_1p_2 \ldots p_m$ in text $T = t_1t_2 \ldots t_n$ with at most k errors allowed such that between symbols p_i and p_{i+1}, $0 < i < m$, in text T can be located any number of symbols. For the approximate sequence matching we can also use the Hamming, Levenshtein, or generalized Levenshtein distances.

Definition 2.20 (Weighted approximate string matching)
Weighted approximate string matching is defined as a searching for all the occurrences of pattern $P = p_1p_2 \ldots p_m$ in text $T = t_1t_2 \ldots t_n$ with cost less than or equal to given integer k. Cost is the minimum sum of weights of edit operations needed to convert P to the found string. Each edit operation has its positive integer weight, which can vary according to the symbols involved in the edit operations.

Definition 2.21 (Exact string matching)
Exact string matching is the approximate string matching where $k = 0$.

Definition 2.22 (Exact sequence matching)
Exact sequence matching is the approximate sequence matching where $k = 0$.

Definition 2.23 (Nondeterministic finite automaton)
Nondeterministic finite automaton (*NFA*) is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a set of states,

Σ is a set of input symbols, δ is a mapping $Q \times (\Sigma \cup \{\varepsilon\}) \mapsto \mathcal{P}(Q)$, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states.

Definition 2.24 (*εCLOSURE*)
Let $M = (Q, \Sigma, \delta, q_0, F)$ be an *NFA* and $P \subseteq Q$. *εCLOSURE*(P) is a set $\{q' \mid q' \in \delta(q, \varepsilon), q \in P\} \cup P$.

Definition 2.25 (Configuration of ***NFA*****)**
A *configuration of NFA* $M = (Q, \Sigma, \delta, q_0, F)$ is a pair $(q, p) \in Q \times \Sigma^*$. The *initial configuration of NFA* is a pair (q_0, p) and a *final (accepting) configuration of NFA* is a pair (q_f, ε), where $q_f \in F$.

Definition 2.26 (Move of ***NFA*****)**
A *move of NFA* $M = (Q, \Sigma, \delta, q_0, F)$ is a relation $\vdash_M \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ defined as $(q_1, aw) \vdash_M (q_2, w)$ where $q_2 \in \delta(q_1, a)$, $a \in \Sigma \cup \{\varepsilon\}$, $w \in \Sigma^*$, $q_1, q_2 \in Q$. The symbol $\vdash_M^*$ denotes a transitive and reflexive closure of relation $\vdash_M$.

Definition 2.27 (Deterministic finite automaton)
Deterministic finite automaton (*DFA*) is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, Σ is a set of input symbols, δ is a mapping $Q \times \Sigma \mapsto Q$, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states.

Definition 2.28 (Configuration of ***DFA*****)**
A *configuration of DFA* $M = (Q, \Sigma, \delta, q_0, F)$ is a pair $(q, p) \in Q \times \Sigma^*$. The *initial configuration of DFA* is a pair (q_0, p) and a *final (accepting) configuration of DFA* is a pair (q_f, ε), where $q_f \in F$.

Definition 2.29 (Move of ***DFA*****)**
A *move of DFA* $M = (Q, \Sigma, \delta, q_0, F)$ is a relation $\vdash_M \subseteq (Q \times (\Sigma^* \setminus \{\varepsilon\})) \times (Q \times \Sigma^*)$ defined as $(q_1, aw) \vdash_M (q_2, w)$, where $\delta(q_1, a) = q_2$, $a \in \Sigma$, $w \in \Sigma^*$, $q_1, q_2 \in Q$. The symbol $\vdash_M^*$ denotes a transitive and reflexive closure of relation $\vdash_M$.

Definition 2.30 (Finite automaton)
A *finite automaton* (*FA*) is *DFA* or *NFA*.

Definition 2.31 (Language accepted by ***FA*****)**
A *language accepted by FA* $M = (Q, \Sigma, \delta, q_0, F)$ is a set $L(M) = \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (q_f, \varepsilon), q_f \in F\}$.

Definition 2.32 (Equivalence of ***FA*****s)**
Two *FA*s M and M' are *equivalent* if $L(M) = L(M')$.

Definition 2.33 (Isomorphism of ***FA*****s)**
Let $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma, \delta', q_0', F')$ be *FA*s. We say that M and M' are *isomorphic* if and only if there exists a mapping $h : Q \mapsto Q'$ such that

- $h(q_0) = q_0'$,
- $\forall q \in Q, \forall a \in \Sigma, \delta'(h(q), a) = h(\delta(q, a))$,
- $\forall q \in Q, h(q) \in F'$ if and only if $q \in F$.

Definition 2.34 (Inaccessible state)
An *inaccessible state* of *FA* $M = (Q, \Sigma, \delta, q_0, F)$ is a state q such that there does not exist any string $w \in \Sigma^*$ such that $(q_0, w) \vdash_M^* (q, \varepsilon)$.

Remark 2.35
In some literature, *NFA* is defined with a set $Q_0 \subseteq Q$ of initial states instead of one initial state. If we get such *NFA*, we can transform it to an equivalent *NFA* according to Definition 2.23 using one of the following methods:

1. We insert new initial state q_0, $q_0 \notin Q$, into set Q of states (i.e., $Q \leftarrow Q \cup \{q_o\}$) and we define new transitions for this new state $\delta(q_0, \varepsilon) = Q_0$.

2. We insert new initial state q_0, $q_0 \notin Q$, into set Q of states (i.e., $Q \leftarrow Q \cup \{q_o\}$), we define new transitions for this new state $\delta(q_0, a) = \bigcup_{q \in Q_0} \delta(q, a)$, $\forall a, a \in \Sigma \cup \{\varepsilon\}$, and if $Q_0 \cap F \neq \emptyset$, we mark q_0 as a final state (i.e., $F \leftarrow \{q_0\} \cup F$). Then we find set $Q' \subseteq Q$ that contains inaccessible states (according to the new initial state q_0) and we remove these states from Q (i.e., $Q \leftarrow Q \setminus Q'$) and from F (i.e., $F \leftarrow F \setminus (F \cap Q')$).

Definition 2.36 (Active state)
An *active state* of *FA* $M = (Q, \Sigma, \delta, q_0, F)$ after reading input string $w \in \Sigma^*$ is each state q such that $(q_0, w) \vdash_M^* (q, \varepsilon)$.

Definition 2.37 (Depth of state)
A *depth of state* q in *FA* $M = (Q, \Sigma, \delta, q_0, F)$, $q \in Q$, is the minimum number of moves that are needed to get from an initial state q_0 to this state q without using ε-transitions. We denote a depth of state q by $depth(q)$.

Remark 2.38
In the figures of *NFA*s presented in this thesis, the states of the same depth are in the same column.

Definition 2.39 (Level of state)
A *level of state* q in *NFA* $M = (Q, \Sigma, \delta, q_0, F)$, $q \in Q$, is the minimum among the numbers of errors associated with all final states reachable from q. We denote a level of state q by $level(q)$.

Remark 2.40
In the figures of *NFA*s presented in this thesis, the states of the same level are in the same line (row).

Remark 2.41
Because of simplicity we label some transitions of *FA*s by set of symbols. This represents a set of transitions each labeled by one symbol of the set of symbols.

Definition 2.42 (Simulation of *NFA*)
An *algorithm* $\mathcal{A}$ *simulates a run of an NFA* $M = (Q, \Sigma, \delta, q_0, F)$, if $\forall w, w \in \Sigma^*$, it holds that $\mathcal{A}$ with given w at the input reports all information associated with each final state q_f, $q_f \in F$, after processing w, if and only if there exists a sequence of configurations $(q_0, w) \vdash_M^* (q_f, \varepsilon)$.

Chapter 3

Finite Automata

In this chapter we present all the *NFA*s that are discussed in this thesis. At the beginning we show the *NFA*s for the exact and approximate string matching. We modify these *NFA*s so that they can be used for the exact and approximate sequence matching. Then we introduce the reduced *NFA*s that can be used in the case that we do not want to know the exact number of errors in the found string but we only want to ensure that the number of errors in the found string is not greater than k. Then we present *NFA*s for string and sequence verification and for the weighted approximate string matching. We also discuss the possibility of the transformation of *NFA*s to the equivalent *DFA*s.

3.1 Nondeterministic Finite Automata

We construct *NFA*s (See Definition 2.23) searching for the first occurrence of pattern P in text T. They can also continue the searching and find all the other overlapping occurrences of the pattern in the text. We check in each step of run of the automaton whether any of the final states is active. Therefore these *NFA*s can determine:

1. whether the pattern is located in the text or not,
2. the first occurrence of the pattern in the text,
3. the number of all the occurrences of the pattern in the text,
4. all the occurrences of the pattern in the text.

It is also possible to construct *NFA*s that just only say after reading the whole input text whether the pattern is located in this text or not (Yes/No version). These *NFA*s are less general than our *NFA*s. Our *NFA*s can be simply transformed to these *NFA*s by inserting self-loops into all final states. These self-loops are labeled by all symbols of input alphabet Σ that provides that once reached the final state will still be active for the rest of the searching process. Then we can let such *NFA* run over the input text and after reading the whole text we just check the final states whether some of them is active. This check is done once at the end of searching but for our *NFA*s it is needed to check the final states after reading each symbol. The disadvantage of this *NFA* (Yes/No version) is that it continues the searching even if the result is known sooner—that could be very inefficient in the case of very large input texts.

Our *NFA*s find all the overlapping occurrences of the pattern. For example, for text $T = abbb$ and pattern $P = bb$ our *NFA*s tell that the pattern has been found twice in the text ($a\underline{bb}b$ and

---

Algorithm 3.1 (Transformation of *NFA* to the equivalent *DFA*)
Input: *NFA* $M = (Q, \Sigma, \delta, q_0, F)$
Output: *DFA* M' accepting language $L(M)$
Method: $M' = (Q', \Sigma, \delta', q_0', F')$ where Q', δ', q_0', and F' are constructed in the following way:

$Q' \leftarrow \mathcal{P}(Q)$
$\delta'(A, a) \leftarrow \bigcup_{q \in A} \varepsilon CLOSURE(\delta(q, a)), \forall a \in \Sigma, A \in Q'$
$q_0' \leftarrow \varepsilon CLOSURE(\{q_0\})$
$F' \leftarrow \{A \mid A \in Q', A \cap F \neq \emptyset\}$

$ab\underline{bb}$). It is also possible to construct the *NFA*s reporting only non-overlapping occurrences of the pattern (just $a\underline{bb}b$ in our example). But their construction is complicated and they are also less general. We can search for such non-overlapping positions using our *NFA*s in such a way that after reaching the final state we reset our *NFA*s and just only the initial state remains to be active. By performing such resetting, the *NFA* forget all the information about the found prefixes of the pattern and cannot find any occurrence of the pattern overlapping the current occurrence.

3.2 Deterministic Finite Automata

NFA cannot be directly used because of its nondeterminism. There are two ways of using it:

1. to transform it to the equivalent *DFA*,
2. to simulate the run of *NFA* (See Definition 2.42).

In this section we discuss the first way and the second one is discussed in the following chapters.

The transformation of any *NFA* to the equivalent *DFA* can be done by the standard subset construction [HU79, Koz97] as shown by Algorithm 3.1.

Algorithm 3.1 constructs the equivalent *DFA* that has always $2^{|Q|}$ states. In many cases the resulting *DFA* contains a lot of inaccessible states. Therefore we use the subset construction that does not insert the inaccessible states into the resulting *DFA*. This construction is shown in Algorithm 3.2.

The time complexity of the run of *DFA* is $\mathcal{O}(n)$ and it is dependent on the size of input text T. In order to get the total time complexity of *DFA* we have to consider also the preprocessing time, which is dependent on size of set Q' of states of *DFA* and on the size of the input alphabet Σ. The space complexity is $\mathcal{O}(|Q'||\Sigma|)$ (i.e., determined by the number of states and the size of the input alphabet Σ). It depends on implementation of mapping δ. δ can be implemented by matrix, sparse matrix, linked list, or tree. We will consider the matrix implementation further in this thesis.

3.3 Alphabet Reduction

A very important factor that influences the space complexity of the run of *FA*s is the size of the input alphabet Σ. Since the transition function δ can be represented by a transition table (matrix) of size $(|Q| \times |\Sigma|)$, we can decrease the space complexity by reducing the input alphabet Σ or by

---

Algorithm 3.2 (Transformation of *NFA* to the equivalent *DFA*)
Input: *NFA* $M = (Q, \Sigma, \delta, q_0, F)$
Output: *DFA* M' accepting language $L(M)$
Method: $M' = (Q', \Sigma, \delta', q'_0, F')$, where Q', δ', q'_0, and F' are constructed in the following way:

```
q'_0 ← εCLOSURE({q_0})
Q' ← ∅
S ← {q'_0}                                        /* set of not yet processed states */
for each q' ∈ S do                                                            /* A */
  for each a ∈ Σ do
    δ'(q', a) ← ⋃_{q∈q'} εCLOSURE(δ(q, a))
    if δ'(q', a) ∉ Q' and δ'(q', a) ∉ S then
      S ← S ∪ δ'(q', a)
    endif
  endfor
  if q' ∩ F ≠ ∅ then
    F' ← F' ∪ {q'}
  endif
  S ← S \ {q'}
  Q' ← Q' ∪ {q'}
endfor
```

---

decreasing the number of states of *FA*. The decreasing the number of states of *FA* leads to the minimization of *FA* [HU79, Wat95]. The reduction of the input alphabet is discussed below.

If some set $\mathcal{A}$, $\mathcal{A} \subseteq \Sigma$, of symbols satisfies the following condition in a *FA* $M = (Q, \Sigma, \delta, q_0, F)$

$$\forall a, b \in \mathcal{A}, a \neq b \text{ and } \forall q \in Q \text{ it holds } \delta(q, a) = \delta(q, b),$$

then we can remove all these symbols from the input alphabet Σ except one $x, x \in \mathcal{A}$, of them, which is a representative of this set. In such case we also remove all the transitions labeled by symbols $\mathcal{A} \setminus \{x\}$ from M but we have to construct a mapping $\sigma : \Sigma \mapsto \Sigma'$, $\Sigma' = \Sigma \setminus (\mathcal{A} \setminus \{x\})$, defined as follows:

$$\sigma(a) = \begin{cases} a, & \textbf{if } a \in \Sigma \setminus \mathcal{A} \\ x, & \textbf{if } a \in \mathcal{A} \end{cases} \tag{3.1}$$

There can be more than one such set $\mathcal{A}$. The reduction of the input alphabet Σ of *FA* as well as a corresponding reduction of the mapping δ is described in Algorithm 3.3, where k, $k > 0$, is the number of such sets $\mathcal{A}$.

Theorem 3.1
Let $M = (Q, \Sigma, \delta, q_0, F)$ be an *FA* and δ is represented by transition table (of size $(|Q| \times |\Sigma|)$). If

1. there exist k disjoint sets $\mathcal{A}_i \subseteq \Sigma$, $0 < k$, $1 \leq i \leq k$, such that $\forall a, b \in \mathcal{A}_i, a \neq b$ and $\forall q \in Q$ it holds $\delta(q, a) = \delta(q, b)$ and

Algorithm 3.3 (Reduction of transition function)
Input: *NFA* $M = (Q, \Sigma, \delta, q_0, F)$, $\mathcal{A}_i$, $1 \le i \le k$.
Output: Mapping σ and *NFA* $M' = (Q, \Sigma', \delta', q_0, F)$ accepting language $L(M)$ according to σ.
Method:

$\Sigma' \leftarrow \emptyset$
$S \leftarrow \Sigma$ /* set of not yet processed symbols */
for $i \leftarrow 1, 2, \dots, k$ **do**
  $x \leftarrow$ any $a \in \mathcal{A}_i$ /* choose a representative of set $\mathcal{A}_i$ */
  for each $a \in \mathcal{A}_i$ **do** /* process members of set $\mathcal{A}_i$ */
    $\sigma(a) \leftarrow x$ /* construct mapping σ */
  endfor
  $\Sigma' \leftarrow \Sigma' \cup \{x\}$
  $S \leftarrow S \setminus \mathcal{A}_i$
endfor
for each $a \in S$ **do** /* process $\forall a, a \notin \mathcal{A}_i$ for any $1 \le i \le k$ */
  $\sigma(a) \leftarrow a$
  $\Sigma' \leftarrow \Sigma' \cup \{a\}$
endfor
for each $q \in Q$ **do**
  for each $a \in \Sigma'$ **do**
    $\delta'(q, a) \leftarrow \delta(q, a)$
  endfor
endfor

2. $|Q| \times (|\bigcup_{i\leftarrow 1}^{k} \mathcal{A}_i| - k) > |\Sigma|$,

then we can reduce the space complexity of the transition table to $(|Q| \times (|\Sigma| - |\bigcup_{i\leftarrow 1}^{k} \mathcal{A}_i| + k)) + |\Sigma|$.

Proof
We construct mapping $\sigma : \Sigma \mapsto \Sigma'$, $\Sigma' \subset \Sigma$, and modified *FA* $M' = (Q, \Sigma', \delta', q_0, F)$ with reduced transition function δ', such that $\forall a \in \Sigma$ and $\forall q \in Q$ hold $\delta(q, a) = \delta'(q, \sigma(a))$.

Since instead of $|\mathcal{A}_i|$ columns in the transition table we need just one for the representative of each set $\mathcal{A}_i$, $1 \le i \le k$, the resulting transition table is of size $(|Q| \times (|\Sigma| - |\bigcup_{i\leftarrow 1}^{k} \mathcal{A}_i| + k))$. We have also to consider the implementation of mapping σ, which is at most $|\Sigma|$.

This reduction is useful only if it really reduces space requirements of transition table, that means if $|Q| \times |\Sigma| > (|Q| \times (|\Sigma| - |\bigcup_{i\leftarrow 1}^{k} \mathcal{A}_i| + k)) + |\Sigma|$. This inequality can be transformed into $|Q| \times (|\bigcup_{i\leftarrow 1}^{k} \mathcal{A}_i| - k) > |\Sigma|$. □

The reduction described above can be also applied to the case when δ is represented by sparse transition table. In such case the benefit of this reduction is hard to estimate theoretically in general since it is more dependent on the individual case. To compute the benefit one has to add up all occurrences of members of sets $\mathcal{A}_i$ in each row of the sparse table and the implementation of mapping function σ and to subtract the number of representatives of $\mathcal{A}_i$ placed to these occurrences.

The reduction of the input alphabet, which was shown for the general *FA*, can be directly used during the construction of *NFA* for the exact and approximate pattern matching. The set $\mathcal{A}$ of symbols used for labels of the same transitions is in this case the set of symbols that are not used in pattern $P = p_1p_2\ldots p_m$. Therefore the reduced input alphabet Σ' would contain just the symbols used in the pattern P and one representative ϖ of the other symbols of the input alphabet Σ (i.e., $\Sigma' = \{a \mid a \in \Sigma, \exists i, a = p_i\} \cup \{\varpi\}$).

When *FA* $M = (Q, \Sigma, \delta, q_0, F)$ for the exact and approximate string matching runs, the mapping $\sigma : \Sigma \mapsto \Sigma'$ is used. Therefore the space complexity is $\mathcal{O}(|Q| \min(|\Sigma|, m + 1) + |\Sigma|)$, when the transition function is implemented by transition table (matrix). This also reduces the space complexity of the other implementations of transitions function (sparse matrix, linked list, tree) but in all cases we must consider the space complexity of the mapping σ. It is possible to decrease the space complexity of the mapping σ by dividing it into intervals $\langle c_i^{begin}, c_i^{end}\rangle$ according to some ordering of symbols of alphabet Σ and construct modified mapping σ' as follows:

$$\sigma'(a) = \begin{cases} \sigma(a) & c_i^{begin} \le a \le c_i^{end}, \quad /*\ \text{I.: mapping} */ \\ a & c_j^{begin} \le a \le c_j^{end}, \quad /*\ \text{II.: identity} */ \\ b & c_l^{begin} \le a \le c_l^{end}, \quad /*\ \text{III.: constant} */ \end{cases} \tag{3.2}$$

$\forall a \in \Sigma$, $b \in \Sigma'$.

The interval of class II. is used for such an interval $\langle c_i^{begin}, c_i^{end}\rangle$ of mapping σ, where each value $\sigma(a)$, $c_i^{begin} \le a \le c_i^{end}$, is equal to its index (i.e., $\sigma(a) = a$). The interval of class III. is used for such an interval $\langle c_i^{begin}, c_i^{end}\rangle$ of mapping σ, where all values $\sigma(a)$, $c_i^{begin} \le a \le c_i^{end}$, of the interval are the same. The interval of class I. is used for such an interval $\langle c_i^{begin}, c_i^{end}\rangle$ of mapping σ, where the type of mapping (identity, constant) varies so that dividing it into intervals of type II. and III. would lead to an inefficient implementation.

Using this mapping, an infinite alphabet can be used as an input alphabet to *FA*s.

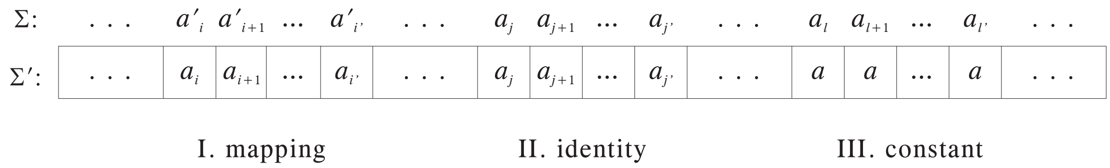


Figure 3.1: Dividing a mapping σ into intervals.

Algorithm 3.4 (Construction of *NFA* for the exact string matching)
Input: Pattern $P = p_1p_2 \dots p_m$.
Output: *NFA* M accepting language $L(M) = \{wP \mid w \in \Sigma^*\}$.
Method: *NFA* $M = (\{q_0, q_1, \dots, q_m\}, \Sigma, \delta, q_0, \{q_m\})$, where the mapping δ is constructed in the following way:

$\delta(q_0, a) \leftarrow \{q_0\}, \forall a \in \Sigma \setminus \{p_1\}$ /∗ self-loop of the initial state ∗/
$\delta(q_0, p_1) \leftarrow \{q_0, q_1\}$
for $i \leftarrow 1, 2, \dots, m-1$ **do**
  $\delta(q_i, p_{i+1}) \leftarrow \{q_{i+1}\}$ /∗ forward transitions ∗/
endfor
$\delta(q_m, a) \leftarrow \emptyset, \forall a \in \Sigma$ /∗ no transition leading from the final state ∗/

3.4 String Matching

3.4.1 Exact String Matching

In the exact string matching we search for the all occurrences of pattern $P = p_1p_2 \dots p_m$ in text $T = t_1t_2 \dots t_n$ and we do not care the string in front of string P. Therefore we construct such *NFA* M that accepts language $L(M) = \{wP \mid w \in \Sigma^*\}$. In this *NFA* each state q_i except the initial one q_0 represents that first i symbols of the pattern has been found in the text. To skip the symbols in front of an occurrence of the pattern we add a self-loop into the initial state for each symbol of input alphabet Σ. The construction of such *NFA* is shown in Algorithm 3.4 and an example of such *NFA* for $m = 4$ is shown in Figure 3.2[1].

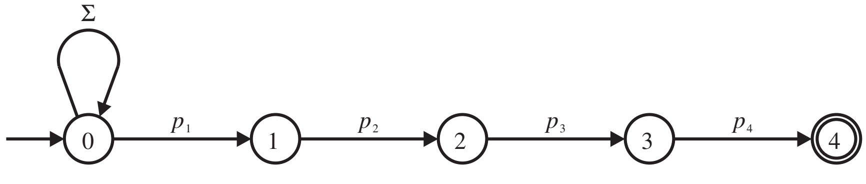


Figure 3.2: *NFA* for the exact string matching ($m = 4$).

Lemma 3.2
Minimal *NFA* for the exact string matching has $(m + 1)$ states, where m is the length of the searched pattern $P = p_1p_2 \dots p_m$.

Proof
See Algorithm 3.4.

[1] In figures 'q' symbol is omitted and states are labeled only by the indices. The initial states are denoted by the arrow leading from no state and the final states are denoted by the double circle.

Theorem 3.3
Minimal *DFA* for the exact string matching has $(m+1)$ states, where m is the length of the searched pattern $P = p_1p_2 \dots p_m$.

Proof
Let $M = (Q, \Sigma, \delta, q_0, F)$ be an *NFA* for the exact string matching constructed using Algorithm 3.4. Lemma 3.2 implies that this *NFA* has $(m+1)$ states and is minimal. Using Algorithm 3.2 we transform this *NFA* to the equivalent *DFA* $M' = (Q', \Sigma, \delta', q'_0, F')$, that has the same number of states. We prove by induction on the depth (See Definition 2.37) of state in automaton that each deterministic state q' except the final state generates one new state among the transitions leading from q'.

It holds that

$$\begin{array}{lcll} \delta(q_0, a) & = & \{q_0\}, \forall a \in \Sigma \setminus \{p_1\} & /* \text{ self-loop of the initial state } */ \\ \delta(q_0, p_1) & = & \{q_0, q_1\}. & /* \text{ matching transition } */ \end{array}$$

When the for-cycle $\mathcal{A}$ of Algorithm 3.2 is performed at first time, it holds that

$$\begin{array}{lcl} \delta'(\{q_0\}, a) & = & \{q_0\}, \forall a \in \Sigma \setminus \{p_1\} \\ \delta'(\{q_0\}, p_1) & = & \{q_0, q_1\}. \end{array}$$

Thus we obtain just one new state $q'_1 = \{q_0, q_1\} = q'_0 \cup \{q_1\}$ in the set S of not yet processed states, where $q'_0 = \{q_0\}$ is an initial state.

It holds that $\forall q_i, q_i \in Q$, $0 < i < m$,

$$\begin{array}{lcl} \delta(q_i, a) & = & \emptyset, \forall a \in \Sigma \setminus \{p_{i+1}\} \\ \delta(q_i, p_{i+1}) & = & \{q_{i+1}\}. \end{array}$$

It implies that when we determine the transitions for new state $q'_i = q'_j \cup \{q_i\}$ in the for-cycle $\mathcal{A}$ in Algorithm 3.2, where $q'_j \in Q'$ and $\forall q_k, q_k \in q'_j$, $k \leq j$:

$$\begin{array}{lcl} \delta'(q'_i, a) & = & \delta'(q'_j, a), \forall a \in \Sigma \setminus \{p_{i+1}\} \\ \delta'(q'_i, p_{i+1}) & = & \delta'(q'_j, p_{i+1}) \cup \delta(q_i, p_{i+1}) \\ & = & q'_l \cup \{q_{i+1}\}, \text{ where } \delta'(q'_j, p_{i+1}) = q'_l \end{array}$$

then there appears just one new not yet processed state $q'_{i+1} = q'_l \cup \{q_{i+1}\}$ in the transition for symbol p_{i+1}.

When determining the transitions for final state q'_m, no new state appears because for final state q_m it holds that $\delta(q_m, a) = \emptyset$, $\forall a \in \Sigma$.

The for-cycle $\mathcal{A}$ of Algorithm 3.2 is performed $(m+1)$ times. Once for the initial state (one new state is generated), $(m-1)$ times for state q'_i, $0 < i < m$, (one new state is generated in each step) and once for final state q'_m (no new state is generated).

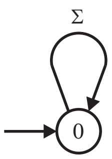
Summing up initial state $q'_0 = \{q_0\}$, new state generated by the initial state, and $(m-1)$ states generated by states q'_i, $0 < i < m$, we get $(m+1)$ states. □

The *DFA* for the exact string matching can also be constructed directly in time $\mathcal{O}(m|\Sigma|)$ as shown in Algorithm 3.5 [Cro97]. In this algorithm, state r in i-th step is the state that is reached from the initial state by reading string w (i.e., $(q_0, w) \vdash^* (r, \varepsilon)$) which is the longest prefix of $p_1p_2 \dots p_{i-1}$ and at the same time also a suffix of $p_1p_2 \dots p_i$. A process of the direct construction of *DFA* is shown in Example 3.4.

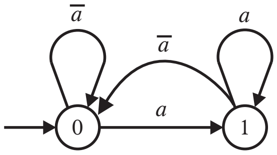
Example 3.4
Let $P = aba$. We construct *DFA* searching for an occurrence of P in an input text.

Algorithm 3.5 (Construction of *DFA* for the exact string matching)
Input: Pattern $P = p_1p_2 \dots p_m$.
Output: *DFA* M accepting language $L(M) = \{wP \mid w \in \Sigma^*\}$.
Method: *DFA* $M = (\{q_0, q_1, \dots, q_m\}, \Sigma, \delta, q_0, \{q_m\})$, where the mapping δ is constructed in the following way:

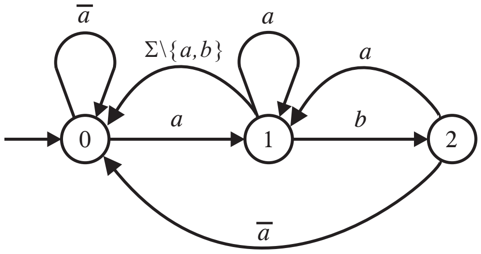
```
for each a ∈ Σ do
   δ(q0, a) ← {q0}                                  /* self-loop of the initial state */
endfor
for i ← 1, 2, ..., m do
   r ← δ(q_{i−1}, p_i)
   δ(q_{i−1}, p_i) ← q_i                             /* forward transition */
   for each a ∈ Σ do
      δ(q_i, a) ← δ(r, a)
   endfor
endfor
```



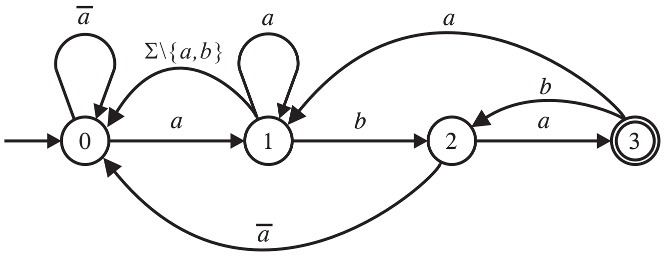
δ	a	b	$\Sigma \setminus \{a,b\}$
0	0	0	0



δ	a	b	$\Sigma \setminus \{a,b\}$
0	1	0	0
1	1	0	0



δ	a	b	$\Sigma \setminus \{a,b\}$
0	1	0	0
1	1	2	0
2	1	0	0



δ	a	b	$\Sigma \setminus \{a,b\}$
0	1	0	0
1	1	2	0
2	3	0	0
3	1	2	0

The *NFA* discussed in Section 3.1, which after reading the whole input text just says whether the pattern is located in the input text or not, can be constructed by Algorithm 3.6 and an example of such *NFA* also for $m = 4$ is shown in Figure 3.3. The final state of this *NFA* is checked just after the whole input text has been read. The number of states of such *NFA* is also given by Lemma 3.2.

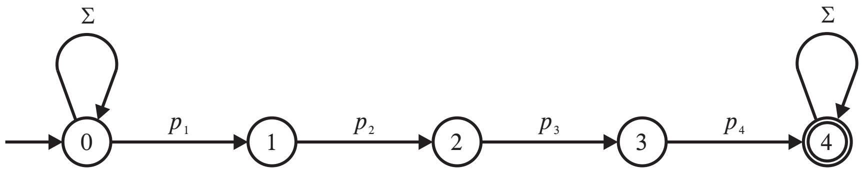


Figure 3.3: *NFA* for the exact string matching ($m = 4$, Yes/No version).

Algorithm 3.6 (Construction of *NFA* for the exact string matching—Yes/No version)
Input: Pattern $P = p_1p_2 \dots p_m$.
Output: *NFA* M accepting language $L(M) = \{wPv \mid w, v \in \Sigma^*\}$.
Method: *NFA* $M = (\{q_0, q_1, \dots, q_m\}, \Sigma, \delta, q_0, \{q_m\})$, where the mapping δ is constructed in the following way:

$\delta(q_0, a) \leftarrow \{q_0\}, \forall a \in \Sigma \setminus \{p_1\}$ /∗ self-loop of the initial state ∗/
$\delta(q_0, p_1) \leftarrow \{q_0, q_1\}$
for $i \leftarrow 1, 2, \dots, m-1$ **do**
  $\delta(q_i, p_{i+1}) \leftarrow \{q_{i+1}\}$ /∗ forward transitions ∗/
endfor
$\delta(q_m, a) \leftarrow \{q_m\}, \forall a \in \Sigma$ /∗ self-loop of the final state ∗/

Theorem 3.5
The *DFA* for the exact string matching (Yes/No version) equivalent to the *NFA* constructed by Algorithm 3.6 has $m + 1$ states, where m is the length of pattern $P = p_1p_2 \dots p_m$.

Proof
Let $M = (Q, \Sigma, \delta, q_0, F)$ be an *NFA* for the exact string matching (Yes/No version) constructed using Algorithm 3.6. Using Algorithm 3.2 we transform this *NFA* to the equivalent *DFA* $M' = (Q', \Sigma, \delta', q'_0, F')$, that has $2m$ states and then we minimize such *DFA* so that it has $m+1$ states. We prove it by induction on the depth of state in automaton. The first m steps are the same like in the proof of Theorem 3.3.

It holds that

$$\begin{array}{lcll} \delta(q_0, a) & = & \{q_0\}, \forall a \in \Sigma \setminus \{p_1\} & /* \text{ self-loop of the initial state } */ \\ \delta(q_0, p_1) & = & \{q_0, q_1\}. & /* \text{ matching transition } */ \end{array}$$

When the for-cycle $\mathcal{A}$ of Algorithm 3.2 is performed at first time, it holds that

$$\begin{array}{lcl} \delta'(\{q_0\}, a) & = & \{q_0\}, \forall a \in \Sigma \setminus \{p_1\} \\ \delta'(\{q_0\}, p_1) & = & \{q_0, q_1\}. \end{array}$$

Thus we obtain just one new state $q'_1 = \{q_0, q_1\} = q'_0 \cup \{q_1\}$ in the set S of not yet processed states, where $q'_0 = \{q_0\}$ is an initial state.

It holds that $\forall q_i, q_i \in Q, 0 < i < m$,

$$\begin{array}{lcl} \delta(q_i, a) & = & \emptyset, \forall a \in \Sigma \setminus \{p_{i+1}\} \\ \delta(q_i, p_{i+1}) & = & \{q_{i+1}\}. \end{array}$$

It implies that when we determine the transitions for new state $q'_i = q'_j \cup \{q_i\}$ in the for-cycle $\mathcal{A}$ in Algorithm 3.2, where $q'_j \in Q'$ and $\forall q_k, q_k \in q'_j, k \le j$:

$$
\begin{aligned}
\delta'(q'_i, a) &= \delta'(q'_j, a), \forall a \in \Sigma \setminus \{p_{i+1}\} \\
\delta'(q'_i, p_{i+1}) &= \delta'(q'_j, p_{i+1}) \cup \delta(q_i, p_{i+1}) \\
&= q'_l \cup \{q_{i+1}\}, \text{ where } \delta'(q'_j, p_{i+1}) = q'_l
\end{aligned}
$$

then there appears just one new not yet processed state $q'_{i+1} = q'_l \cup \{q_{i+1}\}$ in the transition for symbol p_{i+1}.

If it would hold that $\delta(q_m, a) = \emptyset$, $\forall a \in \Sigma$, (like in Theorem 3.3), no new state would appear among the transitions for state q'_m. But in Yes/No version it holds that $\delta(q_m, a) = \{q_m\}$, $\forall a \in \Sigma$. Therefore the states generated among the transitions for state q'_m differs from the already existing states in such a way that they contain in addition state q_m in their sets (i.e., $\forall q'_j \in Q', q'_j = \delta(q'_m, a), a \in \Sigma$ there exists $q'_\ell \in Q', q'_j = q'_\ell \cup \{q'_m\}$).

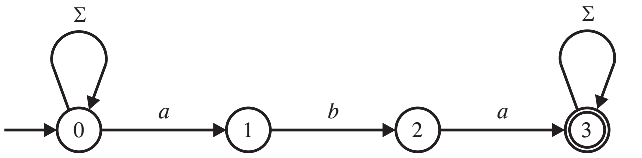
Since all such not yet processed states differs from the already existing states only in the presence of q_m in their sets, they can generate only the states that also differs from the already existing states only in the presence of q_m in their sets.

There is m states not having q_m in their sets (i.e., $q'_0, q'_1, \dots, q'_{m-1}$) and the same number of states having q_m in their sets. Since all the states having q_m in their sets are final states, they are equivalent and can be replaced by one final state. Therefore the total number of states of the resulting *DFA* is $m + 1$. □

Example 3.6

Let $P = aba$. We construct *DFA* for the exact string matching (Yes/No version) that after whole input text is read determines whether the input text contains P or not.

At the beginning we construct *NFA* using Algorithm 3.6.



δ_{NFA}	a	b	$\Sigma \setminus \{a, b\}$
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\emptyset$
q_3	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$

We transform in to the equivalent *DFA* using Algorithm 3.2.

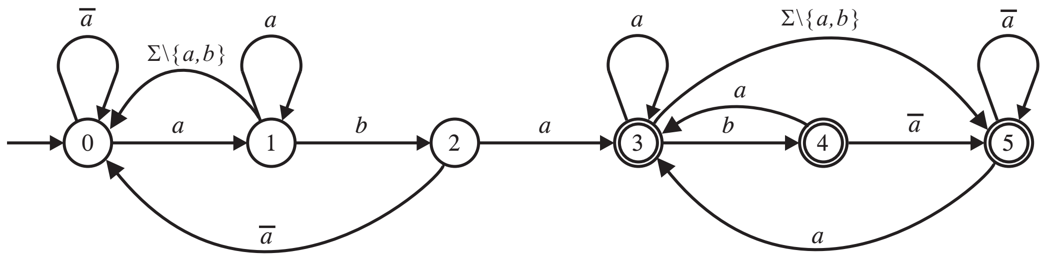
δ_{DFA}	a	b	$\Sigma \setminus \{a, b\}$
$[q_0]$	$[q_0, q_1]$	$[q_0]$	$[q_0]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_2]$	$[q_0]$
$[q_0, q_2]$	$[q_0, q_1, q_3]$	$[q_0]$	$[q_0]$
$[q_0, q_1, q_3]$	$[q_0, q_1, q_3]$	$[q_0, q_2, q_3]$	$[q_0, q_3]$
$[q_0, q_2, q_3]$	$[q_0, q_1, q_3]$	$[q_0, q_3]$	$[q_0, q_3]$
$[q_0, q_3]$	$[q_0, q_1, q_3]$	$[q_0, q_3]$	$[q_0, q_3]$

We rename the states of *DFA*.

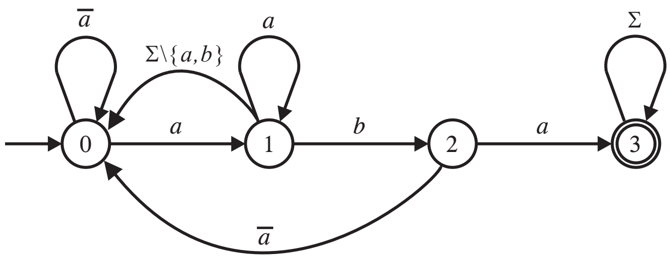
δ_{DFA}	a	b	$\Sigma \setminus \{a, b\}$
q_0	q_1	q_0	q_0
q_1	q_1	q_2	q_0
q_2	q_3	q_0	q_0
q_3	q_3	q_4	q_5
q_4	q_3	q_5	q_5
q_5	q_3	q_5	q_5

Algorithm 3.7 (Construction of *DFA* for the exact string matching—Yes/No version)
Input: Pattern $P = p_1p_2 \dots p_m$.
Output: *DFA* M accepting language $L(M) = \{wPv \mid w, v \in \Sigma^*\}$.
Method: *DFA* $M = (\{q_0, q_1, \dots, q_m\}, \Sigma, \delta, q_0, \{q_m\})$, where the mapping δ is constructed in the following way:

```
for each a ∈ Σ do
   δ(q_0, a) ← {q_0}                         /* self-loop of the initial state */
endfor
for i ← 1, 2, ..., m do
   r ← δ(q_{i-1}, p_i)
   δ(q_{i-1}, p_i) ← q_i                     /* forward transition */
   if i < m then
      δ(q_i, a) ← δ(r, a), ∀a ∈ Σ
   else
      δ(q_m, a) ← q_m, ∀a ∈ Σ
   endif
endfor
```



Then we minimize the resulting *DFA*.



δ_{DFA}	a	b	$\Sigma \setminus \{a, b\}$
q_0	q_1	q_0	q_0
q_1	q_1	q_2	q_0
q_2	q_3	q_0	q_0
q_3	q_3	q_3	q_3

The *DFA* for the exact string matching (Yes/No version) can also be constructed directly in time $\mathcal{O}(m|\Sigma|)$ as shown in Algorithm 3.7.

We do not consider Yes/No versions of *FA*s further in this thesis since they are less general than the considered versions of *FA*s. Moreover, the *FA*s considered in this thesis can be transformed to these Yes/No versions. In case of *NFA*s, one should only add self-loop labeled by Σ to each final state. In case of *DFA*s, the situation is little bit complicated since one has to ensure determinism as well as to provide possible transitions into final states with lower number of errors.

3.4.2 Approximate String Matching Using Hamming Distance

In the approximate string matching using the Hamming distance for pattern $P = p_1 p_2 \dots p_m$ we have to consider maximum number k of errors allowed. Therefore we use $k+1$ copies of *NFA* for the exact string matching $(M_0, M_1, \dots, M_k)$—one for 'no error' (level 0), one for 'one error' (level 1), ..., one for 'k errors' (level k). These $k+1$ automata are connected by the transitions representing edit operation *replace*. Each such transition is labeled by symbol $\overline{p}_{j+1}$ (mismatching symbol p_{j+1} in pattern P) and leads from state q_j of automaton M_i to state q_{j+1} of automaton M_{i+1}, $0 \leq i < k$, $0 \leq j < m$—the depth in the automaton (corresponding to a pointer in the pattern) is increased as well as the minimum number of errors. The initial state of such automaton is the initial state of automaton M_0. Inaccessible states of the resulting automaton are removed.

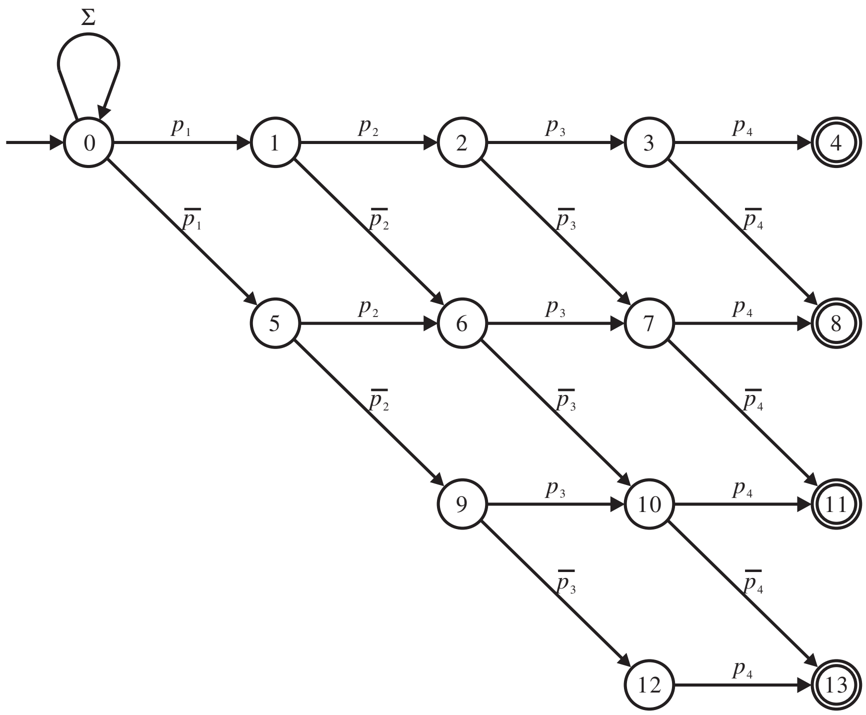


Figure 3.4: *NFA* for the approximate string matching using the Hamming distance—$\overline{p}$ version ($m = 4$, $k = 3$).

This automaton can be constructed by connecting $k+1$ copies of *NFA* as described above or directly as shown by Algorithm 3.8[2]. An example of such *NFA* for $m = 4$ and $k = 3$ is shown in Figure 3.4. The resulting *NFA* was also presented in [Dar92] and [Mel95].

There is also another possibility for representing *replace* transition. We can label *replace* transition by Σ (instead of $\overline{p}_i$). This *NFA* can be also constructed by connecting $k+1$ copies of *NFA* or directly as shown by Algorithm 3.9. This algorithm is similar to Algorithm 3.8 but the *replace* transitions are modified (*replace* transition can be used also for matching symbol). An example of such *NFA* for $m = 4$ and $k = 3$ is shown in Figure 3.5.

[2]It is possible to identify states of *NFA* by pairs (*depth*, *level*). The construction of such *NFA* is straightforward and therefore we present algorithms that identify the states by numbers $0, 1, \dots, |Q|-1$. This way is more practical for direct implementation.

Algorithm 3.8 (Construction of *NFA* for the approximate string matching using the Hamming distance—$\overline{p}$ version)
Input: Pattern $P = p_1p_2\dots p_m$, maximum number of errors allowed k, $k < m$.
Output: *NFA* M accepting language $L(M) = \{wu \mid w, u \in \Sigma^*,\ D_H(P, u) \leq k\}$.
Method: *NFA* $M = (Q, \Sigma, \delta, q_0, F)$, where Q, δ, and F are constructed in the following way:

$Q \leftarrow \{q_0, q_1, \dots, q_{|Q|-1}\}$, $|Q| = (k+1)(m+1-\frac{k}{2})$
$l \leftarrow 0$ /∗ number of level ∗/
$r \leftarrow 1$ /∗ order of symbol in P (= depth of state) ∗/
for $i \leftarrow 0, 1, \dots, |Q|-1$ **do**
  if $r > m$ **then**
    $\delta(q_i, a) \leftarrow \emptyset$, $\forall a \in \Sigma$ /∗ final state ∗/
    $F \leftarrow F \cup \{q_i\}$
    $l \leftarrow l+1$; $r \leftarrow l+1$ /∗ go to the first state of next level ∗/
  else
    $\delta(q_i, p_r) \leftarrow \{q_{i+1}\}$ /∗ matching transition ∗/
    if $l < k$ **then**
      $s \leftarrow i + m + 1 - l$ /∗ s = number of right lower neighbor ∗/
      $\delta(q_i, a) \leftarrow \{q_s\}$, $\forall a \in \Sigma \setminus \{p_r\}$ /∗ *replace* transition ∗/
    endif
    $r \leftarrow r+1$ /∗ go to the next depth of level l ∗/
  endif
endfor
$\delta(q_0, a) \leftarrow \delta(q_0, a) \cup \{q_0\}$, $\forall a \in \Sigma$ /∗ self-loop of the initial state ∗/

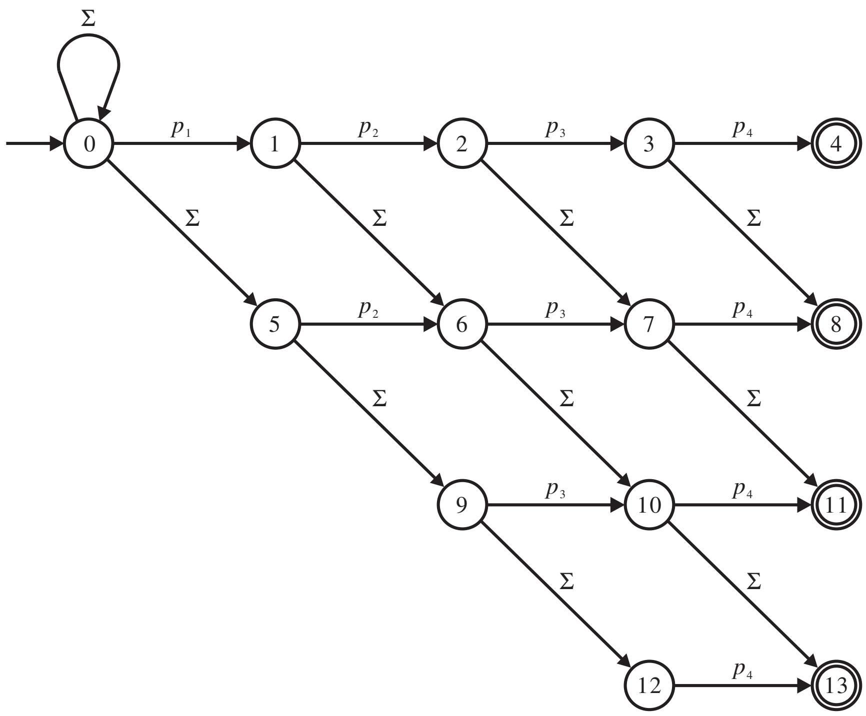


Figure 3.5: *NFA* for the approximate string matching using the Hamming distance—Σ version ($m = 4$, $k = 3$).

Lemma 3.7
The *NFA*s for the approximate string matching using the Hamming distance constructed by Algorithm 3.8 and Algorithm 3.9 for a pattern of length m and maximum number of errors k have $(k+1)(m+1-\frac{k}{2})$ states.

Proof
In Algorithm 3.8 as well as in Algorithm 3.9, $k+1$ copies of the *NFA* for the exact string matching (constructed by Algorithm 3.4) are built and in each i-th copy, $0 \leq i \leq k$, first i states are removed.

Therefore the number of states is $(m+1)+(m+1-1)+(m+1-2)+\cdots = \sum_{i=0}^{k}(m+1-i) = (k+1)(m+1-\frac{k}{2})$. □

Lemma 3.8
If *NFA* $M = (Q, \Sigma, \delta, q_0, F)$ for the approximate string matching using the Hamming distance ($\overline{p}$ version) constructed by Algorithm 3.8 reaches a state q (i.e., q is active) at depth j, $0 \leq j \leq m$, and level ℓ, $0 \leq \ell \leq k$, $\ell \leq j$, then there is no other active state at depth j.

Proof
The proof is given by induction on the depth in *NFA*.

Lemma 3.8 holds for depth 0, since there is only one initial state q_0.

Assume that Lemma 3.8 holds for depth j, $0 < j < m$, in i-th step of run of *NFA* (after i-th input symbol is read; $j \leq i$). Let q' be the only active state at depth j and let its level be ℓ. There is one matching transition leading from q' labeled by symbol p_{j+1} and if $\ell \leq j$ and $\ell \leq k$, there is also one *replace* transition labeled by $\Sigma \setminus \{p_{j+1}\}$. Both transitions lead to depth $j+1$. Since

Algorithm 3.9 (Construction of *NFA* for the approximate string matching using the Hamming distance—Σ version)
Input: Pattern $P = p_1 p_2 \dots p_m$, maximum number of errors allowed k, $k < m$.
Output: *NFA* M accepting language $L(M) = \{wu \mid w, u \in \Sigma^*, D_H(P, u) \leq k\}$.
Method: *NFA* $M = (Q, \Sigma, \delta, q_0, F)$, where Q, δ, and F are constructed in the following way:

$Q \leftarrow \{q_0, q_1, \dots, q_{|Q|-1}\}$, $|Q| = (k+1)(m+1-\frac{k}{2})$
$l \leftarrow 0$ /* number of level */
$r \leftarrow 1$ /* order of symbol in P (= depth of state) */
for $i \leftarrow 0, 1, \dots, |Q|-1$ **do**
  if $r > m$ **then**
    $\delta(q_i, a) \leftarrow \emptyset$, $\forall a \in \Sigma$ /* final state */
    $F \leftarrow F \cup \{q_i\}$
    $l \leftarrow l+1$; $r \leftarrow l+1$ /* go to the first state of next level */
  else
    if $l < k$ **then**
      $s \leftarrow i + m + 1 - l$ /* s = number of right lower neighbor */
      $\delta(q_i, p_r) \leftarrow \{q_{i+1}, q_s\}$ /* matching and *replace* transitions */
      $\delta(q_i, a) \leftarrow \{q_s\}$, $\forall a \in \Sigma \setminus \{p_r\}$ /* *replace* transition */
    else
      $\delta(q_i, p_r) \leftarrow \{q_{i+1}\}$ /* matching transition in k-th level */
    endif
    $r \leftarrow r+1$ /* go to the next depth of level l */
  endif
endfor
$\delta(q_0, a) \leftarrow \delta(q_0, a) \cup \{q_0\}$, $\forall a \in \Sigma$ /* self-loop of the initial state */

the labels of both transitions are disjoint, there can be at most one active state in depth $j+1$ in $(i+1)$-th step. □

Lemma 3.9
If *NFA* $M = (Q, \Sigma, \delta, q_0, F)$ for the approximate string matching using the Hamming distance (Σ version) constructed by Algorithm 3.9 reaches a state q (i.e., q is active) at depth i, $0 \le i \le m$, and level ℓ, $0 \le \ell \le k$, $\ell \le i$, then the states at depth i and levels j, $\ell < j \le k$, $j \le i$, are also active.

Proof
We prove by induction on the depth of state in the *NFA*, that

($*$) if a state in level ℓ, $0 \le \ell \le k$, and depth i, $0 \le i \le m$, is active, then all states at levels j, $\ell < j \le k$, $j \le i$, are also active.

There is only one state in depth 0 (i.e., q_0), therefore condition ($*$) holds.

Assume that state q in depth i and level ℓ is active and condition ($*$) holds.

If symbol p_{i+1} is the current input symbol, then the state q' in depth $i+1$ and level ℓ become active. Since all states of the same depth under q are active (because of condition ($*$)) and all matching transitions leading from the states of depth i to the states of depth $i+1$ are labeled by p_{i+1}, all states under q' also become active. If $i < k$, then the state q'' under q' at level $i+1$ (i.e., q'' is the lowest state in depth $i+1$) cannot become active using any matching transition, but it really becomes active, since there is a *replace* transition from the state at level i under q labeled by all symbols of alphabet Σ and this transition leads to q''. Except this state q'', all states, which could become active because of *replace* transitions labeled by all symbols of alphabet Σ, are already active because of the matching transitions.

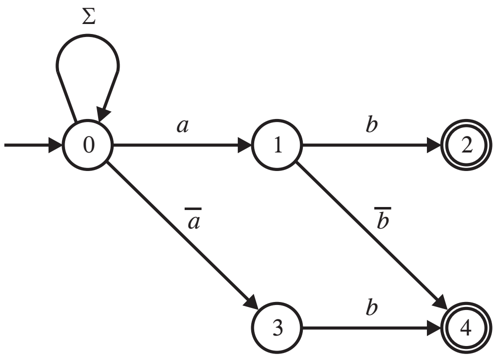
If the current input symbol is not p_{i+1}, only the *replace* transitions labeled by all symbols of alphabet Σ are used. So each of the active states in depth i and level j, $\ell \le j < k$, $j \le i$, (including q) causes that the state in depth $i+1$ and level $j+1$ becomes active. □

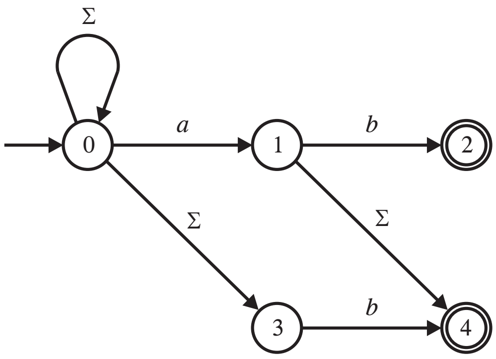
Theorem 3.10
The *NFA* constructed by Algorithm 3.8 ($\overline{p}$ version) and the *NFA* constructed by Algorithm 3.9 (Σ version) are equivalent with respect to the approximate string matching using the Hamming distance.

Proof
These *NFA*s differ only in representations of *replace* transitions. In the *NFA* constructed by Algorithm 3.9 (Σ version), all states under the topmost active state in the same depth are also active (See Lemma 3.9), but in the *NFA* constructed by Algorithm 3.8 ($\overline{p}$ version), only the topmost active state in each depth is active. It is because in each state (except the final states) there is only one transition for each symbol of alphabet Σ leading to the next depth (See Lemma 3.8).

In the approximate string matching using the Hamming distance we are interested in the minimum number of errors—we are interested in the topmost active final state and they are the same in both *NFA*s. Thus these *NFA*s are equivalent with respect to the approximate string matching using the Hamming distance. □

Example 3.11
Transform both $\overline{p}$ version and Σ version of *NFA* for the approximate string matching using the Hamming distance constructed for pattern $P = ab$ to their equivalent *DFA*s using Algorithm 3.2.





$\delta^{\overline{p}}_{NFA}$	a	b	$\Sigma \setminus \{a,b\}$
q_0	$\{q_0, q_1\}$	$\{q_0, q_3\}$	$\{q_0, q_3\}$
q_1	$\{q_4\}$	$\{q_2\}$	$\{q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\emptyset$	$\emptyset$	$\emptyset$

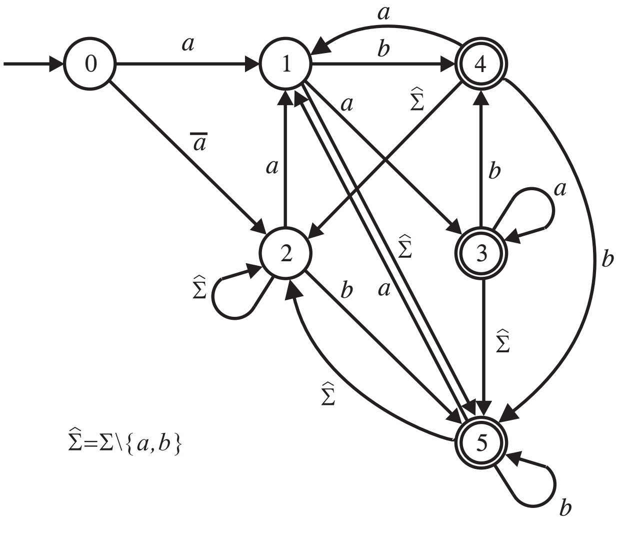
δ^{Σ}_{NFA}	a	b	$\Sigma \setminus \{a,b\}$
q_0	$\{q_0, q_1, q_3\}$	$\{q_0, q_3\}$	$\{q_0, q_3\}$
q_1	$\{q_4\}$	$\{q_2, q_4\}$	$\{q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\emptyset$	$\emptyset$	$\emptyset$

$\delta^{\overline{p}}_{DFA}$	a	b	$\Sigma \setminus \{a,b\}$
$[q_0]$	$[q_0, q_1]$	$[q_0, q_3]$	$[q_0, q_3]$
$[q_0, q_1]$	$[q_0, q_1, q_4]$	$[q_0, q_2, q_3]$	$[q_0, q_3, q_4]$
$[q_0, q_3]$	$[q_0, q_1]$	$[q_0, q_3, q_4]$	$[q_0, q_3]$
$[q_0, q_1, q_4]$	$[q_0, q_1, q_4]$	$[q_0, q_2, q_3]$	$[q_0, q_3, q_4]$
$[q_0, q_2, q_3]$	$[q_0, q_1]$	$[q_0, q_3, q_4]$	$[q_0, q_3]$
$[q_0, q_3, q_4]$	$[q_0, q_1]$	$[q_0, q_3, q_4]$	$[q_0, q_3]$

δ^{Σ}_{DFA}	a	b	$\Sigma \setminus \{a,b\}$
$[q_0]$	$[q_0, q_1, \mathbf{q_3}]$	$[q_0, q_3]$	$[q_0, q_3]$
$[q_0, q_1, \mathbf{q_3}]$	$[q_0, q_1, \mathbf{q_3}, q_4]$	$[q_0, q_2, q_3, \mathbf{q_4}]$	$[q_0, q_3, q_4]$
$[q_0, q_3]$	$[q_0, q_1, \mathbf{q_3}]$	$[q_0, q_3, q_4]$	$[q_0, q_3]$
$[q_0, q_1, \mathbf{q_3}, q_4]$	$[q_0, q_1, \mathbf{q_3}, q_4]$	$[q_0, q_2, q_3, \mathbf{q_4}]$	$[q_0, q_3, q_4]$
$[q_0, q_2, q_3, \mathbf{q_4}]$	$[q_0, q_1, \mathbf{q_3}]$	$[q_0, q_3, q_4]$	$[q_0, q_3]$
$[q_0, q_3, q_4]$	$[q_0, q_1, \mathbf{q_3}]$	$[q_0, q_3, q_4]$	$[q_0, q_3]$

The resulting two *DFA*s are isomorphic. If we rename the states of *DFA*s, we get the same *DFA*.

δ_{DFA}	a	b	$\Sigma \setminus \{a,b\}$
q_0	q_1	q_2	q_2
q_1	q_3	q_4	q_5
q_2	q_1	q_5	q_2
q_3	q_3	q_4	q_5
q_4	q_1	q_5	q_2
q_5	q_1	q_5	q_2



Theorem 3.12
Let $M^{\overline{p}} = (Q, \Sigma, \delta^{\overline{p}}, q_0, F)$ be the *NFA* for the approximate string matching using the Hamming distance ($\overline{p}$ version) constructed by Algorithm 3.8 for pattern $P = p_1 p_2 \ldots p_m$ and $M^{\Sigma} = (Q, \Sigma, \delta^{\Sigma}, q_0, F)$ be the *NFA* for the approximate string matching using the Hamming distance (Σ version) constructed by Algorithm 3.9 for pattern P. If we transform these *NFA*s to their equivalent *DFA*s using Algorithm 3.2, we get isomorphic *DFA*s.

Proof
$M^{\overline{p}}$ and M^{Σ} differ only in the representations of *replace* transitions. These different representations provide that if a state q is active in $M^{\overline{p}}$, no other state is active in the same depth (See Lemma 3.8), while if a state q is active in M^{Σ}, all states under q are also active in the same depth (See Lemma 3.9).

Algorithm 3.2 for transforming an *NFA* to the equivalent *DFA* assigns to each possible situation in *NFA* (represented by a set of active states) one deterministic state. Therefore it holds for each state q of M^{Σ} that if q is in some deterministic state q' (i.e., $q \in q'$), then all states under q in the same depth are also in q'.

If we remove all states under the topmost active state of each depth from each deterministic state of the *DFA* constructed from M^{Σ}, we get the *DFA* constructed from $M^{\overline{p}}$. □

It is also possible to estimate upper bound of number of states of *DFA* for the approximate string matching using the Hamming distance [Mel96a].

Theorem 3.13 ([Mel96a])
Let M_{NFA} be the *NFA* for the approximate string matching using the Hamming distance constructed by Algorithm 3.8 ($\overline{p}$ version) or Algorithm 3.9 (Σ version) for pattern $P = p_1p_2 \ldots p_m$ and maximum number of errors k. The equivalent *DFA* M_{DFA} constructed by Algorithm 3.2 has at most $\frac{(k+1)!}{2}(k+2)^{m-k+1}$ states.

Proof
In this proof we consider *NFA* constructed by Algorithm 3.8 ($\overline{p}$ version) without loss of generality (See Theorem 3.12).

We count a number of possibilities for active state in each depth. The number of possibilities is the number of states in each depth plus one possibility for the case when there is no active state in the depth.

At 0-th depth of the *NFA* there is only one state, which is always active, therefore there is only one possibility for active state in this depth.

Depths $1, 2, \ldots, k-1$ have $2, 3, \ldots, k$ states respectively and therefore there are $3, 4, \ldots, k+1$ possibilities for active state respectively.

In each of the last $(m-k+1)$ depths there are $k+1$ states therefore there are $k+2$ possibilities in these depths.

If we make a product of possibilities of all depths, we get the total number of possibilities for whole *NFA*, which is also the maximum number of states of the equivalent *DFA*.

$$1 * 3 * 4 * \cdots * (k+1) * \underbrace{(k+2) * \cdots * (k+2)}_{m-k+1} = \frac{(k+1)!}{2}(k+2)^{m-k+1}$$

□

3.4.3 Approximate String Matching Using Levenshtein Distance

In the approximate string matching using the Levenshtein distance we can use also $k+1$ copies of *NFA* for the exact string matching $(M_0, M_1, \ldots, M_k)$ and connect them by the transitions representing edit operation *replace*. Since in the Levenshtein distance there are besides edit operation *replace* other two edit operations allowed, we have to insert new transitions for these two edit operations into the *NFA* for the Hamming distance. The initial state of the resulting *NFA* is the initial state of automaton M_0 and inaccessible states are also removed.

The transition representing edit operation *insert* is labeled by symbol $\overline{p}_{j+1}$ and leads from state q_j of automaton M_i to state q_j of automaton M_{i+1}, $0 \le i < k$, $0 < j < m$—the minimum number of errors is increased but the depth in automaton remains the same. The transition representing edit operation *delete* is labeled by ε and leads from state q_j of automaton M_i to state q_{j+1} of automaton M_{i+1}, $0 \le i < k$, $0 \le j < m$—the depth in automaton is increased as well as the minimum number of errors but no symbol is read from the input.

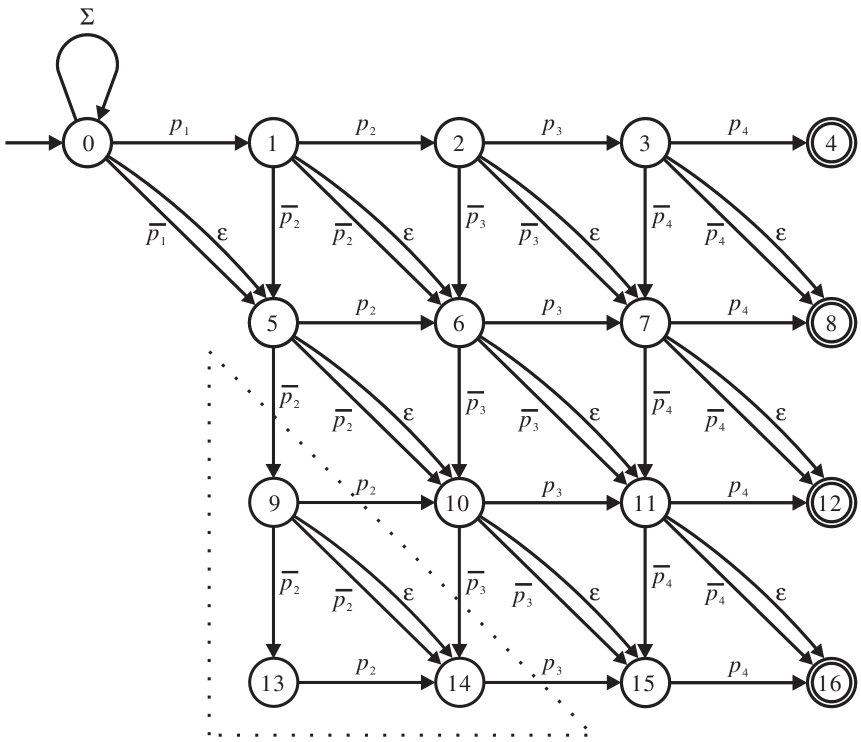


Figure 3.6: *NFA* for the approximate string matching using the Levenshtein distance ($m = 4$, $k = 3$).

The automaton, constructed by connecting $k + 1$ copies of *NFA* by transitions *replace*, *insert*, and *delete* as described above, was presented in [Mel96b] and an example of such *NFA* for $m = 4$ and $k = 3$ is shown in Figure 3.6. But in this *NFA* there are redundant states that can be removed (in Figure 3.6 bordered by the dotted line).

These states induce redundant states also in the resulting *DFA* when transforming this *NFA* to the equivalent *DFA* using Algorithm 3.2.

Theorem 3.14 (Redundant states)
In the *NFA* for the approximate string matching using the Levenshtein distance [Mel96b], the states under the diagonal leading from the initial states (i.e., states of depth i from the *NFA* M_j for the exact string matching such that $1 < j \le k + 1$, $1 \le i < j$) can be removed without any change of the behavior of the *NFA*.

Proof
Because of self-loop of the initial state q_0 of the *NFA* the initial state is always active. Since the states located on 0-th diagonal, which leads from the initial state q_0, are connected with the

initial state by ε-transitions, they are also active all the time (i.e., states of $\varepsilon CLOSURE(\{q_0\})$ are always active).

The states located under 0-th diagonal are separated from any of the final states by the states of 0-th diagonal. Any path leading from any state under 0-th diagonal must cross the states of 0-th diagonal, but these states are always active. The states under 0-th diagonal do not influence any state of 0-th diagonal and therefore they do not influence any of the final states. Hence, they are redundant and can be removed. □

In words of edit distance: each such redundant state of level j and depth i, $1 < j \leq k$, $1 \leq i < j$, represents number j of errors greater than depth i, which is obviously not minimal—it is greater than using i-times edit operation *delete*.

In order to eliminate such states from *NFA* directly in the construction of the *NFA* we put *insert* transition only to states q_j (of depth j) in automata M_i, $0 \leq i < k$, $i < j < m$.

The *NFA* with the redundant states eliminated can be constructed as described above, by a modification of *NFA* for the Hamming distance. An example of such *NFA* for $m = 4$ and $k = 3$ is shown in Figure 3.7.

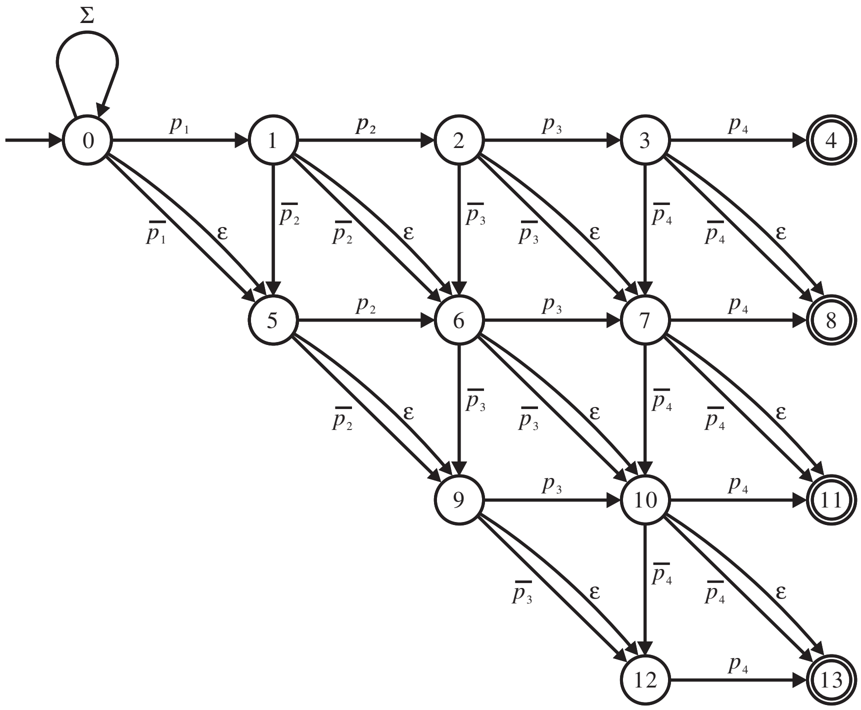


Figure 3.7: *NFA* for the approximate string matching using the Levenshtein distance—$\overline{p}$ version ($m = 4$, $k = 3$).

It is also possible to construct directly an equivalent *NFA* without ε-transitions as shown in Algorithm 3.10. In this algorithm the transitions are constructed for the states starting from the right-bottom corner. When all the transitions for state q of level l, $0 \leq l < k$, and depth r, $l \leq r < m$, are constructed (except *delete* transitions), all the transitions of state q' of level $l + 1$ and depth $r + 1$ are added to these transitions of state q, because there is an ε-transition leading from q to q'. If you replace the line for *delete* transition by $\delta(q_i, \varepsilon) = \{q_s\}$ in Algorithm 3.10, you would get the algorithm constructing exactly the *NFA* with ε-transitions for the approximate string matching using the Levenshtein distance as described above.

Like in the case of the approximate string matching using the Hamming distance, we can also construct another *NFA* for the approximate string matching using the Levenshtein distance in which transitions representing *replace* and *insert* are labeled by Σ. This *NFA* can be also constructed by connecting $k+1$ copies of *NFA* or directly as shown by Algorithm 3.11. This algorithm is similar to Algorithm 3.10 but the *replace* and *insert* transitions are modified (*replace* and *insert* transitions can be used also for matching symbol). An example of such *NFA* for $m=4$ and $k=3$ is shown in Figure 3.8.

Lemma 3.15
The *NFA*s for the approximate string matching using the Levenshtein distance constructed by Algorithm 3.10 or Algorithm 3.11 for a pattern of length m and maximum number of errors k have $(k+1)(m+1-\frac{k}{2})$ states.

Proof
The *NFA*s constructed by Algorithm 3.10 and Algorithm 3.11 have the same number of states as the *NFA*s for the Hamming distance constructed by Algorithm 3.8 and Algorithm 3.9 because of Theorem 3.14. □

The following lemma was proven in [MP80, Ukk85] for dynamic programming algorithm, which simulates *NFA* for the approximate string matching using the Levenshtein distance (Σ version) constructed by Algorithm 3.11 (the simulation is shown in Chapter 5). We prove it directly according to the *NFA* for the approximate string matching using the Levenshtein distance.

Lemma 3.16 ([MP80, Ukk85])
Let M be an *NFA* for the approximate string matching using the Levenshtein distance (Σ version) constructed by Algorithm 3.11 for pattern $P=p_1p_2\dots p_m$ and maximum number of errors k. Let lev_j^i be the level of the topmost active state in depth j, $0\le j\le m$, in i-th step of run of *NFA* (after i-th input symbol is read), or $k+1$ if there is no active state in depth j in i-th step. During the run of *NFA* with an input text of length n the following holds:

$$\forall j, 0\le j<m, \forall i, 1\le i\le n,\quad lev_j^i-1\le lev_{j+1}^i\le lev_j^i+1 \tag{3.3}$$

Proof
We prove it by induction on step of run of M.

At the beginning of the run the lemma holds, because only the states in 0-th diagonal are active (i.e., $\forall j, 0\le j\le k, lev_j^0=j$)—their levels always differ by 1—and states in depths not reachable by 0-th diagonal have no active state (i.e., $\forall j, k<j\le m, lev_j^0=k+1$).

Assume that in i-th step of the run the lemma holds. Let q be the topmost active state in depth j in i-th step and its level $lev_j^i=l$. Let q' be the topmost active state in depth $j+1$ in i-th step (its level must be $l-1$, l, or $l+1$ according to the assumption).

If symbol p_{j+1} is read from the input, then all transitions leading from q are used. Therefore in step $i+1$ the topmost state in depth j is at least[3] at level $l+1$ (i.e., $lev_j^{i+1}\le l+1$; vertical *insert* transition is used) and the topmost active state in depth $j+1$ is at level l (i.e., $lev_{j+1}^{i+1}=l$; horizontal matching transition is used). Note, that the level of the topmost active state of depth

[3] "At least" means that we do not consider impact of states of the previous depth in this step, but it is considered in this proof.

Algorithm 3.10 (Construction of *NFA* for the approximate string matching using the Levenshtein distance—$\overline{p}$ version)
Input: Pattern $P = p_1p_2 \dots p_m$, maximum number of errors allowed k, $k < m$.
Output: *NFA* M accepting language $L(M) = \{wu \mid w, u \in \Sigma^*, D_L(P, u) \leq k\}$.
Method: *NFA* $M = (Q, \Sigma, \delta, q_0, F)$, where Q, δ, and F are constructed in the following way:

$Q \leftarrow \{q_0, q_1, \dots, q_{|Q|-1}\}$, $|Q| = (k+1)(m+1-\frac{k}{2})$
$l \leftarrow k$ /* number of level */
$r \leftarrow m$ /* order of symbol in P (= depth of state) */
for $i \leftarrow |Q|-1, |Q|-2, \dots, 0$ **do**
  if $r = m$ **then**
    $\delta(q_i, a) \leftarrow \emptyset$, $\forall a \in \Sigma$ /* final state */
    $F \leftarrow F \cup \{q_i\}$
  else
    $\delta(q_i, p_{r+1}) \leftarrow \{q_{i+1}\}$ /* matching transition */
    if $l < k$ **then**
      $s \leftarrow i + m + 1 - l$ /* s = number of right lower neighbor */
      $\delta(q_i, a) \leftarrow \{q_s\}$, $\forall a \in \Sigma \setminus \{p_{r+1}\}$ /* *replace* transition */
      if $r > l$ **then**
        $\delta(q_i, a) \leftarrow \delta(q_i, a) \cup \{q_{s-1}\}$, $\forall a \in \Sigma \setminus \{p_{r+1}\}$ /* *insert* transition */
      endif
      $\delta(q_i, a) \leftarrow \delta(q_i, a) \cup \delta(q_s, a)$, $\forall a \in \Sigma$ /* *delete* transition */
    else
      $\delta(q_i, a) \leftarrow \emptyset$, $\forall a \in \Sigma \setminus \{p_{r+1}\}$
    endif
  endif
  $r \leftarrow r - 1$ /* go to the previous depth of level l */
  if $r < l$ **then** /* is r out of states of level l? */
    $r \leftarrow m$ /* go to the final state of the previous level */
    $l \leftarrow l - 1$
  endif
endfor
$\delta(q_0, a) \leftarrow \delta(q_0, a) \cup \{q_0\}$, $\forall a \in \Sigma$ /* self-loop of the initial state */

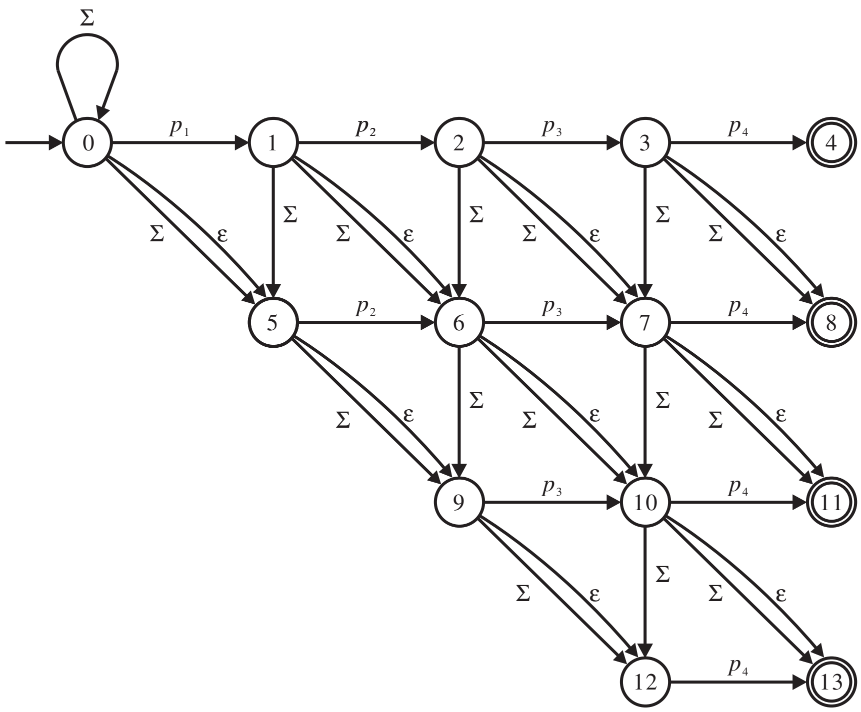


Figure 3.8: *NFA* for the approximate string matching using the Levenshtein distance—Σ version ($m = 4$, $k = 3$).

$j+1$ in step $i+1$ cannot be higher (i.e., $lev_{j+1}^{i+1} \not< l$) since if the topmost active state in this depth in i-th step would be at level $l-1$, the only transition leading from this state within this depth leads to the state in level l.

If any symbol mismatching p_{j+1} is read from the input, then in step $i+1$ the topmost active state in depth j is at least at level $l+1$ (i.e., $lev_j^{i+1} \leq l+1$; vertical *insert* transition is used) and the topmost active state in depth $j+1$ is also at least at level $l+1$ (i.e., $lev_{j+1}^{i+1} \leq l+1$; diagonal transition *replace* is used). If the topmost active state of depth $j+1$ in i-th step of the run would be in level $l-1$, then the topmost active state of depth $j+1$ in step $i+1$ would be l.

Moreover, level of the topmost active state in depth $j+1$ cannot be greater than $l+1$ since q is connected with state q'' of depth $j+1$ and level $l+1$ by ε-transition, therefore whenever q is active, q'' is active as well ($q'' \in \varepsilon CLOSURE(\{q\})$). □

A similar lemma can also be formed for $\overline{p}$ version of *NFA* for the Levenshtein distance, as shown in Lemma 3.17.

Lemma 3.17

Let M be an *NFA* for the approximate string matching using the Levenshtein distance ($\overline{p}$ version) constructed by Algorithm 3.10 for pattern $P = p_1p_2 \ldots p_m$ and maximum number of errors k. Let lev_j^i be the level of the topmost active state in depth j, $0 \leq j \leq m$, in i-th step of run of *NFA* (after i-th input symbol is read), or $k+1$ if there is no active state in depth j in i-th step. During the run of *NFA* with an input text of length n the following holds:

Algorithm 3.11 (Construction of *NFA* for the approximate string matching using the Levenshtein distance—Σ version)
Input: Pattern $P = p_1 p_2 \dots p_m$, maximum number of errors allowed k, $k < m$.
Output: *NFA* M accepting language $L(M) = \{wu \mid w, u \in \Sigma^*, D_L(P, u) \leq k\}$.
Method: *NFA* $M = (Q, \Sigma, \delta, q_0, F)$, where Q, δ, and F are constructed in the following way:

```
Q ← {q_0, q_1, ..., q_{|Q|-1}}, |Q| = (k+1)(m+1-k/2)
l ← k                                              /* number of level */
r ← m                           /* order of symbol in P (= depth of state) */
for i ← |Q|-1, |Q|-2, ..., 0 do
  if r = m then
    δ(q_i, a) ← ∅, ∀a ∈ Σ                                /* final state */
    F ← F ∪ {q_i}
  else
    if l < k then
      s ← i + m + 1 - l             /* s = number of right lower neighbor */
      δ(q_i, p_{r+1}) ← {q_{i+1}, q_s}   /* matching and replace transitions */
      δ(q_i, a) ← {q_s}, ∀a ∈ Σ \ {p_{r+1}}          /* replace transition */
      if r > l then
        δ(q_i, a) ← δ(q_i, a) ∪ {q_{s-1}}, ∀a ∈ Σ     /* insert transition */
      endif
      δ(q_i, a) ← δ(q_i, a) ∪ δ(q_s, a), ∀a ∈ Σ       /* delete transition */
    else
      δ(q_i, p_{r+1}) ← {q_{i+1}}         /* matching transition in k-th level */
      δ(q_i, a) ← ∅, ∀a ∈ Σ \ {p_{r+1}}
    endif
  endif
  r ← r - 1                          /* go to the previous depth of level l */
  if r < l then                            /* is r out of states of level l? */
    r ← m                       /* go to the final state of the previous level */
    l ← l - 1
  endif
endfor
δ(q_0, a) ← δ(q_0, a) ∪ {q_0}, ∀a ∈ Σ        /* self-loop of the initial state */
```

$$\forall j, 0 \leq j < m, \forall i, 1 \leq i \leq n, \quad lev_{j+1}^i \leq lev_j^i + 1 \tag{3.4}$$

Proof
Each state q of depth j, $0 \leq j < m$, and level l, $0 \leq l < k$, has an ε-transition leading from q to the state q' of depth $j+1$ and level $l+1$. Therefore if q is active in i-th step, then q' is also active in i-th step ($q' \in \varepsilon CLOSURE(\{q\})$). Thus if q is the topmost active state in depth j and q'' is the topmost active state of depth $j+1$, then $level(q'') \leq level(q') = level(q) + 1$. □

In the following example we show differences among *NFA* with *insert* transitions leading from final states (not considered further in this thesis), Σ version and $\overline{p}$ version of *NFA*.

Example 3.18
Let $P = abc$, $T = dabcddd$, and $k = 2$. We construct *NFA* M_0 for the approximate string matching using the Levenshtein distance, which has *insert* transitions leading from final states—it finds also trivial occurrences. We also construct M_Σ and $M_{\overline{p}}$, which are respectively Σ version and $\overline{p}$ version of *NFA* for the approximate string matching using the Levenshtein distance.

These *NFA*s are shown in Figure 3.9.

If we let these *NFA*s run over input text T, we get the following sequences of configurations:

M_0:
$(\{q_0, q_4, q_7\}, dabcddd) \vdash (\{q_0, q_4, q_7\}, abcddd) \vdash (\{q_0, q_1, q_4, q_5, q_7, q_8\}, bcddd)$
$\vdash (\{q_0, q_2, q_4, q_5, q_6, q_7, q_8\}, cddd) \vdash (\{q_0, q_3, q_4, q_5, q_6, q_7, q_8\}, ddd) \vdash (\{q_0, q_4, q_6, q_7, q_8\}, dd)$
$\vdash (\{q_0, q_4, q_7, q_8\}, d) \vdash (\{q_0, q_4, q_7\}, \varepsilon)$

M_Σ:
$(\{q_0, q_4, q_7\}, dabcddd) \vdash (\{q_0, q_4, q_7\}, abcddd) \vdash (\{q_0, q_1, q_4, q_5, q_7, q_8\}, bcddd)$
$\vdash (\{q_0, q_2, q_4, q_5, q_6, q_7, q_8\}, cddd) \vdash (\{q_0, q_3, q_4, q_5, q_6, q_7, q_8\}, ddd) \vdash (\{q_0, q_4, q_7, q_8\}, dd)$
$\vdash (\{q_0, q_4, q_7\}, d) \vdash (\{q_0, q_4, q_7\}, \varepsilon)$

$M_{\overline{p}}$:
$(\{q_0, q_4, q_7\}, dabcddd) \vdash (\{q_0, q_4, q_7\}, abcddd) \vdash (\{q_0, q_1, q_4, q_5, q_7, q_8\}, bcddd)$
$\vdash (\{q_0, q_2, q_4, q_6, q_7, q_8\}, cddd) \vdash (\{q_0, q_3, q_4, q_7, q_8\}, ddd) \vdash (\{q_0, q_4, q_7\}, dd)$
$\vdash (\{q_0, q_4, q_7\}, d) \vdash (\{q_0, q_4, q_7\}, \varepsilon)$

The following table shows the results (numbers of errors of the found strings) reported by *NFA*s:

	1	2	3	4	5	6	7
T	d	a	b	c	d	d	d
M_0	-	2	1,2	0,1,2	1,2	2	-
M_Σ	-	2	1,2	0,1,2	2	-	-
$M_{\overline{p}}$	-	2	1,2	0,2	-	-	-

All three *NFA*s report occurrences of a pattern with $k, k-1, \ldots, l+1$ errors (i.e., $k, k-1, \ldots, l+1$ symbols are deleted from the end of the pattern) in front of an occurrence of the pattern with l errors[4], $0 \leq l \leq k$.

As we can see, the difference between M_0 and M_Σ is that M_0 reports also trivial occurrences (at position 5 d is appended and at position 6 dd is appended)—M_0 reports occurrences of the

[4] $l = 0$ in this example.

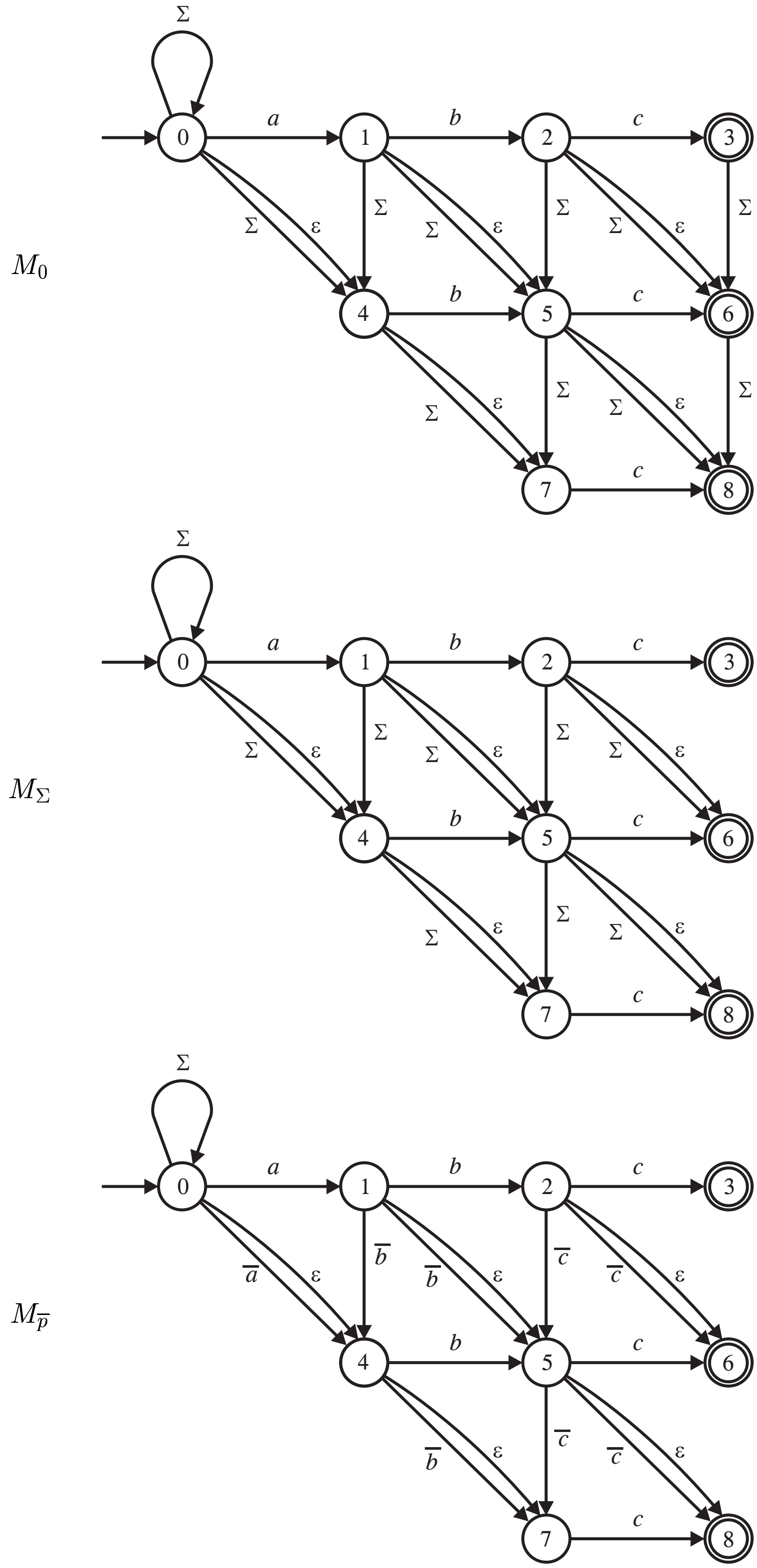


Figure 3.9: *NFA*s M_0, M_Σ, and $M_{\overline{p}}$.

pattern with $l+1, l+2, \ldots, k$ errors behind an occurrence of the pattern with l errors (i.e., $l+1, l+2, \ldots, k$ are inserted at the end of the pattern).

M_Σ reports the occurrence with 2 errors at position 5 like M_0, but this occurrence is not caused just by appending symbols behind the previous occurrence, this is new occurrence created by inserting symbol c into P (at position 4) and replacing symbol c in P by symbol d (at position 5). This occurrence is not found by $M_{\overline{p}}$ since it is also caused by occurrence without errors.

$M_{\overline{p}}$ also does not report the occurrence with 1 error at position 4. Since we are interested in occurrences with the lowest number of errors, we do not need to report such occurrences. This occurrence is caused by replacing c by c at position 4.

From the comparison of M_0, M_Σ, and $M_{\overline{p}}$ we can also see that $M_{\overline{p}}$ has usually less active states than M_Σ (but at most as many as M_Σ) and that M_0 has more active states than M_Σ behind some occurrence, but it has never less active states than M_Σ.

Theorem 3.19 ([Mel96a])
Let M_{NFA} be the *NFA* for the approximate string matching using the Levenshtein distance constructed by Algorithm 3.10 ($\overline{p}$ version) for pattern $P = p_1p_2\ldots p_m$ and maximum number of errors k. The equivalent *DFA* M_{DFA} constructed from M_{NFA} by Algorithm 3.2 has at most $(k+1)!(k+2)^{m-k}$ states.

Proof
We count a number of possibilities for active states in each ε-diagonal. The number of possibilities is the number of states in each ε-diagonal plus one possibility for the case when there is no active state in the ε-diagonal.

At the beginning of 0-th ε-diagonal of the *NFA* there is the initial state, which is always active, therefore there is only one possibility for active states in this ε-diagonal—all states of 0-th ε-diagonal are always active.

Each of ε-diagonals $1, 2, \ldots, m-k$ has $k+1$ states therefore there are $k+2$ possibilities in each of these ε-diagonals—if a state q is active, then all states under q in the same ε-diagonal are also active, since they are connected with q by ε-transitions (i.e., they are in $\varepsilon CLOSURE(\{q\})$).

In each of the last k ε-diagonals there are $k, k-1, \ldots, 1$ states respectively therefore there are $k+1, k, \ldots, 2$ possibilities for active states respectively.

If we make a product of possibilities of all ε-diagonals, we get the total number of possibilities for whole *NFA*, which is also the maximum number of states of the equivalent *DFA*.

$$1 * \underbrace{(k+2) * \cdots * (k+2)}_{m-k} * (k+1) * k * \cdots * 2 = (k+2)^{m-k} * (k+1)!$$

□

Theorem 3.20
Let M_{NFA} be an *NFA* for the approximate string matching using the Levenshtein distance (Σ version) constructed by Algorithm 3.11 for pattern $P = p_1p_2\ldots p_m$ and maximum number of errors k. The equivalent *DFA* M_{DFA} constructed from M_{NFA} by Algorithm 3.2 has at most $\mathbf{min}(3^m, (k+1)!(k+2)^{m-k})$ states.

Proof
The upper bound $(k+1)!(k+2)^{m-k}$ is proven in the proof of Theorem 3.19. The upper bound 3^m follows from Theorem 3.16 and it was shown in [Ukk85]. It is based on ternary[5] tree representation of topmost active states (i.e., ternary tree representation of deterministic states). □

[5]Since differences among levels of the topmost active states of adjacent depths are $-1, 0, +1$.

3.4.4 Approximate String Matching Using Generalized Levenshtein Distance

In the approximate string matching using the generalized Levenshtein distance we can use also $k+1$ copies of *NFA* for the exact string matching $(M_0, M_1, \ldots, M_k)$ and connect them by the transitions representing edit operations *replace*, *insert*, and *delete*. Since in the generalized Levenshtein distance there is besides edit operations *replace*, *insert*, and *delete* a new edit operation *transpose* allowed, we have to insert new transitions for this edit operation into the *NFA* for the Levenshtein distance.

The transition representing edit operation *transpose* leads from state q_j, $0 < j < m-1$, of automaton M_i, $0 \le i < k$, to new state q' and is labeled by symbol p_{j+2}. Then from state q' a transition labeled by symbol p_{j+1} leads to state q_{j+2} of automaton M_{i+1}. So two adjacent symbols q_{j+1} and q_{j+2} are accepted in reverse order, the depth in automaton is increased of 2 and the minimum number of errors is increased of 1.

Let us note that edit operation *transpose* can be replaced by two adjacent edit operations *replace* but it increases the number of errors of 2. Therefore if we want to value edit operation *transpose* only at 1 error as well as for *replace*, *insert*, and *delete*, we have to use the generalized Levenshtein distance.

The *NFA* for the approximate string matching using the generalized Levenshtein distance can be constructed by connecting $k+1$ copies of *NFA* as described above. An example of such *NFA* for $m = 4$ and $k = 3$ is shown in Figure 3.10.

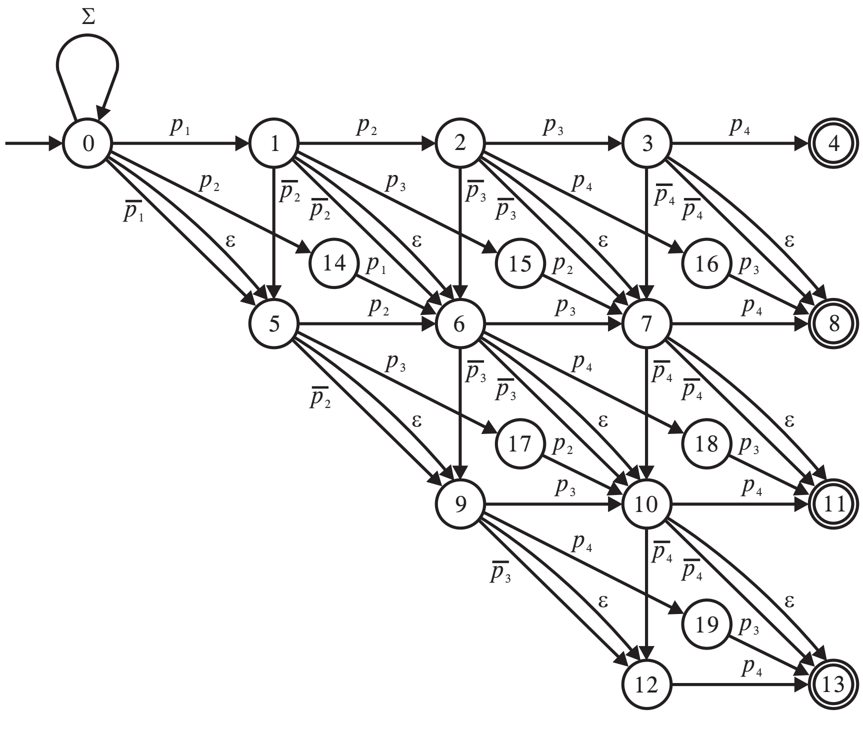


Figure 3.10: *NFA* for the approximate string matching using the generalized Levenshtein distance—$\overline{p}$ version ($m = 4$, $k = 3$).

Algorithm 3.12 constructs an equivalent *NFA* with ε-transitions eliminated. If we replace the line for *delete* transition (like in the previous section for Algorithm 3.10), we get the direct construction of the *NFA* with ε-transitions for the approximate string matching using the generalized Levenshtein distance as described in the previous paragraphs.

Algorithm 3.12 (Construction of *NFA* for the approximate string matching using the generalized Levenshtein distance—$\overline{p}$ version)
Input: Pattern $P = p_1p_2 \dots p_m$, maximum number of errors allowed k, $k < m$.
Output: *NFA* M accepting language $L(M) = \{wu \mid w, u \in \Sigma^*, D_G(P, u) \leq k\}$.
Method: *NFA* $M = (Q, \Sigma, \delta, q_0, F)$, where Q, δ, and F are constructed in the following way:

```
Q ← {q_0, q_1, ..., q_{|Q|-1}}, |Q| = (m + 1) + k(2m − k)
l ← k                                                        /* number of level */
r ← m                                      /* order of symbol in P (= depth of state) */
j ← |Q| − 1                                      /* index of the last transpose state */
for i ← (k + 1)(m + 1 − k/2) − 1, (k + 1)(m + 1 − k/2) − 2, ..., 0 do
  if r = m then
    δ(q_i, a) ← ∅, ∀a ∈ Σ                                         /* final state */
    F ← F ∪ {q_i}
  else
    δ(q_i, p_{r+1}) ← {q_{i+1}}                               /* matching transition */
    if l < k then
      s ← i + m + 1 − l                      /* s = number of right lower neighbor */
      δ(q_i, a) ← {q_s}, ∀a ∈ Σ \ {p_{r+1}}                  /* replace transition */
      if r > l then
        δ(q_i, a) ← δ(q_i, a) ∪ {q_{s−1}}, ∀a ∈ Σ \ {p_{r+1}}   /* insert transition */
      endif
      if r < m − 1 then
        δ(q_i, p_{r+2}) ← δ(q_i, p_{r+2}) ∪ {q_j}              /* transpose transition */
        δ(q_j, p_{r+1}) ← {q_{s+1}}
        j ← j − 1
      endif
      δ(q_i, a) ← δ(q_i, a) ∪ δ(q_s, a), ∀a ∈ Σ                  /* delete transition */
    else
      δ(q_i, a) ← ∅, ∀a ∈ Σ \ {p_{r+1}}
    endif
  endif
  r ← r − 1                                   /* go to the previous depth of level l */
  if r < l then                                     /* is r out of states of level l? */
    r ← m                               /* go to the final state of the previous level */
    l ← l − 1
  endif
endfor
δ(q_0, a) ← δ(q_0, a) ∪ {q_0}, ∀a ∈ Σ                  /* self-loop of the initial state */
```

Like in the case of the approximate string matching using the Levenshtein distance, we can also construct another *NFA* for the approximate string matching using the generalized Levenshtein distance in which transitions representing *replace* and *insert* are labeled by Σ. This *NFA* can be also constructed by connecting $k+1$ copies of *NFA* or directly as shown by Algorithm 3.13. This algorithm is similar to Algorithm 3.12 but the *replace* and *insert* transitions are modified (*replace* and *insert* transitions can be used also for matching symbol). An example of such *NFA* for $m = 4$ and $k = 3$ is shown in Figure 3.11.

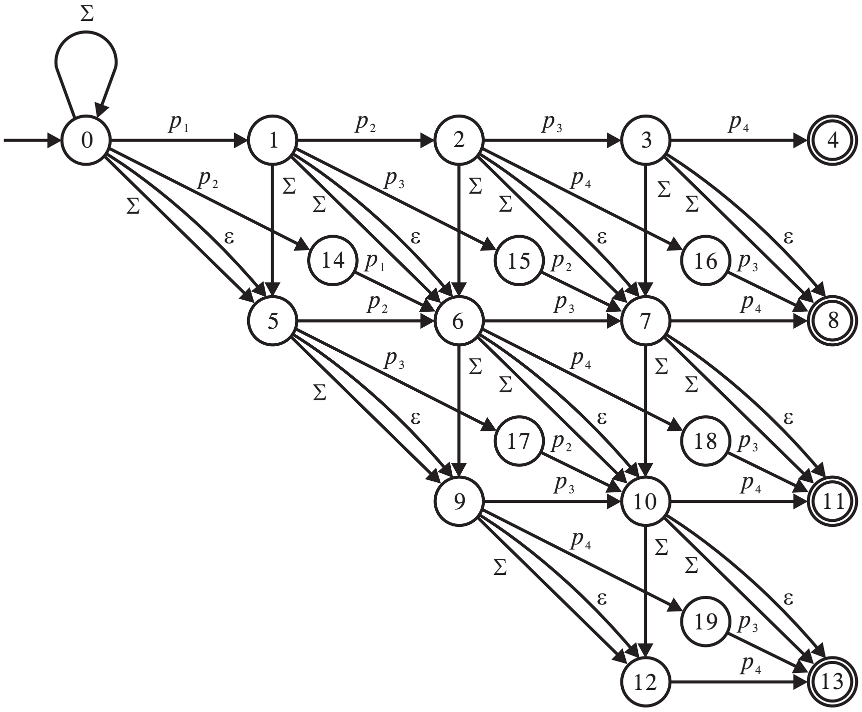


Figure 3.11: *NFA* for the approximate string matching using the generalized Levenshtein distance—Σ version ($m = 4$, $k = 3$).

Lemma 3.21
The *NFA*s for the approximate string matching using the generalized Levenshtein distance constructed by Algorithm 3.12 and Algorithm 3.13 for a pattern of length m and maximum number of errors k have both $(m+1) + k(2m-k)$ states.

Proof
In the *NFA*s for the approximate string matching using the generalized Levenshtein distance we can use Theorem 3.14. If we add the states that are necessary for *transpose* transitions, we get $(m+1) + (m-1+m+1-1) + (m-2+m+1-2) + \cdots = (m+1) + \sum_{i=1}^{k}(2m+1-2i) = (m+1) + k(2m-k)$ states. $\square$

The difference between behavior of $\overline{p}$ and Σ version of *NFA* for the approximate string matching using the generalized Levenshtein distance is the same as in the case of *NFA*s for the approximate string matching using the Levenshtein distance discussed in the previous section.

For the *DFA* built from an *NFA* for the approximate string matching using the generalized Levenshtein distance constructed by Algorithm 3.12 or Algorithm 3.13 (both $\overline{p}$ and Σ versions) for a pattern of length m and maximum number of errors k, we still do not know, how to determine

Algorithm 3.13 (Construction of *NFA* for the approximate string matching using the generalized Levenshtein distance—Σ version)
Input: Pattern $P = p_1 p_2 \dots p_m$, maximum number of errors allowed k, $k < m$.
Output: *NFA* M accepting language $L(M) = \{wu \mid w, u \in \Sigma^*, D_G(P, u) \leq k\}$.
Method: *NFA* $M = (Q, \Sigma, \delta, q_0, F)$, where Q, δ, and F are constructed in the following way:

```
Q ← {q_0, q_1, ..., q_{|Q|-1}}, |Q| = (m + 1) + k(2m − k)
l ← k                                                    /* number of level */
r ← m                                  /* order of symbol in P (= depth of state) */
j ← |Q| − 1                                   /* index of the last transpose state */
for i ← (k + 1)(m + 1 − k/2) − 1, (k + 1)(m + 1 − k/2) − 2, ..., 0 do
  if r = m then
    δ(q_i, a) ← ∅, ∀a ∈ Σ                                     /* final state */
    F ← F ∪ {q_i}
  else
    if l < k then
      s ← i + m + 1 − l                      /* s = number of right lower neighbor */
      δ(q_i, p_{r+1}) ← {q_{i+1}, q_s}         /* matching and replace transitions */
      δ(q_i, a) ← {q_s}, ∀a ∈ Σ \ {p_{r+1}}                 /* replace transition */
      if r > l then
        δ(q_i, a) ← δ(q_i, a) ∪ {q_{s−1}}, ∀a ∈ Σ            /* insert transition */
      endif
      if r < m − 1 then
        δ(q_i, p_{r+2}) ← δ(q_i, p_{r+2}) ∪ {q_j}          /* transpose transition */
        δ(q_j, p_{r+1}) ← {q_{s+1}}
        j ← j − 1
      endif
      δ(q_i, a) ← δ(q_i, a) ∪ δ(q_s, a), ∀a ∈ Σ              /* delete transition */
    else
      δ(q_i, p_{r+1}) ← {q_{i+1}}              /* matching transition of k-th level */
      δ(q_i, a) ← ∅, ∀a ∈ Σ \ {p_{r+1}}
    endif
  endif
  r ← r − 1                              /* go to the previous depth of level l */
  if r < l then                                 /* is r out of states of level l? */
    r ← m                            /* go to the final state of the previous level */
    l ← l − 1
  endif
endfor
δ(q_0, a) ← δ(q_0, a) ∪ {q_0}, ∀a ∈ Σ                /* self-loop of the initial state */
```

an upper bound of the number of states of the *DFA* that could be significantly different from the general worst case bound $2^{|Q_{NFA}|}$, which is here $2^{(m+1)+k(2m-k)}$.

3.5 Exact and Approximate Sequence Matching

The sequence matching problem differs from the string matching problem only in such a way that between the occurrences of two adjacent symbols p_j and p_{j+1}, $0 < j < m$, of pattern P in text T there can be located any number of symbols x, $x \in \Sigma \setminus \{p_{j+1}\}$. Therefore we add self-loop labeled by $\overline{p}_{j+1}$ to each state of depth j, $0 < j < m$, in the *NFA*s for the string matching and we get *NFA*s for the sequence matching [Hol97]. These self-loops skip all the symbols located between two occurrences of two adjacent symbols of the pattern. The *NFA* for the exact sequence matching can be constructed by Algorithm 3.14 and an example of such *NFA* is shown in Figure 3.12 where $m = 4$.

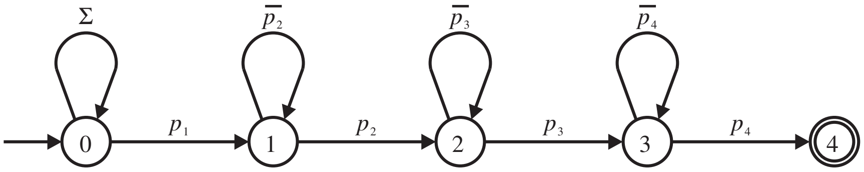


Figure 3.12: *NFA* for the exact sequence matching ($m = 4$).

The best case according to the number of states of the *DFA* equivalent to *NFA* for the exact sequence matching (constructed by Algorithm 3.14) is, when all symbols of the pattern $P = p_1 p_2 \dots p_m$ are the same (i.e., $p_i = p_j, \forall i, j, 0 < i \leq m, 0 < j \leq m$) as shown in Lemma 3.22 and Example 3.23.

Lemma 3.22
Let M_{NFA} be the *NFA* for the exact sequence matching constructed by Algorithm 3.14 for pattern $P = p_1 p_2 \dots p_m$, where $p_i = p_j, \forall i, j, 1 \leq i \leq m, 1 \leq j \leq m$. Let M_{DFA} be the *DFA* constructed from M_{NFA} by Algorithm 3.2. M_{DFA} has $m + 1$ states as well as M_{NFA}.

Proof
Let $M_{NFA} = (Q, \Sigma, \delta, q_0, F)$ and $M_{DFA} = (Q', \Sigma, \delta', q'_0, F')$. Let $p = p_1$ ($p = p_2, p = p_3, \dots, p = p_m$ as well).

M_{NFA} has $m + 1$ states (See Algorithm 3.14). Using Algorithm 3.2 we transform M_{NFA} to M_{DFA} that has the same number of states. We prove by induction on the depth of state in automaton that each deterministic state q' except the final state generates one new state among the transitions leading from q'.

It holds that

$$\begin{array}{lcll} \delta(q_0, a) & = & \{q_0\}, \forall a \in \Sigma \setminus \{p\} & /* \text{ self-loop of initial state } */ \\ \delta(q_0, p) & = & \{q_0, q_1\}. & /* \text{ matching transition } */ \end{array}$$

When the for-cycle $\mathcal{A}$ of Algorithm 3.2 is performed at first, it holds that

$$\begin{array}{lclcl} \delta'(q'_0, a) & = & \{q_0\} & = & q'_0, \forall a \in \Sigma \setminus \{p\} \\ \delta'(q'_0, p) & = & \{q_0, q_1\} & = & q'_1. \end{array}$$

Thus we obtain just one new state $q'_1 = \{q_0, q_1\} = q'_0 \cup \{q_1\}$ in the set S of not yet processed states, where $q'_0 = \{q_0\}$ is an initial state.

Assume that for all states q'_i, $0 \leq i < m - 1$, it holds that

$$\begin{array}{lcl} \delta'(q'_i, a) & = & q'_i, \forall a \in \Sigma \setminus \{p\} \\ \delta'(q'_i, p) & = & q'_{i+1}, \end{array}$$

and $q'_i = \{q_j \mid 0 \leq j \leq i\}$.

We compute transitions for q'_{i+1}. $\forall q_j, 0 < j < m$, it holds that

---

Algorithm 3.14 (Construction of *NFA* for the exact sequence matching)
Input: Pattern $P = p_1p_2 \ldots p_m$.
Output: *NFA* M accepting language $L(M) = \{w \mid w \in \Sigma^*, \exists\{i_1, i_2, \ldots, i_m \mid i_\ell < i_{\ell+1}, 0 < \ell < m, p_j = w_{i_j}, 0 < j \leq m\}\}$.
Method: *NFA* $M = (\{q_0, q_1, \ldots, q_m\}, \Sigma, \delta, q_0, \{q_m\})$, where the mapping δ is constructed in the following way:

$\delta(q_0, a) \leftarrow \{q_0\}, \forall a \in \Sigma \setminus \{p_1\}$ /∗ self-loop of the initial state ∗/
$\delta(q_0, p_1) \leftarrow \{q_0, q_1\}$
for $i \leftarrow 1, 2, \ldots, m-1$ **do**
  $\delta(q_i, a) \leftarrow \{q_i\}, \forall a \in \Sigma \setminus \{p_{i+1}\}$ /∗ self-loop of mediate states ∗/
  $\delta(q_i, p_{i+1}) \leftarrow \{q_{i+1}\}$ /∗ forward transitions ∗/
endfor
$\delta(q_m, a) \leftarrow \emptyset, \forall a \in \Sigma$ /∗ no transition leads from the final state ∗/

$$\begin{aligned}\delta(q_j, a) &= \{q_j\}, \forall a \in \Sigma \setminus \{p\}\\ \delta(q_j, p) &= \{q_{j+1}\},\end{aligned}$$

and $q'_{i+1} = q'_i \cup \{q_{i+1}\}$, therefore

$$\begin{aligned}\delta'(q'_{i+1}, a) &= \delta'(q'_i, a) \cup \delta(q_{i+1}, a) = q'_i \cup \{q_{i+1}\} = q'_{i+1}, \forall a \in \Sigma \setminus \{p\}\\ \delta'(q'_{i+1}, p) &= \delta'(q'_i, p) \cup \delta(q_{i+1}, p) = q'_{i+1} \cup \{q_{i+2}\} = q'_{i+2}.\end{aligned}$$

When determining transitions for final state q'_m, no new state appears because for final state q_m it holds that $\delta(q_m, a) = \emptyset, \forall a \in \Sigma$.

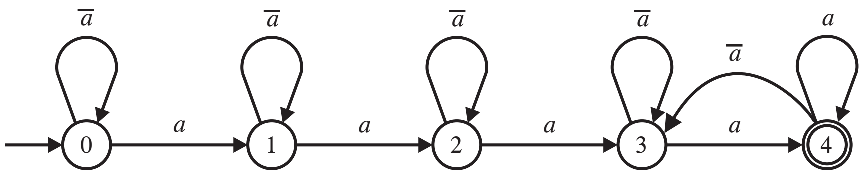
The for-cycle $\mathcal{A}$ of Algorithm 3.2 is performed $(m+1)$ times; once for the initial state (one new state is generated), $(m-1)$ times for state q'_i, $0 < i < m$, (one new state is generated in each step) and once for final state q'_m (no new state is generated).

Summing up initial state $q'_0 = \{q_0\}$, new state generated by the initial state, and $(m-1)$ states generated by states q'_i, $0 < i < m$, we get $(m+1)$ states. □

Example 3.23
Let $P = aaaa$. We construct M_{NFA} using Algorithm 3.14 and then we transform it to M_{DFA} using Algorithm 3.2.

M_{NFA}	a	$\Sigma \setminus \{a\}$
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\{q_1\}$
q_2	$\{q_3\}$	$\{q_2\}$
q_3	$\{q_4\}$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$

M_{DFA}	a	$\Sigma \setminus \{a\}$
$[q_0]$	$[q_0, q_1]$	$[q_0]$
$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_0, q_1]$
$[q_0, q_1, q_2]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_2]$
$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_2, q_3, q_4]$	$[q_0, q_1, q_2, q_3]$
$[q_0, q_1, q_2, q_3, q_4]$	$[q_0, q_1, q_2, q_3, q_4]$	$[q_0, q_1, q_2, q_3]$



The number of states of M_{NFA} is 5 and the number of states of M_{DFA} is also 5.

Theorem 3.24
Let M_{NFA} be an *NFA* for the exact sequence matching constructed by Algorithm 3.14 for pattern $P = p_1p_2 \ldots p_m$, $m > 1$. The equivalent *DFA* M_{DFA} constructed from M_{NFA} by Algorithm 3.2 has at most $\frac{3}{4}2^m$ states.

Proof
Let q_i be a state of M_{NFA} in depth i, $0 \leq i \leq m$. At the beginning, only state q_0 (the initial state) is active—there is only one possibility.

If q_m is active, then there is at most 2^{m-2} ways, how states $q_1, q_2, \ldots, q_{m-1}$ can be active:

- If $p_m \neq p_{m-1}$, then q_m cannot be active at the same time. q_m is active only after q_{m-1} had been active and p_m was read, but q_{m-1} cannot be active at the same time, since no transition labeled by p_m leads to q_{m-1}.
- If $p_m = p_{m-1}$, then q_m and q_μ cannot be active at the same time, where μ is the index of the last symbol of P mismatching p_m. q_μ cannot be active when q_m is active, since no transition labeled by p_m leads to q_μ—self-loop of q_μ is labeled by $\overline{p}_m$ ($p_m = p_{\mu+1}$) and transition leading from $q_{\mu-1}$ to q_μ is labeled by p_μ ($p_m \neq p_\mu$).

  If there does not exist any μ, such that $p_m \neq p_\mu$ (i.e., $p_i = p_j, \forall i, j, 0 < i \leq m, 0 < j \leq m$), then the number of states of the *DFA* is $m + 1$ according to Lemma 3.22.

If q_m is not active, then there are 2^{m-1} ways, how states $q_1, q_2, \ldots, q_{m-1}$ can be active.

Summing up all these possibilities together we get $2^{m-2} + 2^{m-1} = \frac{3}{4}2^m$, which is the maximum number of states of *DFA*. □

One of the two worst cases according to the number of states of the *DFA* equivalent to *NFA* for the exact sequence matching (constructed by Algorithm 3.14) arises, when no symbol is used more than once in pattern $P = p_1p_2 \ldots p_m$ (i.e., $p_i \neq p_j, \forall i, j, i \neq j, 0 < i \leq m, 0 < j \leq m$) as shown in Example 3.25. If we replace p_m by p_{m-1} or vice versa (i.e., $p_i \neq p_j, \forall i, j, i \neq j, 0 < i < m, 0 < j < m$, and $p_{m-1} = p_m$), then we get the other of the two worst cases.

Example 3.25
Let $P = abcd$. We construct M_{NFA} using Algorithm 3.14 and then we transform it to M_{DFA} using Algorithm 3.2.

M_{NFA}	a	b	c	d	$\Sigma \setminus \{a, b, c, d\}$
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$	$\{q_0\}$	$\{q_0\}$
q_1	$\{q_1\}$	$\{q_2\}$	$\{q_1\}$	$\{q_1\}$	$\{q_1\}$
q_2	$\{q_2\}$	$\{q_2\}$	$\{q_3\}$	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_4\}$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

M_{DFA}	a	b	c	d	$\Sigma \setminus \{a,b,c,d\}$
$[q_0]$	$[q_0,q_1]$	$[q_0]$	$[q_0]$	$[q_0]$	$[q_0]$
$[q_0,q_1]$	$[q_0,q_1]$	$[q_0,q_2]$	$[q_0,q_1]$	$[q_0,q_1]$	$[q_0,q_1]$
$[q_0,q_2]$	$[q_0,q_1,q_2]$	$[q_0,q_2]$	$[q_0,q_3]$	$[q_0,q_2]$	$[q_0,q_2]$
$[q_0,q_1,q_2]$	$[q_0,q_1,q_2]$	$[q_0,q_2]$	$[q_0,q_1,q_3]$	$[q_0,q_1,q_2]$	$[q_0,q_1,q_2]$
$[q_0,q_3]$	$[q_0,q_1,q_3]$	$[q_0,q_3]$	$[q_0,q_3]$	$[q_0,q_4]$	$[q_0,q_3]$
$[q_0,q_1,q_3]$	$[q_0,q_1,q_3]$	$[q_0,q_2,q_3]$	$[q_0,q_1,q_3]$	$[q_0,q_1,q_4]$	$[q_0,q_1,q_3]$
$[q_0,q_4]$	$[q_0,q_1]$	$[q_0]$	$[q_0]$	$[q_0]$	$[q_0]$
$[q_0,q_2,q_3]$	$[q_0,q_1,q_2,q_3]$	$[q_0,q_2,q_3]$	$[q_0,q_3]$	$[q_0,q_2,q_4]$	$[q_0,q_2,q_3]$
$[q_0,q_1,q_4]$	$[q_0,q_1]$	$[q_0,q_2]$	$[q_0,q_1]$	$[q_0,q_1]$	$[q_0,q_1]$
$[q_0,q_1,q_2,q_3]$	$[q_0,q_1,q_2,q_3]$	$[q_0,q_2,q_3]$	$[q_0,q_1,q_3]$	$[q_0,q_1,q_2,q_4]$	$[q_0,q_1,q_2,q_3]$
$[q_0,q_2,q_4]$	$[q_0,q_1,q_2]$	$[q_0,q_2]$	$[q_0,q_3]$	$[q_0,q_2]$	$[q_0,q_2]$
$[q_0,q_1,q_2,q_4]$	$[q_0,q_1,q_2]$	$[q_0,q_2]$	$[q_0,q_1,q_3]$	$[q_0,q_1,q_2]$	$[q_0,q_1,q_2]$

If we sort the states by depth-most states we get the following table.

0	1	2	3	4
$[q_0]$	$[q_0,q_1]$	$[q_0,q_2]$	$[q_0,q_3]$	$[q_0,q_4]$
		$[q_0,q_1,q_2]$	$[q_0,q_1,q_3]$	$[q_0,q_1,q_4]$
			$[q_0,q_2,q_3]$	$[q_0,q_2,q_4]$
			$[q_0,q_1,q_2,q_3]$	$[q_0,q_1,q_2,q_4]$

The number of states of M_{NFA} is 5 and the number of states of M_{DFA} is 12.

The *NFA* presented above searches for all the overlapping occurrences of the pattern P, but the overlapping is limited. There are not allowed the overlappings such that the same symbol of the pattern is located in the same position in the text in both occurrences.

Example 3.26
Let $P = abc$ be a pattern and $T_1 = aabc$, $T_2 = abbc$, and $T_3 = abcc$ be texts. There is one overlapping occurrence in each text ($T_1 : \{\underline{a}a\underline{bc}\}$, $T_2 : \{\underline{ab}b\underline{c}\}$, and $T_3 : \{\underline{ab}c\underline{c}\}$).

It is also possible to build the *NFA* searching for all the occurrences with the unlimited overlapping. Such *NFA* has in each non-final state self-loops labeled by all symbols of input alphabet Σ. Such *NFA* for the exact sequence matching can be constructed by Algorithm 3.15 and an example of such *NFA* is shown in Figure 3.13 where $m = 4$.

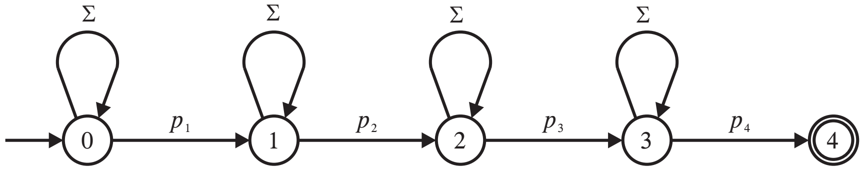


Figure 3.13: *NFA* for the exact sequence matching ($m = 4$, unlimited overlapping).

Example 3.27
Let $P = abc$ be a pattern and $T_1 = aabc$, $T_2 = abbc$, and $T_3 = abcc$ be texts. There are two occurrences with the unlimited overlapping in each text ($T_1 : \{\underline{a}a\underline{bc}, a\underline{abc}\}$, $T_2 : \{\underline{ab}b\underline{c}, \underline{a}b\underline{bc}\}$, and $T_3 : \{\underline{abc}c, \underline{ab}c\underline{c}\}$).

Algorithm 3.15 (Construction of *NFA* for the exact sequence matching (unlimited overlapping))
Input: Pattern $P = p_1p_2 \dots p_m$.
Output: *NFA* M accepting language $L(M) = \{w \mid w \in \Sigma^*, \exists\{i_1, i_2, \dots, i_m \mid i_\ell < i_{\ell+1}, 0 < \ell < m, p_j = w_{i_j}, 0 < j \le m\}\}$.
Method: *NFA* $M = (\{q_0, q_1, \dots, q_m\}, \Sigma, \delta, q_0, \{q_m\})$, where the mapping δ is constructed in the following way:

for $i \leftarrow 0, 1, \dots, m-1$ **do**
  $\delta(q_i, a) \leftarrow \{q_i\}, \forall a \in \Sigma \setminus \{p_{i+1}\}$ /* self-loop of any non-final state */
  $\delta(q_i, p_{i+1}) \leftarrow \{q_i, q_{i+1}\}$ /* forward transition */
endfor
$\delta(q_m, a) \leftarrow \emptyset, \forall a \in \Sigma$ /* no transition leads from the final state */

Note, that if any non-final state become active, it remains active forever. So if all the non-final states are active (the subsequence of length $m-1$ has been found), the behavior of such *NFA* degrades in such a way that it reports an occurrence of the pattern whenever the current symbol in the input text is p_m.

Lemma 3.28
A minimal (in number of states) *NFA* for the exact sequence matching with the unlimited overlapping has $(m+1)$ states, where m is the length of the searched pattern $P = p_1p_2 \dots p_m$.

Proof
See Algorithm 3.15.

Theorem 3.29
The minimal *DFA* for the exact sequence matching with the unlimited overlapping has $(m+1)$ states, where m is the length of the searched pattern $P = p_1p_2 \dots p_m$.

Proof
Let $M = (Q, \Sigma, \delta, q_0, F)$ be an *NFA* for the exact sequence matching with the unlimited overlapping constructed using Algorithm 3.15. Lemma 3.28 implies that this *NFA* has $(m+1)$ states and is minimal. Using Algorithm 3.2 we transform this *NFA* to the equivalent *DFA* $M' = (Q', \Sigma, \delta', q'_0, F')$, that has the same number of states. We prove by induction on the depth of state in automaton that each deterministic state q' except the final state generates one new state among the transitions leading from q'.

It holds that

$$\begin{aligned} \delta(q_0, a) &= \{q_0\}, \forall a \in \Sigma \setminus \{p_1\} && \text{/* self-loop of initial state */} \\ \delta(q_0, p_1) &= \{q_0, q_1\}. && \text{/* matching transition */} \end{aligned}$$

When the for-cycle $\mathcal{A}$ of Algorithm 3.2 is performed at first, it holds that

$$\begin{aligned} \delta'(\{q_0\}, a) &= \{q_0\}, \forall a \in \Sigma \setminus \{p_1\} \\ \delta'(\{q_0\}, p_1) &= \{q_0, q_1\}. \end{aligned}$$

Thus we obtain just one new state $q'_1 = \{q_0, q_1\} = q'_0 \cup \{q_1\}$ in the set S of not yet processed states, where $q'_0 = \{q_0\}$ is an initial state.

It holds that $\forall q_i \in Q$, $0 < i < m$,

$$\begin{aligned} \delta(q_i, a) &= \{q_i\}, \forall a \in \Sigma \setminus \{p_{i+1}\} \\ \delta(q_i, p_{i+1}) &= \{q_i, q_{i+1}\}. \end{aligned}$$

Algorithm 3.16 (Construction of *DFA* for the exact sequence matching (unlimited overlapping))
Input: Pattern $P = p_1p_2 \dots p_m$.
Output: *DFA* M accepting language $L(M) = \{w \mid w \in \Sigma^*, \exists\{i_1, i_2, \dots, i_m \mid i_\ell < i_{\ell+1}, 0 < \ell < m, p_j = w_{i_j}, 0 < j \leq m\}\}$.
Method: *DFA* $M = (\{q_0, q_1, \dots, q_m\}, \Sigma, \delta, q_0, \{q_m\})$, where the mapping δ is constructed in the following way:

for $i \leftarrow 1, 2, \dots, m$ **do**
  $\delta(q_{i-1}, p_i) \leftarrow q_i$ /* forward transition */
  $\delta(q_{i-1}, a) \leftarrow q_{i-1}$, $\forall a \in \Sigma \setminus \{p_i\}$ /* self-loop for any mismatching symbol */
endfor
$\delta(q_m, a) \leftarrow \delta(q_{m-1}, a)$, $\forall a \in \Sigma$ /* transitions of the final state */

That implies that when we determine the transitions for the new state $q'_i = \{q_0, q_1, \dots, q_i\}$ in the for-cycle $\mathcal{A}$ in Algorithm 3.2:

$$\begin{array}{lllll} \delta'(q'_i, a) & = & \delta'(q'_{i-1}, a) \cup \delta(q_i, a) & = & \{q_0, q_1, \dots, q_i\}, \forall a \in \Sigma \setminus \{p_{i+1}\} \\ \delta'(q'_i, p_{i+1}) & = & \delta'(q'_{i-1}, p_{i+1}) \cup \delta(q_i, p_{i+1}) & = & \{q_0, q_1, \dots, q_{i+1}\} \end{array}$$

then there appears just one new not yet processed state $q'_{i+1} = \{q_0, q_1, \dots, q_{i+1}\}$ in the transition for symbol p_{i+1}.

When determining transitions for final state q'_m, no new state appears because for final state q_m it holds that $\delta(q_m, a) = \emptyset$, $\forall a \in \Sigma$.

The for-cycle $\mathcal{A}$ of Algorithm 3.2 is performed $(m+1)$ times; once for the initial state (one new state is generated), $(m-1)$ times for state q'_i, $0 < i < m$, (one new state is generated in each step) and once for final state q'_m (no new state is generated).

Summing up initial state $q'_0 = \{q_0\}$, new state generated by the initial state, and $(m-1)$ states generated by states q'_i, $0 < i < m$, we get $(m+1)$ states. □

The *DFA* for the exact sequence matching with unlimited overlapping can be also constructed directly as shown in Algorithm 3.16. An example of such *DFA* is shown in Figure 3.14.

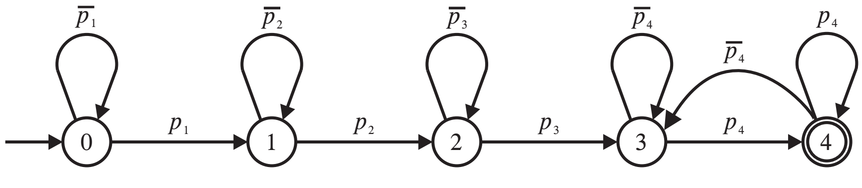


Figure 3.14: *DFA* for the exact sequence matching ($m = 4$, unlimited overlapping).

Example 3.30
Let $P = p_1p_2p_3p_4$ be a pattern, $M = (\{q_0, q_1, \dots, q_4\}, \Sigma, \delta, q_0, \{q_4\})$ be an *NFA* for the exact sequence matching with unlimited overlapping with pattern P, and $\hat{\Sigma} = \Sigma \setminus \{p_1, p_2, p_3, p_4\}$. Let us construct the *DFA* M' equivalent to *NFA* M using Algorithm 3.2. Transition table of M is as follows:

$\delta_{NFA\ M}$	p_1	p_2	p_3	p_4	$\hat{\Sigma}$
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$	$\{q_0\}$	$\{q_0\}$
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\{q_1\}$	$\{q_1\}$	$\{q_1\}$
q_2	$\{q_2\}$	$\{q_2\}$	$\{q_2, q_3\}$	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3, q_4\}$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Then we use Algorithm 3.2 and we get *DFA* M'' whose transition table is shown below.

$\delta_{DFA\ M''}$	p_1	p_2	p_3	p_4	$\hat{\Sigma}$
$[q_0]$	$[q_0, q_1]$	$[q_0]$	$[q_0]$	$[q_0]$	$[q_0]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_1]$
$[q_0, q_1, q_2]$	$[q_0, q_1, q_2]$	$[q_0, q_1, q_2]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_2]$	$[q_0, q_1, q_2]$
$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_2, q_3, q_4]$	$[q_0, q_1, q_2, q_3]$
$[q_0, q_1, q_2, q_3, q_4]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_2, q_3, q_4]$	$[q_0, q_1, q_2, q_3]$

Now, we rename states of *DFA* M'' and we get *DFA* M', whose transition table is shown below, and which is also shown in Figure 3.14.

$\delta_{DFA\ M'}$	p_1	p_2	p_3	p_4	$\hat{\Sigma}$
q_0	q_1	q_0	q_0	q_0	q_0
q_1	q_1	q_2	q_1	q_1	q_1
q_2	q_2	q_2	q_3	q_2	q_2
q_3	q_3	q_3	q_3	q_4	q_3
q_4	q_3	q_3	q_3	q_4	q_3

Note, that the *DFA* for the exact sequence matching with unlimited overlapping just only points to all occurrences of the last symbol of the pattern in an input text after a first occurrence has been found. Therefore we can use time and space optimal algorithm running in time $\mathcal{O}(n)$ and space[6] $\mathcal{O}(m)$ as shown in Algorithm 3.17.

The *FA*s for the sequence matching with the unlimited overlapping are less general than the *FA*s for the sequence matching with the limited overlapping, but their advantage is lower number of states of their equivalent *DFA*s. We can use them for:

- the exact and approximate sequence matching with the unlimited overlapping,
- the exact and approximate sequence matching without any overlapping (*FA* is restarted after finding each nonoverlapping occurrence),
- searching for first occurrence in the exact and approximate sequence matching.

*NFA*s for the approximate sequence matching can be constructed from the *NFA*s for the approximate string matching by an algorithm, which inserts self-loops (labeled either by $\overline{p}$ or by Σ) into all non-final states. It is obvious that one can easily construct algorithms constructing directly the *NFA*s for the approximate sequence matching.

NFA for the approximate sequence matching using the Hamming distance is shown in Figure 3.15, *NFA* for the approximate sequence matching using the Levenshtein distance is shown in Figure 3.16, and *NFA* for the approximate sequence matching using the generalized Levenshtein distance is shown in Figure 3.17.

[6]The space complexity does not consider the space needed for storing the input text.

Algorithm 3.17 (Exact sequence matching (unlimited overlapping))
Input: Pattern $P = p_1 p_2 \dots p_m$, text $T = t_1 t_2 \dots t_n$.
Output: First occurrence of sequence P in T.
Method:

```
i ← 1                                              /* pointer in T */
j ← 1                                              /* pointer in P */
while  i ≤ n  and j ≤ m  do
   if  t_i = p_j  then
      j ← j + 1                                    /* match */
   endif
   i ← i + 1
endwhile
if  j > m  then
   write('Sequence P ends at position i − 1 in text T.')
endif
```

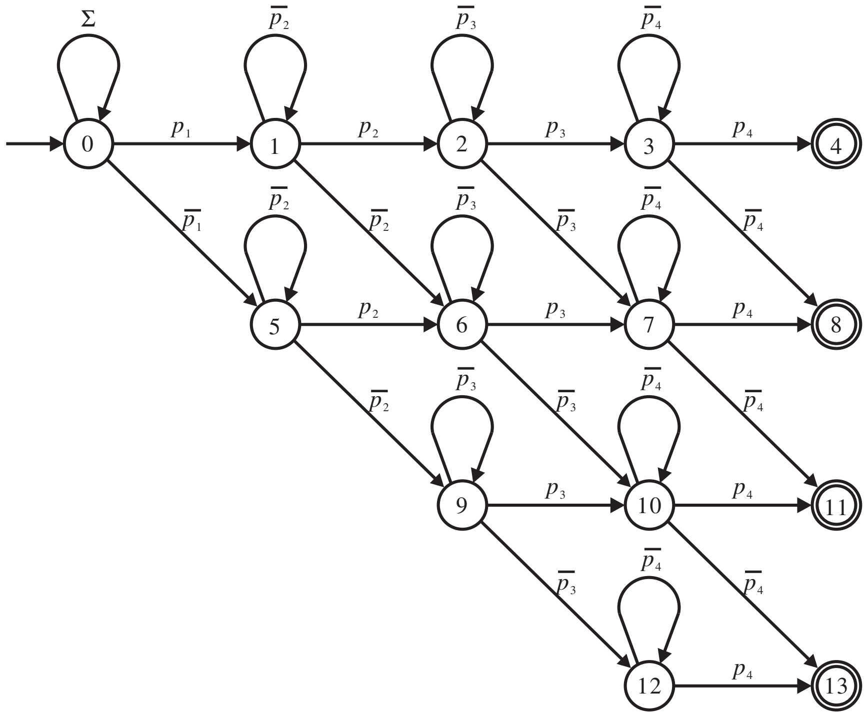


Figure 3.15: *NFA* for the approximate sequence matching using the Hamming distance ($\overline{p}$ version, $m = 4$, $k = 3$).

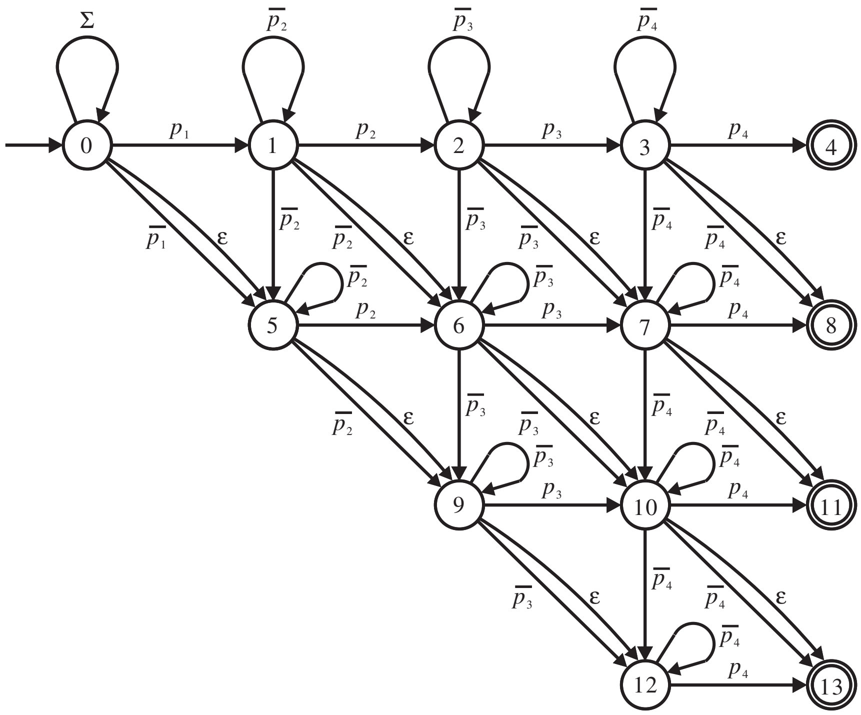


Figure 3.16: *NFA* for the approximate sequence matching using the Levenshtein distance ($\overline{p}$ version, $m = 4$, $k = 3$).

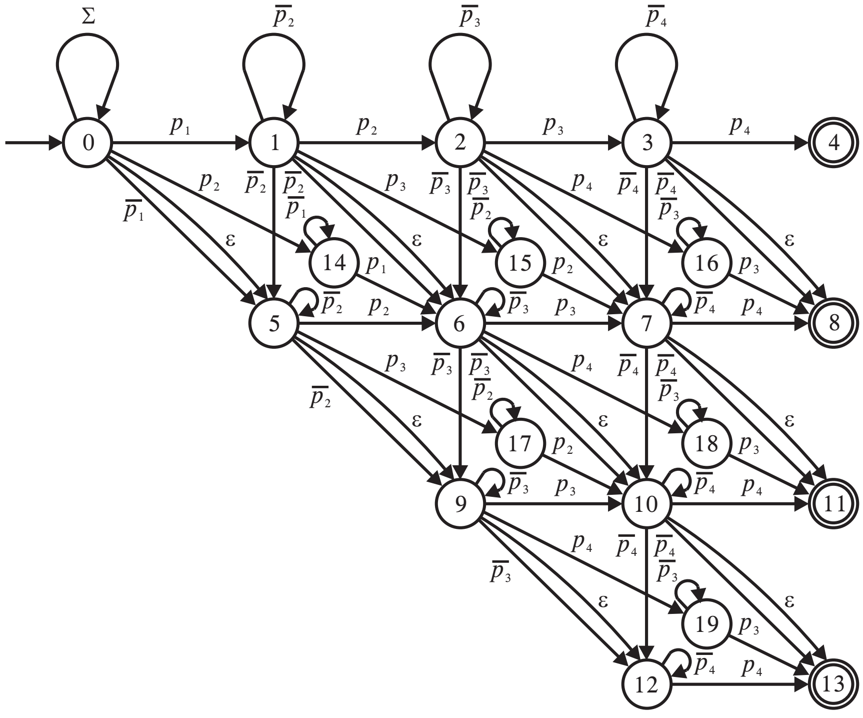


Figure 3.17: *NFA* for the approximate sequence matching using the generalized Levenshtein distance ($\overline{p}$ version, $m = 4$, $k = 3$).

3.6 Reduced Finite Automata

There are some situations, in which we want to know all the occurrences of pattern P in text T with at most k errors but we are not interested in knowing the exact number of errors in the found string. In such cases the states, which are needed only to recognize how many errors are in the found string, can be removed from *NFA*s [Hol96a]. These states are the states of depth j and level l such that $j > m - k + l$. The number of states of the reduced *NFA* for the approximate string matching using the Hamming distance as well as for the approximate string matching using the Levenshtein distance is $(m + 1 - k)(k + 1)$.

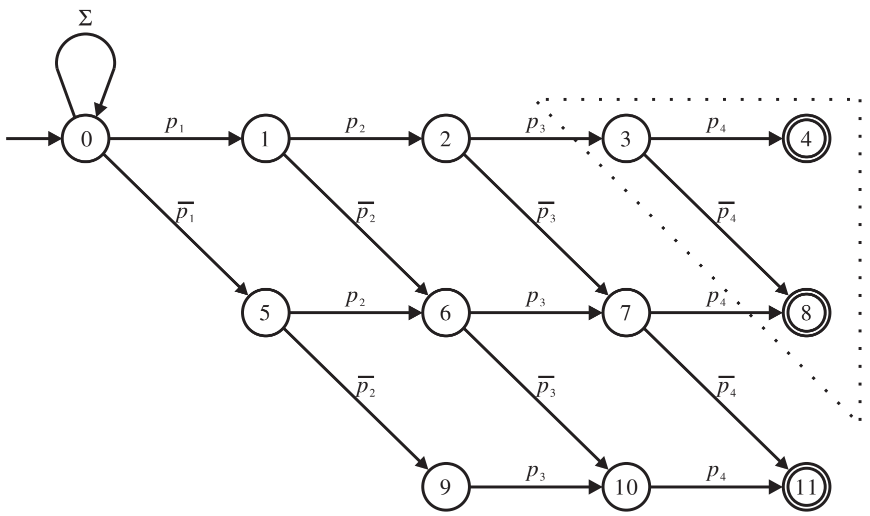


Figure 3.18: *NFA* for the approximate string matching using the Hamming distance ($m = 4$, $k = 2$).

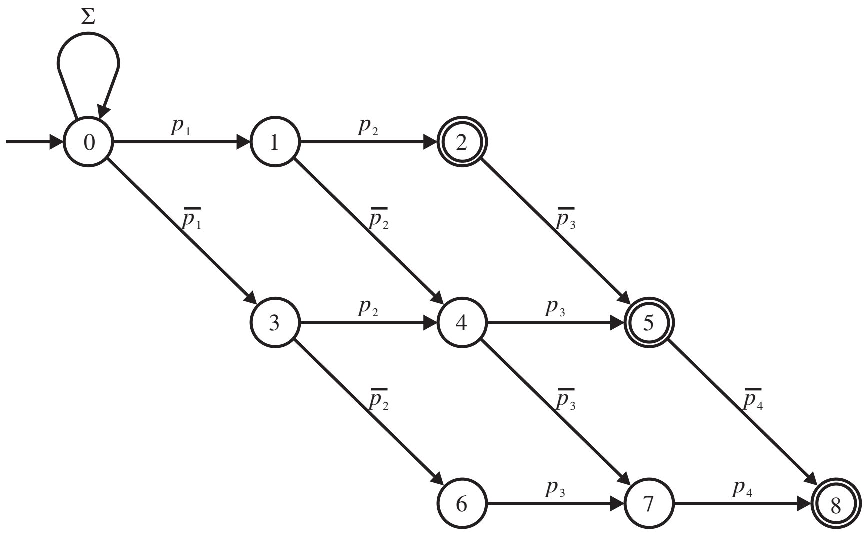


Figure 3.19: Reduced *NFA* for the approximate string matching using the Hamming distance ($m = 4$, $k = 2$).

In the *NFA* for the approximate string matching using the Hamming distance shown in Figure 3.18, the states that can be removed are bordered by the dotted line. If we remove these states, we get the reduced *NFA*, in which each state of level l and depth $m-k+l$ become the final state as shown in Figure 3.19. If such state of level l is active in step i of searching, it means that the pattern P will be found with at most k mismatches at location $t_{i+k-l-m+1}\dots t_{i+k-l}$. Moreover, we can say that $l \le D_H(P, t_{i+k-l-m+1}\dots t_{i+k-l}) \le k$. Of course, we have to check whether $i+k-l \le n$—the ending position is not located behind the end of the text T.

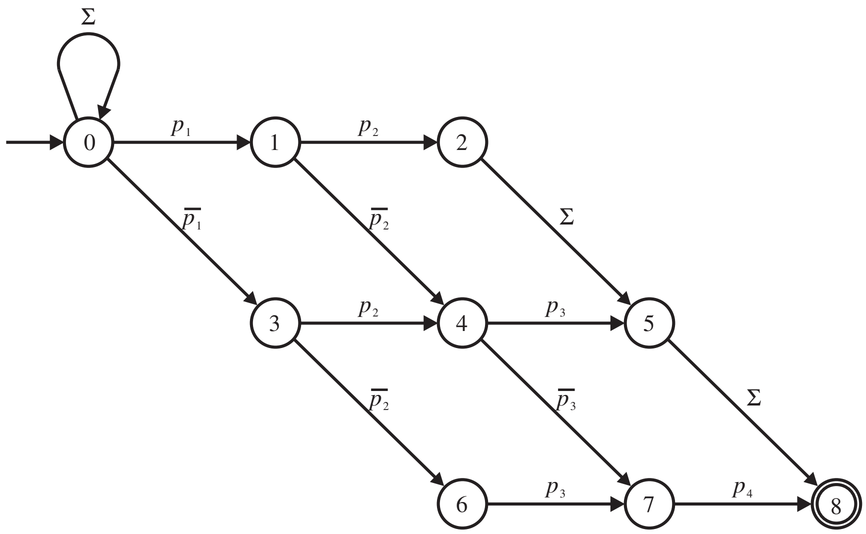


Figure 3.20: Reduced *NFA* for the approximate string matching using the Hamming distance ($m = 4$, $k = 2$).

In the reduced *NFA* for the approximate string matching using the Hamming distance we do not need each state of level l and depth $m-k+l$ to become the final state. We can connect all such states by the transitions labeled by all symbols of input alphabet Σ such that from the state of level l and depth $m-k+l$ the transitions labeled by all symbols of input alphabet Σ lead to the state of level $l+1$ and depth $m-k+l+1$, $0 \le l < k$, as shown in Figure 3.20. This *NFA* has only one final state that, if it is active in step i, reports that the pattern P has been found in text T at the position $t_{i-m+1}\dots t_i$ with at most k mismatches.

The *NFA* for the approximate string matching using the Levenshtein distance can be also reduced as described above. Such reduced *NFA* for $m = 4$ and $k = 2$ is shown in Figure 3.21. In this case we do not need to change the final states since ε-transitions make each state of level l and depth $m-k+l$ also the final state, because they lead into the same final state, which is in level k and depth m.

The *NFA* for the approximate string matching using the generalized Levenshtein distance can be also reduced as in the previous two cases. The reduced *NFA* is shown in Figure 3.22.

The reduction of states of the *NFA*s for the approximate string matching was shown for $\overline{p}$ version of *NFA*s but it is obvious that it can be used for Σ version of *NFA*s as well. We can also apply this reduction to the *NFA*s for the approximate sequence matching.

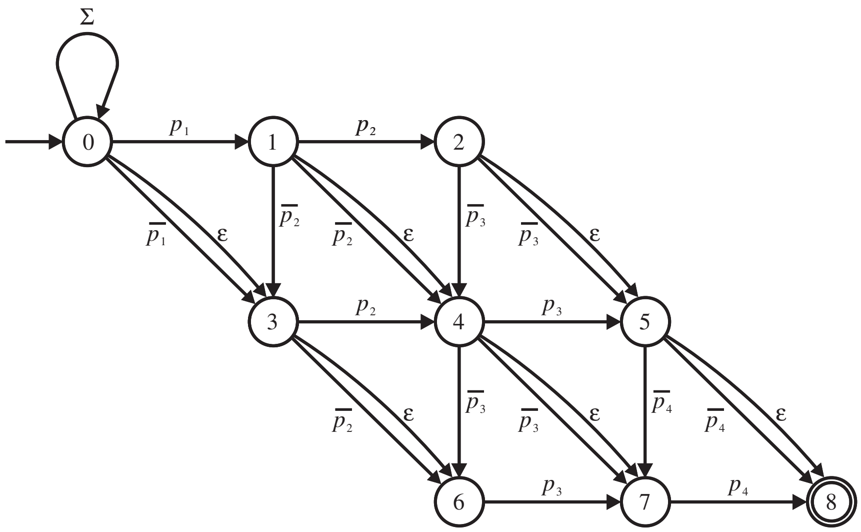


Figure 3.21: Reduced *NFA* for the approximate string matching using the Levenshtein distance ($m = 4$, $k = 2$).

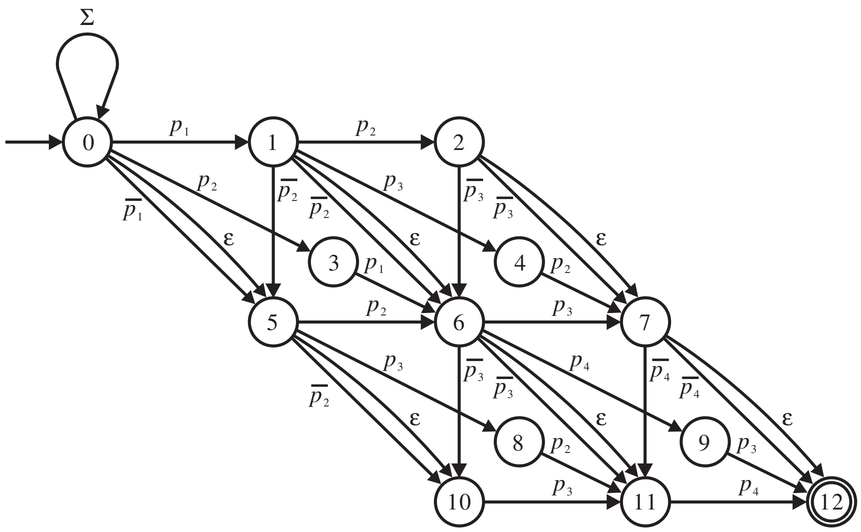


Figure 3.22: Reduced *NFA* for the approximate string matching using the generalized Levenshtein distance ($m = 4$, $k = 2$).

Algorithm 3.18 (Exact string verification—naive method)
Input: Pattern $P = p_1p_2 \dots p_m$, text $T = t_1t_2 \dots t_n$, $m \leq n$.
Output: Yes, if string P is at the beginning of T, No, otherwise.
Method:

```
i ← 1                                   /* pointer in T and P */
while  i ≤ m  do
  if  t_i = p_i  then
    i ← i + 1                           /* match */
  else
    return('No.')
  endif
endwhile
return('Yes.')
```

3.7 String and Sequence Verification

In our *NFA*s for the string and sequence matching we skip the symbols preceding the occurrences by the self-loop in the initial state. If we remove such self-loop, we get *NFA*s that verify whether the required occurrence is located at the beginning of the text.

Note, that in such case, we cannot remove the states under the diagonal leading from the initial state (Theorem 3.14) in case of the approximate string and sequence matching using the Levenshtein and generalized Levenshtein distances. These states are needed for edit operation *insert* performed at the beginning of comparing—up to k symbols can be inserted in front of the pattern.

In case of exact pattern verification, the use of *DFA*s is less effective than the use of naive comparison method. Algorithm 3.18 shows the naive comparison method for the exact string verification and Algorithm 3.19[7] shows the naive comparison method for the exact sequence verification.

In case of approximate pattern matching, the naive comparison method is not effective and therefore *FA* approach is more suitable.

[7]This algorithm differs from Algorithm 3.17 only in such a way, that it tests, whether the first symbol of the pattern is located at the beginning of the input text.

Algorithm 3.19 (Exact sequence verification)
Input: Pattern $P = p_1p_2 \dots p_m$, text $T = t_1t_2 \dots t_n$, $m \leq n$.
Output: Yes, if sequence P is at the beginning of T, No, otherwise.
Method:

```
if  t_1 ≠ p_1  then
  return('No.')                          /* mismatch of the first symbols */
endif
i ← 2                                    /* pointer in T */
j ← 2                                    /* pointer in P */
while  i ≤ n  and j ≤ m  do
  if  t_i = p_j  then
    j ← j + 1                            /* match */
  endif
  i ← i + 1
endwhile
if  j > m  then
  return('Yes.')
else
  return('No.')
endif
```

3.8 Nonuniform Weights of Edit Operations

In the previous sections we have considered constant weights for all edit operations (*replace*, *insert*, *delete*, and *transpose*). But we can also assign different positive integer weight to each edit operation. If we do so, we get the weighted approximate string and sequence matching, which is defined as a searching for all the occurrences of a pattern in an input text that have their costs lower or equal to some limit k, where the cost is the minimum sum of weights of edit operations needed to convert the pattern to the found string.

We construct *NFA* for the weighted approximate string matching using the Levenshtein distance ($\overline{p}$ version) for pattern $P = p_1p_2 \dots p_m$ and limit k. Let w_R be a weight of edit operation *replace*, w_I be a weight of edit operation *insert*, and w_D be a weight of edit operation *delete*. To construct *NFA* for the weighted approximate string matching we use $k+1$ copies of *NFA* for the exact string matching $(M_0, M_1, \dots, M_k)$. Now, we insert transitions for edit operations *replace*, *insert*, and *delete*. From each state q_i (in depth i, $0 \leq i < m$) of automaton M_j (level j of the resulting automaton, $0 \leq j < k$) a transition *replace* labeled by $\overline{p}_{i+1}$ leads to state q_{i+1} of automaton M_{j+w_R} (if $j + w_R \leq k$), a transition *insert* labeled by $\overline{p}_{i+1}$ leads to state q_i of automaton M_{j+w_I} (if $j + w_I \leq k$), and a transition *delete* labeled by ε leads to state q_{i+1} of automaton M_{j+w_D} (if $j + w_D \leq k$). Then we remove all inaccessible states. The initial state of the resulting *NFA* is the initial state of automaton M_0. An example of such *NFA* for $m = 4$, $k = 3$, $w_R = 1$, $w_I = 2$, $w_D = 2$ is shown in Figure 3.23.

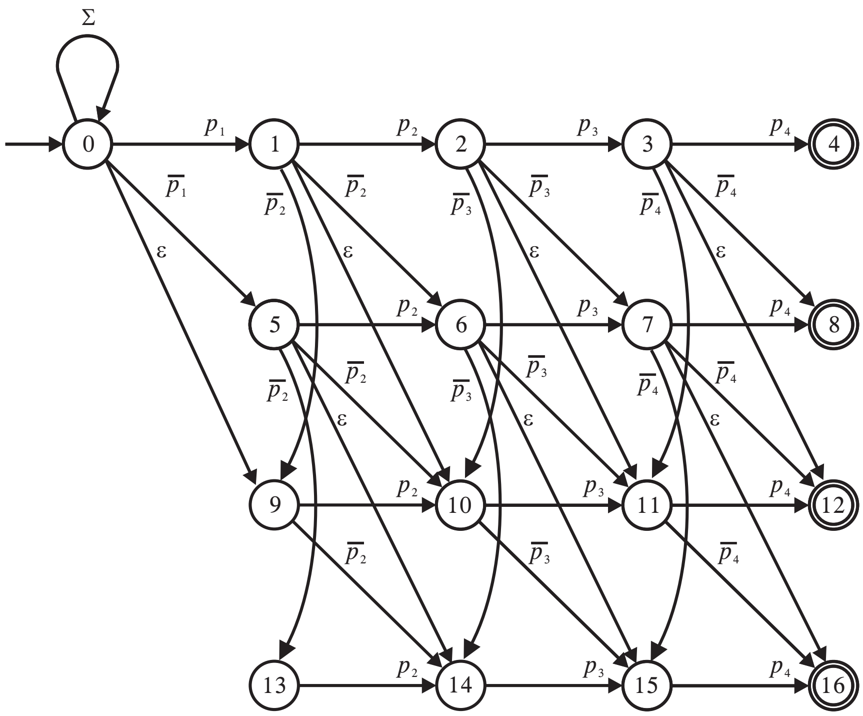


Figure 3.23: *NFA* for the weighted approximate string matching using the Levenshtein distance ($m = 4$, $k = 3$, $w_R = 1$, $w_I = 2$, $w_D = 2$).

The equivalent *NFA* without ε-transitions can be also constructed by Algorithm 3.20. Note that this algorithm can insert inaccessible states, e.g., if all weights are greater than one, all states

---

Algorithm 3.20 (Construction of *NFA* for the weighted approximate string matching using the Levenshtein distance)
Input: Pattern $P = p_1p_2 \dots p_m$, limit k, w_R, w_I, w_D.
Output: *NFA* M for the weighted approximate string matching using the Levenshtein distance.
Method: *NFA* $M = (Q, \Sigma, \delta, q_0^0, F)$, where Q, δ, and F are constructed in the following way:

```
Q ← {q_i^j | 0 < i ≤ m, 0 ≤ j ≤ k} ∪ {q_0^0}
for j ← k, k−1, ..., 0 do
  for i ← m, m−1, ..., 0 do
    if j > 0 or (j = 0 and i = 0) then                        /* q_i^j ∈ Q? */
      δ(q_i, a) ← ∅, ∀a ∈ Σ                                   /* initial setting */
      if i = m then
        F ← F ∪ {q_i}                                          /* final state */
      else
        δ(q_i^j, p_{i+1}) ← {q_{i+1}^j}                        /* matching transition */
        if j + w_R ≤ k then
          δ(q_i^j, a) ← δ(q_i^j, a) ∪ {q_{i+1}^{j+w_R}}, ∀a ∈ Σ \ {p_{i+1}}   /* replace transition */
        endif
        if j + w_I ≤ k then
          δ(q_i^j, a) ← δ(q_i^j, a) ∪ {q_i^{j+w_I}}, ∀a ∈ Σ \ {p_{i+1}}       /* insert transition */
        endif
        if j + w_D ≤ k then
          δ(q_i^j, a) ← δ(q_i^j, a) ∪ δ(q_{i+1}^{j+w_D}), ∀a ∈ Σ \ {p_{i+1}}  /* delete transition */
        endif
      endif
    endif
  endfor
endfor
δ(q_0^0, a) ← δ(q_0^0, a) ∪ {q_0^0}, ∀a ∈ Σ                    /* self-loop of the initial state */
```

---

of level 1 are inaccessible. Therefore we should remove inaccessible states from such *NFA*, but if we transform such *NFA* to the equivalent *DFA* using Algorithm 3.2, the inaccessible states would be removed directly by Algorithm 3.2.

The weighted approximate string matching using the Hamming distance makes no sense (only *replace* edit operation is allowed) but for the weighted approximate string matching using the generalized Levenshtein distance we have to consider also a weight w_T for edit operation *transpose* in the same way like described above for *replace*, *insert*, and *delete* transitions.

Note, that if $w_T \geq 2w_R$, edit operation *transpose* makes no sense to be considered. Also if $w_R \geq w_I + w_D$, edit operation *replace* makes no sense to be considered.

Another possibility for weights of edit operations is that the weights vary with respect to the symbols involved in these operations. When replacing symbol a by symbol b, some weight $w_R(a, b)$ is considered [BC92] ($w_R(a, a) = 0$), when inserting symbol a to the pattern, some weight $w_I(a)$ is considered, and when deleting symbol a from the pattern, some weight $w_D(a)$ is considered

[WF74, LW75, Kur96]. The weights can also vary with respect to the position in the pattern. All these extensions can be covered by the approach described above.

The construction of *NFA*s for the weighted approximate sequence matching is almost the same like for the weighted approximate string matching and therefore is is not further considered in this thesis.

Chapter 4

Basic Simulation Method

In the case when the space complexity or the preprocessing time of *DFA* makes it unusable, we can use some of deterministic simulation methods of corresponding *NFA*. In this chapter we describe a basic simulation method, which is the base of the other simulation methods presented further in this thesis. This method can be used for general *NFA*. The other simulation methods improve complexities of simulation, but on the other hand they set some requirements to *NFA*s in order to be more efficient.

4.1 Simulation of Nondeterministic Finite Automata

We remind that *NFA* accepts an input text w, $w \in \Sigma^*$, if and only if there exists a sequence of transitions from the initial state to one of the final states, labeled by symbols of w in the corresponding order (i.e., $(q_0, w) \vdash^*_{NFA} (q_f, \varepsilon)$, $q_f \in F$). So in each step of the run, *NFA* must predict the correct transition leading to the correct final state (i.e., the final state accepting w, if such final state exists) and then use it. We do not consider any prediction method in this thesis therefore we have to check all the possible paths through *NFA* for the input word w leading from the initial state in order to be sure if there exists a path accepting w or not. These paths can be checked using the width-first method or the depth-first method. In this thesis we consider only the width-first method.

In Algorithm 4.1 we show the basic algorithm, which is very similar to the transformation of *NFA* to the equivalent *DFA* in Algorithm 3.2. In each step of the run of *NFA* a new set of active states is computed by evaluating all transitions from all states of the previous set of active states. This provides that all possible paths labeled by input string are considered in *NFA*. The simulation finishes when the end of the input text is reached or when there is no active state (i.e., no accepting path exists).

In the transformation of *NFA* to the equivalent *DFA* shown in Algorithm 3.2, all the possible configurations of set S of active states are evaluated as well as all the possible transitions among these configurations and a deterministic state is assigned to each such configuration of S. Using the simulation method from Algorithm 4.1 only the current configuration of set S is evaluated in each step of the simulation of *NFA*—only used configurations are evaluated during the simulation.

It is also possible to combine the simulation and the transformation. When processing an input text, we can store the used configurations of S in some state-cache and assign them deterministic states. In such a way we transform *NFA* to *DFA* incrementally, but we evaluate only used states and transitions. If the state-cache is full, we can use one of cache techniques for making room in the state-cache, e.g., removing least recently used states.

Algorithm 4.1 (Simulation of run of *NFA*—basic method)
Input: *NFA* $M = (Q, \Sigma, \delta, q_0, F)$, input text $T = t_1 t_2 \ldots t_n$.
Output: Output of run of *NFA*.
Method: Set S of active states is used.

$S \leftarrow \varepsilon CLOSURE(\{q_0\})$
$i \leftarrow 1$
while $i \leq n$ **and** $S \neq \emptyset$ **do**
  $S \leftarrow \bigcup_{q \in S} \varepsilon CLOSURE(\delta(q, t_i))$ /* transitions are performed for all elements of S */
  if $S \cap F \neq \emptyset$ **then**
    write(information associated with each final state in $S \cap F$)
  endif
  $i \leftarrow i + 1$
endwhile

This solution has the following advantages: we can control the amount of used memory (size of state-cache) and for the most frequent configurations we do not need always to compute the most frequent transitions, but we can use directly the information stored in the state-cache (using a *DFA* transition has better time complexity than computing a new configuration).

Theorem 4.1
The basic simulation method shown in Algorithm 4.1 simulates run of *NFA*.

Proof
Let $M = (Q, \Sigma, \delta, q_0, F)$ be an *NFA* and $T = t_1 t_2 \ldots t_n$ be an input text. Algorithm 4.1 considers really all paths (i.e., sequences of configurations) leading from q_0.

At the beginning of the algorithm, set S of active states contains $\varepsilon CLOSURE(\{q_0\})$—each path must start in q_0 and then some of ε-transitions leading from q_0 can be used. In this way all configurations (q_j, T), $q_j \in Q$, $(q_0, T) \vdash_M^* (q_j, T)$, are considered.

In each i-th step of the algorithm, $1 \leq i \leq n$, all transitions (relevant to T) leading from all states of S are evaluated, i.e., both t_i labeled transitions as well as ε-transitions. At first all transitions reachable by transitions labeled by t_i from each state of S are inserted in new set S and then $\varepsilon CLOSURE$ of new set S is also inserted in new set S. In this way all configurations $(q_j, t_{i+1} t_{i+2} \ldots t_n)$, $q_j \in Q$, $(q_0, T) \vdash_M^* (q_j, t_{i+1} t_{i+2} \ldots t_n)$, are considered in i-th step of the algorithm.

In each step of the algorithm, the set S of active states is tested for final state and for each such found active final state the information associated with such state is reported. $\square$

4.2 Implementation

4.2.1 *NFA* without ε-transitions

This basic method can be implemented using bit vectors. Let $M = (Q, \Sigma, \delta, q_0, F)$ be an *NFA* without ε-transitions and $T = t_1 t_2 \ldots t_n$ be an input text of M. We implement a transition function δ as a transition table $\mathcal{T}$ (of size $|Q| \times |\Sigma|$) of bit vectors:

$$\mathcal{T}[i,a] = \begin{bmatrix} \tau_0 \\ \tau_1 \\ \vdots \\ \tau_{|Q|-1} \end{bmatrix} \tag{4.1}$$

where $a \in \Sigma$ and bit $\tau_j = 1$, if $q_j \in \delta(q_i, a)$, or $\tau_j = 0$ otherwise. Then we also implement a set F of final states as a bit vector $\mathcal{F}$:

$$\mathcal{F} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{|Q|-1} \end{bmatrix} \tag{4.2}$$

where bit $f_j = 1$, if $q_j \in F$, or $f_j = 0$ otherwise. In each step i, $0 \leq i \leq n$, (i.e., after reading symbol t_i) of the simulation of the *NFA* run, the set S of active states is represented by bit vector $\mathcal{S}$:

$$\mathcal{S}_i = \begin{bmatrix} s_{0,i} \\ s_{1,i} \\ \vdots \\ s_{|Q|-1,i} \end{bmatrix} \tag{4.3}$$

where bit $s_{j,i} = 1$, if state q_j is active (i.e. $q_j \in \mathcal{S}$) in i-th simulation step, or $s_{j,i} = 0$ otherwise.

When constructing $\mathcal{S}_{i+1}$, we can evaluate all the transitions labeled by t_{i+1} leading from state q_j, which is active in i-th step, at once—just using the bitwise operation OR for bit-vector $\mathcal{S}_{i+1}$ and bit-vector $\mathcal{T}[j, t_{i+1}]$. This implementation is used in Algorithm 4.2.

Note, that the first **for** cycle of Algorithm 4.2 is a multiplication of vector $\mathcal{S}_i$ by matrix $\mathcal{T}[*, t_i]$. This approach is also used in quantum automata [MC97], where quantum computation is used.

Theorem 4.2
The simulation of the run of general *NFA* runs in time $\mathcal{O}(n|Q|\lceil\frac{|Q|}{w}\rceil)$ and space[1] $\mathcal{O}(|\Sigma||Q|\lceil\frac{|Q|}{w}\rceil)$, where n is the length of the input text, $|Q|$ is the number of states of the *NFA*, Σ is the input alphabet, and w is the size of the used computer word in bits.

Proof
See the basic simulation method in Algorithm 4.2. The main while-cycle is performed at most n-times and both inner while-cycles are performed just $|Q|$ times. If $|Q| > w$ (i.e., more than one computer word must be used to implement bit-vector for all states of *NFA*), each elementary bitwise operations must be split into $\lceil\frac{|Q|}{w}\rceil$ bitwise operations. It gives us the time complexity $\mathcal{O}(n|Q|\lceil\frac{|Q|}{w}\rceil)$. The space complexity is given by the implementation of transition function δ by transition table $\mathcal{T}$, which contains $(|Q| \times |\Sigma|)$ bit-vectors each of size $\lceil\frac{|Q|}{w}\rceil$. □

Example 4.3
Let M be an *NFA* for the exact string matching for pattern $P = aba$ and $T = accabcaaba$ be an input text. M is shown in Figure 4.1.

[1] For the space complexity of this theorem we expect full transition table for representation of transition function.

Algorithm 4.2 (Simulation of run of *NFA*—bit-vector implementation of basic method)
Input: A transition table $\mathcal{T}$ and a set $\mathcal{F}$ of final states of *NFA*, input text $T = t_1 t_2 \dots t_n$.
Output: Output of run of *NFA*.
Method:

```
S_0 ← [100...0]                                   /* only q_0 is active at the beginning */
i ← 1
while  i ≤ n  and S_{i−1} ≠ [00...0]  do
   S_i ← [00...0]
   for  j ← 0, 1, ... |Q| − 1  do
      if  s_{j,i−1} = 1  then                     /* q_j is active in (i − 1)-th step */
         S_i ← S_i OR T[j, t_i]                   /* evaluate transitions for q_j */
      endif
   endfor
   for  j ← 0, 1, ... |Q| − 1  do
      if  s_{j,i} = 1  and  f_j = 1  then         /* if q_j is active final state */
         write(information associated with final state q_j)
      endif
   endfor
   i ← i + 1
endwhile
```

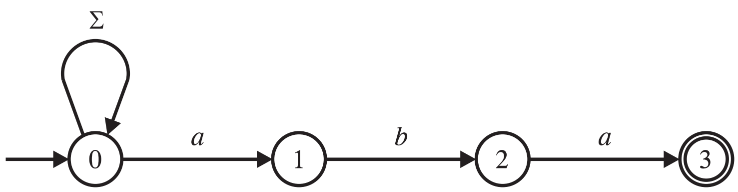


Figure 4.1: *NFA* for the exact string matching ($P = aba$).

Transition table δ and its bit-vector representation $\mathcal{T}$ are as follows:

δ_M	a	b	$\Sigma \setminus \{a,b\}$
0	{0,1}	{0}	{0}
1	$\emptyset$	{2}	$\emptyset$
2	{3}	$\emptyset$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

$\mathcal{T}_M$	a	b	$\Sigma \setminus \{a,b\}$
0	1	1	1
	1	0	0
	0	0	0
	0	0	0
1	0	0	0
	0	0	0
	0	1	0
	0	0	0
2	0	0	0
	0	0	0
	0	0	0
	1	0	0
3	0	0	0
	0	0	0
	0	0	0
	0	0	0

The process of simulation of M over T is displayed by set S of active states and its bit-vector representation $\mathcal{S}$.

	-	a	c	c	a	b	c	a	a	b	a
$\mathcal{S}$	1	1	1	1	1	1	1	1	1	1	1
	0	1	0	0	1	0	0	1	1	0	1
	0	0	0	0	0	1	0	0	0	1	0
	0	0	0	0	0	0	0	0	0	0	1
S	q_0	q_0 q_1	q_0	q_0	q_0 q_1	q_0 q_2	q_0	q_0 q_1	q_0 q_1	q_0 q_2	q_0 q_1 q_3

4.2.2 *NFA* with ε-transitions

If we have an *NFA* with ε-transitions, we can transform it to an equivalent *NFA* without ε-transitions using Algorithm 4.3. There are also other algorithms for removing ε-transitions but this algorithm does not change the states of the *NFA*. Note, that Algorithm 3.2 not only removes ε-transitions but it also constructs the equivalent *DFA*.

Algorithm 4.3 provides that all transitions labeled by symbols (not ε-transitions) leading from all states of $\varepsilon CLOSURE(\{q_0\})$ lead also from q_0. Then for each state $q \in Q$ and symbol $a \in \Sigma$ all states accessible from $\delta(q,a)$ (i.e., $\varepsilon CLOSURE(\delta(q,a))$) are inserted into $\delta'(q,a)$.

Note, that the resulting *NFA* M' can contain inaccessible states. Each state q, $q \in S \setminus \{q_0\}$, of *NFA* M' is inaccessible if the only incoming transitions into this state q in *NFA* M are the ε-transitions leading from q_0.

The ε-transitions can also be removed directly during a construction of *NFA* as shown in Chapter 3.

There is also other possibility of simulation of *NFA* with ε-transitions. We can implement ε-transitions by a table $\mathcal{E}$ (of size $|Q|$) of bit-vectors:

Algorithm 4.3 (Removing ε-transitions from *NFA*)
Input: *NFA* $M = (Q, \Sigma, \delta, q_0, F)$ with ε-transitions.
Output: *NFA* $M' = (Q, \Sigma, \delta', q_0, F')$ without ε-transitions.
Method:

$S \leftarrow \varepsilon CLOSURE(\{q_0\})$
for each $a \in \Sigma$ **do**
  $\delta'(q_0, a) \leftarrow \bigcup_{q \in S} \varepsilon CLOSURE(\delta(q, a))$
endfor
for each $q \in Q \setminus \{q_0\}$ **do**
  for each $a \in \Sigma$ **do**
    $\delta'(q, a) \leftarrow \varepsilon CLOSURE(\delta(q, a))$
  endfor
endfor
if $S \cap F \neq \emptyset$ **then**
  $F' \leftarrow F \cup \{q_0\}$
else
  $F' \leftarrow F$
endif

$$\mathcal{E}[i] = \begin{bmatrix} e_0 \\ e_1 \\ \vdots \\ e_{|Q|-1} \end{bmatrix} \tag{4.4}$$

where bit $e_j = 1$, if $q_j \in \varepsilon CLOSURE(\{q_i\})$, or $e_j = 0$ otherwise.

This implementation is used in Algorithm 4.4, where $\varepsilon CLOSURE(S)$ is computed in each step of the simulation. The time and space complexities are asymptotically same as in Algorithm 4.2, therefore Theorem 4.2 holds for all *NFA*s (with as well as without ε-transitions).

Lemma 4.4
Let *NFA* $M = (Q, \Sigma, \delta, q_0, F)$ is implemented by bit-vectors as shown in the previous section. Then Algorithm 4.3 runs in time $\mathcal{O}(|\Sigma||Q|^2\lceil\frac{|Q|}{w}\rceil)$ and space $\mathcal{O}(|\Sigma||Q|\lceil\frac{|Q|}{w}\rceil)$, where w is a length of used computer word in bits.

Proof
Let ε-transitions be implemented by table $\mathcal{E}$ as shown above (See Formula 4.4).

The statement in the first **for** cycle of Algorithm 4.3 contains the cycle performed for all states of S ($\mathcal{O}(|Q|)$), in which there is nested another **for** cycle performed for all states of $\delta(q, a)$ ($\mathcal{O}(|Q|)$).

The statement inside next two nested **for** cycles ($\mathcal{O}(|Q|)$ and $\mathcal{O}(\Sigma)$) also contains a cycle ($\varepsilon CLOSURE()$ for all states of $\delta(q, a)$—$\mathcal{O}(|Q|)$).

Therefore the total time complexity is $\mathcal{O}(|\Sigma||Q|^2\lceil\frac{|Q|}{w}\rceil)$. The space complexity is given by the size of new copy M' of *NFA* and no other space is needed. □

Algorithm 4.4 (Simulation of run of *NFA* with ε-transitions—bit-vector implementation of basic method)
Input: Transition tables $\mathcal{T}$ and $\mathcal{E}$, and a set $\mathcal{F}$ of final states of *NFA*, input text $T = t_1t_2 \dots t_n$.
Output: Output of run of *NFA*.
Method:

```
S_0 ← [100...0] OR E[0]          /* only εCLOSURE({q_0}) is active at the beginning */
i ← 1
while  i ≤ n  and S_{i-1} ≠ [00...0]  do
   S_i ← [00...0]
   for  j ← 0,1,...|Q| − 1  do
      if  s_{j,i-1} = 1  then                /* q_j is active in (i − 1)-th step */
         S_i ← S_i OR T[j, t_i]              /* evaluate transitions for q_j */
      endif
   endfor
   for  j ← 0,1,...|Q| − 1  do               /* construct εCLOSURE(S_i) */
      if  s_{j,i} = 1  then
         S_i ← S_i OR E[j]
      endif
   endfor
   for  j ← 0,1,...|Q| − 1  do
      if  s_{j,i} = 1  and f_j = 1  then     /* if q_j is active final state */
         write(information associated with final state q_j)
      endif
   endfor
   i ← i + 1
endwhile
```

The basic simulation method presented in this chapter can be used for any *NFA* and it runs in time $\mathcal{O}(n|Q|\lceil\frac{|Q|}{w}\rceil)$ and space $\mathcal{O}(|\Sigma||Q|\lceil\frac{|Q|}{w}\rceil)$, where w is a length of used computer word in bits. The other simulation methods in this thesis attempt to improve the time and space complexity, but they cannot be used for general *NFA*.

Chapter 5

Dynamic Programming

The dynamic programming is a general technique widely used in various branches of computer science. It was also utilized in the approximate string matching using the Hamming distance [Mel95] (and [GL89] for detection of all permutations of pattern with at most k errors in text) and in the approximate string matching using the Levenshtein distance [WF74, Sel80, Ukk85, LV88, Mel95]. In [WF74, Sel80, Ukk85, LV88] they consider also trivial occurrences.

In this chapter we describe how the dynamic programming simulates the *NFA*s for the approximate string matching using the Hamming and Levenshtein distances. In the dynamic programming, the set of active states is represented by a vector of integer variables. Then we show how one can use this simulation method for the other *NFA*s described in Chapter 3. At the end of this chapter we try to set some rules for using this simulation method to general *NFA*s.

5.1 Algorithm

The dynamic programming for the string and sequence matching computes matrix D of size $(m+1) \times (n+1)$. Each element $d_{j,i}$, $0 \leq j \leq m$, $0 < i \leq n$, usually contains the edit distance between the string ending at i-th position in text T and the prefix of pattern P of length j.

5.2 String Matching

5.2.1 Exact String Matching

The dynamic programming for the exact string matching is the same as the dynamic programming for the approximate string matching using the Hamming distance in which $k = 0$. See the below section.

5.2.2 Approximate String Matching Using Hamming Distance

In the approximate string matching using the Hamming distance each element $d_{j,i}$, $0 < i \leq n$, $0 \leq j \leq m$, contains the Hamming distance between string $t_{i-j+1} \ldots t_i$ and string $p_1 \ldots p_j$. Elements of matrix D are computed as follows [Mel95]:

$$\begin{array}{lcll} d_{j,0} & \leftarrow & j, & 0 \leq j \leq m \\ d_{0,i} & \leftarrow & 0, & 0 \leq i \leq n \\ d_{j,i} & \leftarrow & \textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} \textbf{ else } d_{j-1,i-1} + 1, & 0 < i \leq n, 0 < j \leq m \end{array} \tag{5.1}$$

Algorithm 5.1 (Simulation of run of *NFA*—dynamic programming)
Input: Pattern $P = p_1p_2 \dots p_m$, input text $T = t_1t_2 \dots t_n$, maximum number of errors allowed k, $k < m$.
Output: Output of run of *NFA*.
Method:

```
Compute 0-th column of matrix D
for  i ← 1, 2, ..., n  do
   Compute i-th column of matrix D for input symbol t_i
   if  d_{m,i} ≤ k  then
      write('Pattern P with d_{m,i} errors ends at position i in text T.')
   endif
endfor
```

In Formula 5.1 term $d_{j-1,i-1}$ represents matching—position i in text T is increased, position j in pattern P is increased and edit distance d is the same. Term $d_{j-1,i-1} + 1$ represents edit operation *replace*—position i in text T is increased, position j in pattern P is increased and edit distance d is increased. The value of $d_{0,i}$, $0 \le i \le n$, is set to 0, because the Hamming distance between two empty strings (the prefix of length 0 of the pattern and string of length 0 ending at position i) is 0. The value of $d_{j,0}$, $0 \le j \le m$, is set to j in [Mel95].

Pattern P is found with at most k mismatches at position $t_{i-m+1} \dots t_i$ if $d_{m,i} \le k$, $m \le i \le n$. The maximum number of mismatches of the found string is $D_H(P, t_{i-m+1} \dots t_i) = d_{m,i}$. Since the Hamming distance compares two strings of equal length, element $d_{j,i}$ is valid only if $j \le i$—the pattern cannot start in front of the first symbol of T. We can remove such condition by setting $d_{j,0}$, $0 < j \le m$, to $k+1$, where k is the maximum number of errors (maximum Hamming distance) allowed in occurrences. In such a way it holds $d_{j,i} > k$, $\forall i, j$, $0 \le i < m$, $i < j \le m$, so all the items not satisfying condition $j \le i$ exceed the maximum acceptable value k.

This solution is shown in Formula 5.2. The only disadvantage of this solution is that at the beginning of the processing the values of integer vector can reach value $k + 1 + m$ (after first m steps).

$$
\begin{array}{lcll}
d_{j,0} & \leftarrow & k+1, & 0 < j \le m \\
d_{0,i} & \leftarrow & 0, & 0 \le i \le n \\
d_{j,i} & \leftarrow & \textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} \textbf{ else } d_{j-1,i-1}+1, & 0 < i \le n, 0 < j \le m
\end{array}
\qquad (5.2)
$$

Algorithm 5.1 shows the use of matrix D in the approximate string matching.

An example of matrix D using Formula 5.2 for searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ with $k = 3$ is shown in Table 5.1.

Since we are interested in at most k errors in the found string, each value of $d_{i,j}$ greater than $k+1$ can be replaced by value $k+1$ and represents that the value is greater than k. It is useful in some implementations, since we need just $\lceil \log_2(k+2) \rceil$ bits for each number.

Theorem 5.1
The dynamic programming algorithm described by Formula 5.2 simulates a run of the *NFA* for the approximate string matching using the Hamming distance constructed by Algorithm 3.8 ($\overline{p}$ version) as well as that one constructed by Algorithm 3.9 (Σ version).

D	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	4	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
d	4	5	0	2	2	1	2	2	1	1	2	0	2	2	2	2
b	4	5	6	1	3	2	2	3	3	1	2	3	0	2	3	3
b	4	5	6	7	2	3	3	3	4	3	2	3	3	0	3	4
c	4	5	6	6	8	3	3	4	4	5	4	3	4	4	0	4
a	4	4	6	7	6	9	4	3	4	5	5	5	4	5	5	0

Table 5.1: Matrix D for pattern $P = adbbca$, text $T = adcabcaabadbbca$, and $k = 3$ using the Hamming distance.

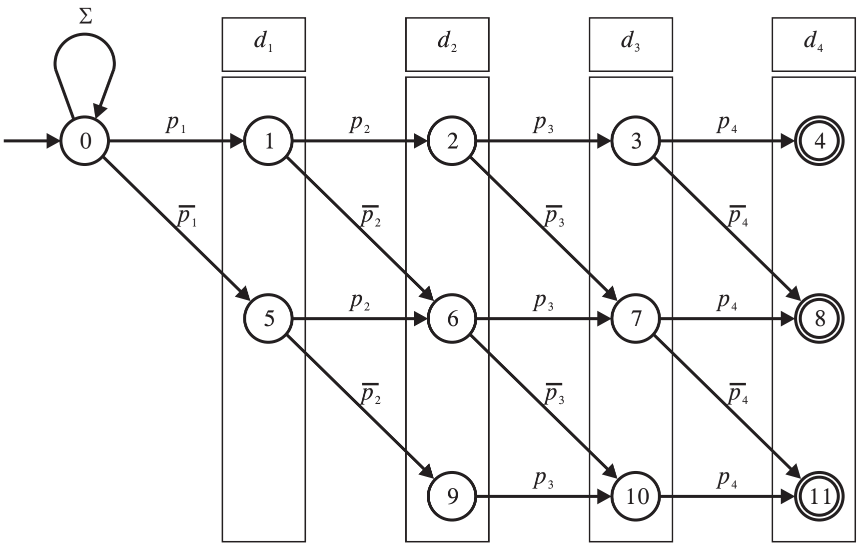


Figure 5.1: Dynamic programming uses for each depth of states of *NFA* one integer variable d.

Proof

In the dynamic programming for the approximate string matching using the Hamming distance there is for each depth j, $0 < j \leq m$, of *NFA* in each step i of the run one integer variable $d_{j,i}$ that contains the Hamming distance between string $p_1 \ldots p_j$ and the string in text T ending at position i. Since for each value l of the Hamming distance there is one level of states in *NFA*, the integer variable $d_{j,i} = l$ and it contains level number of the topmost active state in j-th depth of *NFA*. Each value of $d_{i,j}$ greater than $k + 1$ can be replaced by value $k + 1$ and it represents that there is no active state in j-th depth of *NFA* in i-th step of the run. So in dynamic programming, the set of active states from Chapter 4 is implemented by the vector of integer variables—each variable for one depth of *NFA*.

In Formula 5.2 term $d_{j-1,i-1}$ represents matching transition—active state is moved from depth $j - 1$ to the next depth j within level $d_{j-1,i-1}$ and symbol t_i is read from the input. Term $d_{j-1,i-1} + 1$ represents transition *replace*—active state is moved from depth $j - 1$ and level $d_{j-1,i-1}$ to the next depth j and the next level $d_{j-1,i-1} + 1$ and symbol t_i is read from the input.

If the condition in **if** statement of Formula 5.2 holds (i.e., p_j is read), only one value ($d_{j-1,i-1}$) is considered. This directly simulates the $\overline{p}$ version of the *NFA* (p_j labels only matching transition).

If we want to simulate the Σ version of the *NFA*, in **if** statement we have to consider two values ($d_{j-1,i-1}$ and $d_{j-1,i-1}+1$) when p_j is read (p_j labels not only the matching transition but also *replace* transition). Since we compute the minimum of these values, we get again Formula 5.2 ($\mathbf{min}(d_{j-1,i-1}, d_{j-1,i-1}+1) = d_{j-1,i-1}$).

The self-loop of the initial state is represented by setting $d_{0,i} \leftarrow 0$, $0 \le i \le n$.

Therefore all transitions (paths) of the *NFA* are considered.

At the beginning, only the initial state is active, therefore $d_{0,0} = 0$ and $d_{j,0} = k+1$, $0 < j \le m$.

If $d_{m,i} \le k$, then we report that the pattern P was found with $d_{m,i}$ errors (the final state of level $d_{m,i}$ is active) ending at position i. □

5.2.3 Approximate String Matching Using Levenshtein Distance

In the approximate string matching using the Levenshtein distance [Sel80, Ukk85] each element $d_{j,i}$, $0 < i \le n$, $0 \le j \le m$, contains the Levenshtein distance between the string ending at position i in T and string $p_1 \dots p_j$. Since the Levenshtein distance compares two strings of not necessary equal length, element $d_{j,i}$ is always valid (symbols can be deleted from the pattern). It also implies that we can directly determine only the ending position of the found string in text.

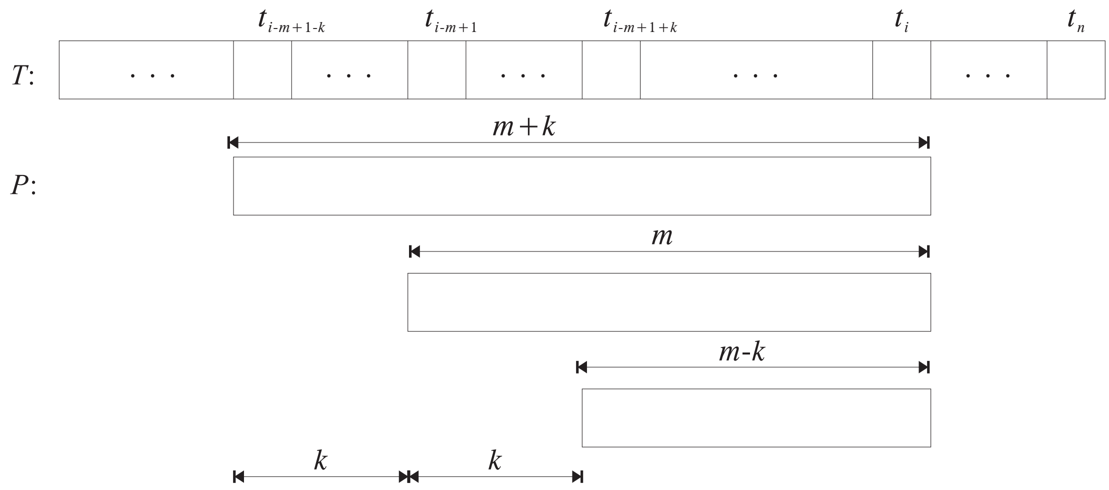


Figure 5.2: Range of beginnings of occurrence of P ending at position i in T.

If the found string ends at position i, it can start in front of position $i-m+1$ (at most k symbols inserted) or behind position $i-m+1$ (at most k symbols deleted)—the beginning of occurrence of pattern P can be located at position $i-m+1+l$, $-k \le l \le k$, as shown in Figure 5.2.

It also implies that the first possible occurrence of the pattern in the text can end at position $m-k$ and the last occurrence can start at $n-m+k+1$ as shown in Figure 5.3.

Elements of matrix D are computed as follows:

$$
\begin{array}{lcll}
d_{j,0} & \leftarrow & j, & 0 \le j \le m \\
d_{0,i} & \leftarrow & 0, & 0 \le i \le n \\
d_{j,i} & \leftarrow & \mathbf{min}(\mathbf{if}\ t_i = p_j\ \mathbf{then}\ d_{j-1,i-1}\ \mathbf{else}\ d_{j-1,i-1}+1, & \\
 & & \quad \mathbf{if}\ j < m\ \mathbf{then}\ d_{j,i-1}+1, & \\
 & & \quad d_{j-1,i}+1), & 0 < i \le n, 0 < j \le m
\end{array}
\tag{5.3}
$$

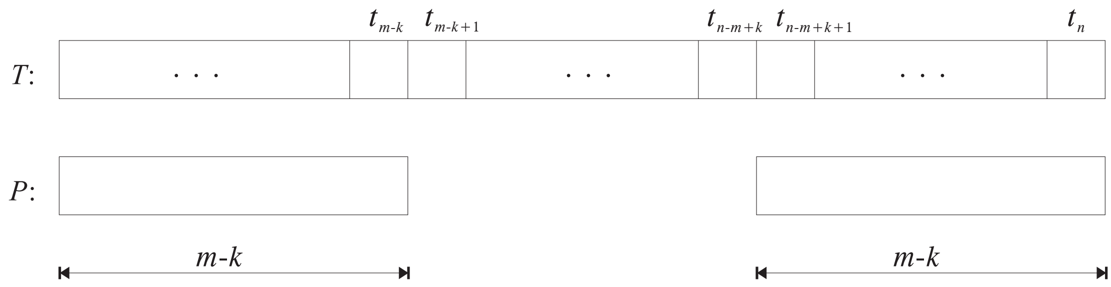


Figure 5.3: The first and the last possible occurrence of P in T.

In Formula 5.3 term $d_{j-1,i-1}$ represents matching and term $d_{j-1,i-1}+1$ represents edit operation *replace*. Term $d_{j,i-1}+1$ represents edit operation *insert*—position i in text T is increased, position j in pattern P is not increased and edit distance d is increased. Term $d_{j-1,i}+1$ represents edit operation *delete*—position i in text T is not increased, position j in pattern P is increased and edit distance d is increased.

Pattern P is found with at most k differences ending at position i if $d_{m,i} \leq k$, $1 \leq i \leq n$. The maximum number of differences of the found string is $D_L(P, t_{i-l+1} \dots t_i) = d_{m,i}$, $m-k \leq l \leq m+k$, $l < i$.

A formula similar to Formula 5.3 is used in [Ukk85], but there is missing the **if** statement in simulation of *insert* transition (term $d_{j,i-1}+1$) there, because they consider also trivial occurrences—their *NFA* would have *insert* transitions leading from final state in addition to our *NFA*. In [Ukk85] the formula is written with term $d_{j-1,i-1}+1$ (for edit operation *replace*) located out of the **else** section, so it is considered even if $t_i = p_j$. As shown in Theorem 3.10 this difference has no effect.

An example of matrix D for searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ is shown in Table 5.2 (Σ version).

D	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
d	2	1	0	1	1	1	2	1	1	1	1	0	1	2	2	1
b	3	2	1	1	2	1	2	2	2	1	2	1	0	1	2	2
b	4	3	2	2	2	2	2	3	3	2	2	2	1	0	1	2
c	5	4	3	2	3	3	2	3	4	3	3	3	2	1	0	1
a	6	5	4	3	2	4	3	2	3	4	3	4	3	2	1	0

Table 5.2: Matrix D for pattern $P = adbbca$ and text $T = adcabcaabadbbca$ using the Levenshtein distance (Σ version).

Theorem 5.2
The dynamic programming algorithm described by Formula 5.3 simulates a run of the *NFA* for the approximate string matching using the Levenshtein distance constructed by Algorithm 3.11 (Σ version).

Proof
The proof is similar to the proof of Theorem 5.1. We only have to show the simulation of edit operations *insert* and *delete*.

In Formula 5.3 term $d_{j-1,i-1}$ represents matching transition and term $d_{j-1,i-1}+1$ represents transition *replace*. Term $d_{j,i-1}+1$ represents transition *insert*—active state is moved from level $d_{j,i-1}$ to the next level $d_{j,i-1}+1$ within depth j and symbol t_i is read from the input. Term **if** $j < m$ **then** provides that *insert* transition is not considered in depth m, where the *NFA* has no *insert* transition. Term $d_{j-1,i}+1$ represents transition *delete*—active state is moved from depth $j-1$ and level $d_{j-1,i}$ to the next depth j and the next level $d_{j-1,i}+1$ and no symbol is read from the input.

Since *replace* and *insert* transitions are labeled by all symbols of input alphabet Σ, values $d_{j-1,i-1}+1$ and $d_{j,i-1}+1$ are considered even if $t_i = p_j$. The contribution of the matching transition (i.e., $d_{j-1,i-1}$) is considered only if $t_i = p_j$.

Thus all transitions of the *NFA* are considered. □

In [GP89] they compress the matrix D. They use the property that $(d_{j,i} - d_{j-1,i-1}) \in \{0,1\}$—the number of errors of an occurrence can only be nondecreasing when reading symbols of that occurrence. Using this property they shorten each column of the matrix D to $k+1$ entries. They represent the matrix D diagonal by diagonal in such a way that each line j, $0 \leq j \leq k$, of that new matrix contains the number of last entry of diagonal of matrix D containing j errors.

For the simulation of the *NFA* for the approximate string matching using the Levenshtein distance constructed by Algorithm 3.10 ($\overline{p}$ version) we can use Formula 5.4 for computing the matrix D. It is not true for this matrix D that each entry of D contains the edit distance between substring of T and prefix of the pattern P. It is because the $\overline{p}$ version of the *NFA* for the approximate string matching using the Levenshtein distance does not report exactly the Levenshtein distance between the substring of T and the prefix of the pattern P, but it skips trivial occurrences as described in Section 3.4.3.

$$
\begin{array}{lcll}
d_{j,0} & \leftarrow & j, & 0 \leq j \leq m \\
d_{0,i} & \leftarrow & 0, & 0 \leq i \leq n \\
d_{j,i} & \leftarrow & \mathbf{min}(\mathbf{if}\ t_i = p_j\ \mathbf{then}\ d_{j-1,i-1}\ \mathbf{else}\ d_{j-1,i-1}+1, & \\
 & & \quad \mathbf{if}\ j < m\ \mathbf{and}\ t_i \neq p_{j+1}\ \mathbf{then}\ d_{j,i-1}+1, & \\
 & & \quad d_{j-1,i}+1), & 0 < i \leq n, 0 < j \leq m
\end{array}
\tag{5.4}
$$

An example of matrix D for searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ is shown in Table 5.3 ($\overline{p}$ version).

D	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
d	2	1	0	1	1	1	2	1	1	1	1	0	2	2	2	1
b	3	2	1	1	2	1	2	2	2	1	2	1	0	2	3	2
b	4	3	2	2	2	2	2	3	3	2	2	2	1	0	3	3
c	5	4	3	2	3	3	2	3	4	3	3	3	2	1	0	4
a	6	5	4	3	2	4	3	2	3	4	3	4	3	2	1	0

Table 5.3: Matrix D for pattern $P = adbbca$ and text $T = adcabcaabadbbca$ using the Levenshtein distance ($\overline{p}$ version).

Theorem 5.3
The dynamic programming algorithm described by Formula 5.4 simulates a run of the *NFA* for the approximate string matching using the Levenshtein distance constructed by Algorithm 3.10 ($\overline{p}$ version).

Proof
The proof is the same as the proof of Theorem 5.2. The only difference is that *replace* transition and *insert* transition are not labeled by Σ. Each of them is labeled by $\overline{p}_j$, where p_j is the symbol used for labeling the matching transition leading from the same state from which the considered transition leads.

Therefore for considering value $d_{j-1,i-1}+1$ (*replace* transition) the current input symbol t_i must be different from p_j. For *insert* transition we have to check if $p_{j+1} \neq t_i$, since *insert* transition leads just within the same depth j of the *NFA*.

At the beginning, only the states of $\varepsilon CLOSURE(\{q_0\})$ (i.e., the initial state q_0 and the states on the ε-diagonal leading from q_0) are active, therefore $d_{j,0} = j$, $0 \leq j \leq m$. □

5.2.4 Approximate String Matching Using Generalized Levenshtein Distance

For the approximate string matching using the generalized Levenshtein distance we have modified Formula 5.3 for the Levenshtein distance such that we have added the term for edit transition *transpose*. The resulting formula is as follows:

$$
\begin{array}{lcll}
d_{j,0} & \leftarrow & j, & 0 \leq j \leq m \\
d_{0,i} & \leftarrow & 0, & 0 \leq i \leq n \\
d_{j,i} & \leftarrow & \mathbf{min}(\mathbf{if}\ t_i = p_j\ \mathbf{then}\ d_{j-1,i-1}\ \mathbf{else}\ d_{j-1,i-1}+1, & \\
& & \qquad \mathbf{if}\ j < m\ \mathbf{then}\ d_{j,i-1}+1, & \\
& & \qquad d_{j-1,i}+1, & \\
& & \qquad \mathbf{if}\ i > 1\ \mathbf{and}\ j > 1\ \mathbf{and}\ t_{i-1} = p_j\ \mathbf{and}\ t_i = p_{j-1} & \\
& & \qquad \mathbf{then}\ d_{j-2,i-2}+1), & 0 < i \leq n, 0 < j \leq m
\end{array}
\tag{5.5}
$$

In Formula 5.5 term $d_{j-1,i-1}$ represents matching, term $d_{j-1,i-1}+1$ represents edit operation *replace*, term $d_{j,i-1}+1$ represents edit operation *insert*, term $d_{j-1,i}+1$ represents edit operation *delete*, and term $d_{j-2,i-2}+1$ represents edit operation *transpose*—position i in text T is increased by 2, position j in pattern P is increased by 2 and edit distance d is increased by 1.

An example of matrix D for searching for pattern $P = adbbca$ in text $T = adbcbaabadbbca$ is shown in Table 5.4 (Σ version).

Theorem 5.4
The dynamic programming algorithm described by Formula 5.5 simulates a run of the *NFA* for the approximate string matching using the generalized Levenshtein distance constructed by Algorithm 3.13 (Σ version).

Proof
The proof is similar to the proof of Theorem 5.2. We only have to show the simulation of edit operation *transpose*.

In Formula 5.5 term $d_{j-1,i-1}$ represents matching transition, term $d_{j-1,i-1}+1$ represents transition *replace*, term $d_{j,i-1}+1$ represents transition *insert*, and term $d_{j-1,i}+1$ represents transition *delete*.

D	-	a	d	b	c	b	a	a	b	a	d	b	b	c	a
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	1	1	0	0	1	0	1	1	1	1	0
d	2	1	0	1	2	2	1	1	1	1	0	1	2	2	1
b	3	2	1	0	1	2	2	2	1	2	1	0	1	2	2
b	4	3	2	1	1	1	2	3	2	2	2	1	0	1	2
c	5	4	3	2	1	1	2	3	3	3	3	2	1	0	1
a	6	5	4	3	2	2	1	2	4	3	4	3	2	1	0

Table 5.4: Matrix D for pattern $P = adbbca$ and text $T = adbcbaabadbbca$ using the generalized Levenshtein distance (Σ version).

Term $d_{j-2,i-2} + 1$ represents transition *transpose*—active state is moved from depth $j-2$ and level $d_{j-2,i-2}$ to depth j and level $d_{j-2,i-2}+1$ and symbols t_{i-1} and t_i are read from the input. When representing transition *transpose* we need not new integer variable for state on transition *transpose* in *NFA*. □

If we modify Formula 5.4 for the Levenshtein distance, we get Formula 5.6 for simulation of the *NFA* for the approximate string matching using the generalized Levenshtein distance ($\overline{p}$ version) constructed by Algorithm 3.12.

$$
\begin{array}{lcll}
d_{j,0} & \leftarrow & j, & 0 \le j \le m \\
d_{0,i} & \leftarrow & 0, & 0 \le i \le n \\
d_{j,i} & \leftarrow & \textbf{min}(\textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} \textbf{ else } d_{j-1,i-1}+1, & \\
 & & \quad d_{j-1,i}+1, & \\
 & & \quad \textbf{if } j < m \textbf{ and } t_i \ne p_{j+1} \textbf{ then } d_{j,i-1}+1, & \\
 & & \quad \textbf{if } i > 1 \textbf{ and } j > 1 \textbf{ and } t_{i-1} = p_j \textbf{ and } t_i = p_{j-1} & \\
 & & \quad \textbf{then } d_{j-2,i-2}+1), & 0 < i \le n, 0 < j \le m
\end{array}
\tag{5.6}
$$

An example of matrix D for searching for pattern $P = adbbca$ in text $T = adbcbaabadbbca$ is shown in Table 5.5 ($\overline{p}$ version).

D	-	a	d	b	c	b	a	a	b	a	d	b	b	c	a
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	1	1	0	0	1	0	1	1	1	1	0
d	2	1	0	2	2	2	1	1	1	1	0	2	2	2	1
b	3	2	1	0	1	2	2	2	1	2	1	0	2	3	2
b	4	3	2	1	1	1	2	3	2	2	2	1	0	3	3
c	5	4	3	2	1	1	2	3	3	3	3	2	1	0	4
a	6	5	4	3	2	2	1	2	4	3	4	3	2	1	0

Table 5.5: Matrix D for pattern $P = adbbca$ and text $T = adbcbaabadbbca$ using the generalized Levenshtein distance ($\overline{p}$ version).

Theorem 5.5
The dynamic programming algorithm described by Formula 5.6 simulates a run of the *NFA*

for the approximate string matching using the generalized Levenshtein distance constructed by Algorithm 3.12 ($\overline{p}$ version).

Proof
The proof is the same as the proof of Theorem 5.3. The only difference is inserting the simulation of *transpose* transition, which is described in the proof of Theorem 5.4. □

5.3 Exact and Approximate Sequence Matching

Since *NFA*s for the string matching and for the sequence matching differs just in the self-loop of each non-final state, we have to insert a term implementing these self-loops to the formulae for the string matching.

5.3.1 Unlimited Overlapping

If we take Formula 5.2 for the approximate string matching using the Hamming distance (which is the same as for the exact string matching) and insert term $d_{j,i-1}$ implementing the self-loop labeled by Σ, we get the following formula for the approximate sequence matching using the Hamming distance with the unlimited overlapping (which is the same as for the exact string matching).

$$\begin{array}{lcll} d_{j,0} & \leftarrow & k+1, & 0 < j \leq m \\ d_{0,i} & \leftarrow & 0, & 0 \leq i \leq n \\ d_{j,i} & \leftarrow & \mathbf{min}(\mathbf{if}\ t_i = p_j\ \mathbf{then}\ d_{j-1,i-1}\ \mathbf{else}\ d_{j-1,i-1}+1, & \\ & & \qquad \mathbf{if}\ j < m\ \mathbf{then}\ d_{j,i-1}), & 0 < i \leq n, 0 < j \leq m \end{array} \tag{5.7}$$

In Formula 5.7 term $d_{j,i-1}$ represents the self-loop labeled by Σ—it provides that the previous value d_j of edit distance is held and can be only decreased.

In the same way we can modify Formula 5.3 for the Levenshtein distance (Σ version). Thus we get Formula 5.8 for the approximate sequence matching with the unlimited overlapping using the Levenshtein distance. The modification can also be done for Formula 5.4 simulating $\overline{p}$ version of *NFA*.

$$\begin{array}{lcll} d_{j,0} & \leftarrow & j, & 0 \leq j \leq m \\ d_{0,i} & \leftarrow & 0, & 0 \leq i \leq n \\ d_{j,i} & \leftarrow & \mathbf{min}(\mathbf{if}\ t_i = p_j\ \mathbf{then}\ d_{j-1,i-1}\ \mathbf{else}\ d_{j-1,i-1}+1, & \\ & & \qquad \mathbf{if}\ j < m\ \mathbf{then}\ d_{j,i-1}+1, & \\ & & \qquad d_{j-1,i}+1, & \\ & & \qquad \mathbf{if}\ j < m\ \mathbf{then}\ d_{j,i-1}), & 0 < i \leq n, 0 < j \leq m \end{array} \tag{5.8}$$

Lemma 5.6
Term **if** $j < m$ **then** $d_{j,i-1}$ in Formulae 5.7 and 5.8 simulates the self-loop labeled by Σ in the *NFA*s for the approximate sequence matching with the unlimited overlapping.

Proof
This value provides that the topmost active state in depth j is considered until the end of processing. Level of the topmost active state in depth j can only be nondecreasing.

The condition **if** $j < m$ is needed to prevent the self-loop to be used also in final state. We can omit this condition, but whenever we detect an active final state, we have to set $d_{m,i} \leftarrow k+1$. □

Term **if** $j < m$ **then** $d_{j,i-1}$ can be also inserted in Formula 5.5 for the generalized Levenshtein distance, but in this case we need special integer variables for immediate states of transitions *transpose*. These variables are needed in the approximate sequence matching since even between two symbols labeling transition *transpose* there can be located any number of symbols of input text.

5.3.2 Limited Overlapping

For the approximate sequence matching with the limited overlapping we can also insert some term simulating the self-loop, which is labeled by $\overline{p}_j$, $0 < j \leq m$, in this case. If we do so, we get following formula for the approximate sequence matching using the Hamming distance with the limited overlapping.

$$\begin{array}{lcll} d_{j,0} & \leftarrow & k+1, & 0 < j \leq m \\ d_{0,i} & \leftarrow & 0, & 0 \leq i \leq n \\ d_{j,i} & \leftarrow & \mathbf{min}(\mathbf{if}\ t_i = p_j\ \mathbf{then}\ d_{j-1,i-1}\ \mathbf{else}\ d_{j-1,i-1}+1, & \\ & & \qquad \mathbf{if}\ j < m\ \mathbf{and}\ t_i \neq p_{j+1}\ \mathbf{then}\ d_{j,i-1}), & 0 < i \leq n, 0 < j \leq m \end{array} \tag{5.9}$$

In Formula 5.9 the second **if** statement checks whether the current text symbol t_i matches p_{j+1} or not. If it mismatches, the position in the pattern is not increased and the edit distance from the previous step $(i-1)$ must be also considered in computing $d_{j,i}$.

Lemma 5.7
Term **if** $j < m$ **and** $t_i \neq p_{j+1}$ **then** $d_{j,i-1}$ in Formula 5.9 simulates the self-loop labeled by $\overline{p}$ in the *NFA*s for the approximate sequence matching with the limited overlapping.

Proof
The condition **if** $j < m$ **and** $t_i \neq p_{j+1}$ checks whether to use the self-loop labeled by $\overline{p}_{j+1}$ or not.
The value $d_{j,i-1}$ provides that the topmost active state in depth j is considered until the end of processing. Level of the topmost active state in depth j can only be nondecreasing. □

For constructing formulae simulating other *NFA*s for the approximate sequence matching with the limited overlapping, term **if** $j < m$ **and** $t_i \neq p_{j+1}$ **then** $d_{j,i-1}$ simulating the self-loop has to be inserted into the corresponding formulae for the approximate string matching.

5.4 Reduced Finite Automata

All the previous dynamic programming simulations used one integer variable for each depth of *NFA*, but there is also another possibility. One can have one integer variable for each ε-diagonal. We describe this method on reduced *NFA*s [Hol98][1], since in this case it reduces the time complexity as well as the space complexity (we omit elements of matrix D for incomplete diagonals). This kind of simulation cannot be used for the approximate string and sequence matching that uses the Hamming distance, since there are no ε-diagonals there.

[1] A similar simulation of reduced *NFA*s for the approximate string matching using the Levenshtein distance was independently presented in [BYN96b].

The last k depths of the reduced *NFA*s do not contain states in all $k+1$ levels of the *NFA* so the simulation method described in the previous sections of this chapter is not suitable for the reduced *NFA*s for the approximate string and sequence matching. Therefore we use other method in this section, in which each ε-diagonal has its integer variable.

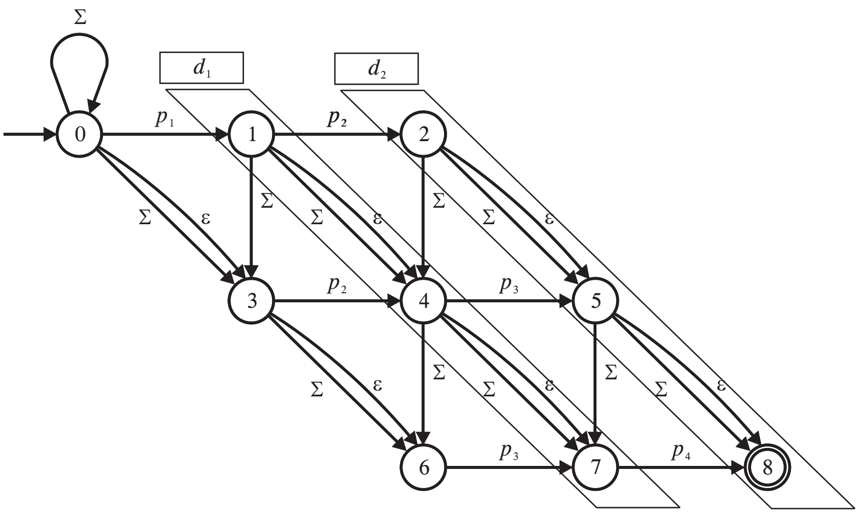


Figure 5.4: Dynamic programming for reduced *NFA*s uses for each diagonal of states of *NFA* one integer variable d.

If any state in a diagonal is active, then all the states located lower in this diagonal are also active because of ε-transitions. Therefore in the element for each diagonal l, $0 \leq l \leq m-k$, we store only the number of level of the highest active state in diagonal l. In this way we get for each step i, $0 \leq i \leq n$, of the run of *NFA* the column $d_i = d_{0,i}, d_{1,i}, \ldots, d_{m-k,i}$ of length $m-k+1$ and each element of the column contains one value ranging from 0 to $k+1$, where value $k+1$ represents that there is no active state in the corresponding ε-diagonal. The formula for computing columns d_i for the approximate string matching using the Levenshtein distance (Σ version) is shown below:

$$\begin{array}{lcll}
d_{0,i} & \leftarrow & 0, & 0 \leq i \leq n \\
d_{j,0} & \leftarrow & k+1, & 0 < j \leq m-k \\
d_{j,i} & \leftarrow & \mathbf{min}(k+1, & \\
 & & \quad g_{d_{j-1,i-1}+j,t_i} + d_{j-1,i-1}, & /*\ \textit{delete}\ \&\ \text{match}\ */ \\
 & & \quad d_{j,i-1}+1, & /*\ \textit{replace}\ */ \\
 & & \quad \mathbf{if}\ j < m-k\ \mathbf{then}\ d_{j+1,i-1}+1), & /*\ \textit{insert}\ */ \\
 & & & 0 < j \leq m-k, 0 < i \leq n
\end{array} \tag{5.10}$$

Formula 5.10 uses matrix G given by Formula 5.11, which contains for each pair j, a, $0 < j \leq m$, $a \in \Sigma$, the number of symbols that has to be skipped to reach the first following occurrence of symbol a from j-th position in the pattern P or value $k+1$ if there is no such occurrence.

$$\begin{array}{lcll}
g_{j,a} & \leftarrow & \mathbf{min}(\{k+1\} \cup \{(l-j \mid p_l = a, j \leq l)\ \mathbf{or} & \\
 & & \qquad (k+1 \mid \text{there is no such } l)\}), & 0 < j \leq m, a \in \Sigma \\
g_{m+1,a} & \leftarrow & k+1, & a \in \Sigma
\end{array} \tag{5.11}$$

Theorem 5.8
The dynamic programming algorithm described by Formula 5.10 simulates a run of the reduced *NFA* for the approximate string matching using the Levenshtein distance (Σ version) described in Section 3.6, where matrix G is defined by Formula 5.11.

Proof
The first line in the formula says that the initial state lying in the 0th diagonal of the *NFA* is always active because of its self-loop. The second one says that at the beginning of the searching there is no active state in diagonals l, $0 < l \leq m - k$, because no state in such diagonals is in $\varepsilon CLOSURE()$ of the initial state.

Term $g_{d_{j-1,i-1}+j,t_i} + d_{j-1,i-1}$ represents match and *delete* transitions. The match is represented by horizontal transitions and edit operation *delete* is represented by diagonal ε-transitions. Implementation of match transition is simple—if the state in diagonal $j-1$ and level $d_{j-1,i-1}$ is active and the horizontal transition leading from this state is labeled by symbol t_i, then the state in diagonal j and level $d_{j-1,i-1}$ becomes active.

For implementation of *delete* transition we have to search for the state in diagonal $j-1$ and level l, $d_{j-1,i-1} \leq l \leq m$, such that there is a match transition labeled by input symbol t_i leading from this state. In order to find such state in constant time, we have to use auxiliary matrix G, in which there is for each position r in pattern P and input symbol t_i the number r', $0 \leq r'$, such that $p_{r+r'} = t_i$, where r' is the lowest possible. If there is no such position, then $r' = k+1$. Since value of $d_{j-1,i-1}$ can reach up to $k+1$ and the maximum number of diagonal, into which match transitions lead, is $m-k$, the maximum position for which a value of matrix G is required is $m - k + k + 1 = m + 1$. Therefore the matrix has to be of size $(m+1) \times (|\Sigma|)$.

Number $d_{j-1,i-1} + j$ gives position of symbol $p_{d_{j-1,i-1}+j}$ in the pattern, which is used as a label of the match transition leading from the highest active state in diagonal $j-1$ to a state in diagonal j. Therefore $g_{d_{j-1,i-1}+j,t_i} + d_{j-1,i-1}$ gives the level of the highest active state in diagonal j that has arisen by using match transition labeled by t_i to each active state in diagonal $j-1$.

Term $d_{j,i-1}+1$ represents *replace* transition. Edit operation *replace* is represented by diagonal transition labeled by Σ. To implement *replace* transition it is only needed to move the highest active state of diagonal j to the next lower position in the same diagonal.

Term $d_{j+1,i-1}+1$ represents *insert* transition. Edit operation *insert* is represented by vertical transition also labeled by Σ. The active state in diagonal $j+1$ and level $d_{j+1,i-1}$ moves to level $d_{j+1,i-1}+1$ in diagonal j.

From these transitions we get minimum in order to obtain the highest active state in each diagonal. □

An example of matrix G for pattern $P = adbbca$ is shown in Table 5.6 and the process of searching for pattern $P = adbbca$ with at most $k = 3$ errors in text $T = adcabcaabadbbca$ is shown in Table 5.7.

This method can be also used for simulation of the run of the reduced *NFA* for the approximate string matching using the generalized Levenshtein distance. We have only to add term representing edit operation *transpose*. In Formula 5.10, the added term is as follows:

$$
\begin{array}{ll}
\textbf{if } p_{d_{j-1,i-2}+j+1} = t_{i-1} \textbf{ and } p_{d_{j-1,i-2}+j} = t_i & \\
\textbf{then } d_{j-1,i-2}+1, & /*\ \textit{transpose}\ */ \\
 & 0 < j \leq m-k, 1 < i \leq n
\end{array}
\qquad (5.12)
$$

G	a	b	c	d	$\Sigma \setminus \{a,b,c,d\}$
1	0	2	4	1	4
2	4	1	3	0	4
3	3	0	2	4	4
4	2	0	1	4	4
5	1	4	0	4	4
6	0	4	4	4	4
7	4	4	4	4	4

Table 5.6: Matrix G for pattern $P = adbbca$ and $k = 3$.

D	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	4	0	1	1	0	1	2	0	0	1	0	1	1	2	2	0
2	4	4	0	1	2	1	2	3	4	1	2	0	1	1	2	3
3	4	4	4	2	3	4	2	3	3	4	3	4	0	1	2	3

Table 5.7: Matrix D for pattern $P = adbbca$, text $T = adcabcaabadbbca$, and $k = 3$.

The resulting formula for the reduced *NFA* for the approximate string matching using the generalized Levenshtein distance is then Formula 5.13.

$$
\begin{array}{lll}
d_{0,i} & \leftarrow & 0, \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 0 \le i \le n \\
d_{j,0} & \leftarrow & k+1, \qquad\qquad\qquad\qquad\qquad\qquad\qquad 0 < j \le m-k \\
d_{j,i} & \leftarrow & \mathbf{min}(k+1, \\
 & & \qquad g_{d_{j-1,i-1}+j,t_i} + d_{j-1,i-1}, \qquad\qquad /*\ \textit{delete}\ \&\ \text{match}\ */ \\
 & & \qquad d_{j,i-1}+1, \qquad\qquad\qquad\qquad\quad /*\ \textit{replace}\ */ \\
 & & \qquad \mathbf{if}\ j < m-k\ \mathbf{then}\ d_{j+1,i-1}+1, \quad /*\ \textit{insert}\ */ \\
 & & \qquad \mathbf{if}\ p_{d_{j-1,i-2}+j+1} = t_{i-1}\ \mathbf{and}\ p_{d_{j-1,i-2}+j} = t_i \\
 & & \qquad \mathbf{then}\ d_{j-1,i-2}+1), \qquad\qquad\qquad /*\ \textit{transpose}\ */ \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad 0 < j \le m-k, 0 < i \le n
\end{array}
\tag{5.13}
$$

Lemma 5.9
Let M be the reduced *NFA* for the approximate string matching using the generalized Levenshtein distance ($\overline{p}$ version) described in Section 3.6. For each state $q_{l,j}$ of diagonal j, $0 \le j < m-k$, and level l, $0 \le l < k$, it holds that if $q_{l,j}$ is the topmost active state in diagonal j, no *transpose* transition leading from diagonal j has to be considered except for the *transpose* transition leading directly from $q_{l,j}$.

Proof
Let depth of $q_{l,j}$ is Δ (i.e., $depth(q_{l,j}) = \Delta$). The *transpose* transition leading directly from $q_{l,j}$ is labeled by symbols $p_{\Delta+2}$ and $p_{\Delta+1}$ and it leads to state $q_{l+1,j+1}$.

The only way how to get from $q_{l,j}$ to $q_{l+1,j+1}$ after reading $p_{\Delta+2}$ and $p_{\Delta+1}$ without using the *transpose* transition could be if $p_{\Delta+2} = p_{\Delta+1}$. Then one can move from $q_{l,j}$ to $q_{l+1,j+1}$ using match transition and then using *replace* transition or using these transitions in inverted order.

For each other *transpose* transition leading from a state $q_{l+i,j}$ located in diagonal j lower than $q_{l,j}$ (i.e., its level is greater than l) it holds that it is labeled by symbols $p_{\Delta+2+i}$ and $p_{\Delta+1+i}$, $i \geq 1$, $\Delta + 2 + i \leq m$, and it leads to state $q_{l+i+1,j+1}$ (with $l+i+1$ errors).

While reading $p_{\Delta+2+i}$, one can move from $q_{l,j}$ to state $q_{l+1,j}$ using *replace* transition (labeled by Σ) and then using $(i-1)$ ε-transitions one can move further to state $q_{l+i,j}$ (of depth $\Delta + i$). Finally while reading symbol $p_{\Delta+1+i}$, one can move to state $q_{l+i,j+1}$ (with $l+i$ errors), which has less errors that the number of errors given by the considered *transpose* transition ($l+i+1$ errors).

Thus the *transpose* transitions leading from states $q_{l+i,j}$, $0 < i$, $l + i \leq k$, do not require to be considered—a combination of *replace*, *delete*, and match transitions obtains better number of errors. □

Theorem 5.10
The dynamic programming algorithm described by Formula 5.13 simulates a run of the reduced *NFA* for the approximate string matching using the generalized Levenshtein distance (Σ version) described in Section 3.6, where matrix G is defined by Formula 5.11.

Proof
The simulation is the same as in Theorem 5.8 (for the Levenshtein distance) except for *transpose* transition.

We need only to simulate the *transpose* transition leading from the topmost active state in each diagonal (See Lemma 5.9).

Let the topmost active state q of diagonal $j-1$ in step $i-2$ be in depth Δ. $\Delta = d_{j-1,i-2}+j-1$. Term $d_{j-1,i-2}+1$ provides that the level of q is increased by one (one error for edit operation *transpose*). The resulting value is considered in diagonal j in step i—two symbols has been read from the input and depth Δ of state in *NFA* has been also increased by 2 (one increase of level and one increase of diagonal).

if statement provides that this value is considered only if $p_{d_{j-1,i-2}+j+1} = t_{i-1}$ (i.e., $p_{\Delta+2} = t_{i-1}$) and $p_{d_{j-1,i-2}+j} = t_i$ (i.e., $p_{\Delta+1} = t_i$)—if labels of transition *transpose* $(p_{\Delta+2}, p_{\Delta+1})$ have been read from the input in steps $i-1$ and i. □

The simulation of $\overline{p}$ version of the reduced *NFA*s needs another auxiliary matrix. For the simulation of *replace* transition we have to replace term $d_{j,i-1}+1$ by **if** $p_{d_{j,i-1}+j+1} \neq t_i$ **then** $d_{j,i-1}+1$ (*replace* transition is labeled by $\overline{p}$ in this case). But for the simulation of *insert* transition we have to use the topmost *insert* transition, which is labeled by $\overline{t_i}$ and leads from an active state of the next diagonal into the current diagonal. To find such relevant *insert* transition in constant time we have to build the matrix (like matrix G for match and *delete* transitions), which gives for each pair j, a, $0 < j \leq m$, $a \in \Sigma$, the number of symbols a located just behind position j in the pattern P.

This type of simulation of the reduced *NFA*s can be also used for the reduced *NFA*s for the approximate sequence matching using the Levenshtein and generalized Levenshtein distances [Hol97]. To modify the presented algorithms so that they could simulate the reduced *NFA*s for the approximate sequence matching, we have to implement the self-loop in each non-final and non-initial state.

If we insert the following term into Formulae 5.10 and 5.13 for the approximate string matching, we get formulae for the approximate sequence matching with the limited overlapping using the Levenshtein and generalized Levenshtein distances respectively.

$$\begin{array}{lll} \textbf{if } (j < m-k \textbf{ or } d_{j,i-1} < k) \textbf{ and } p_{d_{j,i-1}+j+1} \neq t_i & & \\ \textbf{then } d_{j,i-1}, & /* \text{ self-loop labeled by } \overline{p} */ & \\ & 0 < j \leq m-k, 0 < i \leq n & \end{array} \quad (5.14)$$

If we insert the following term into Formulae 5.10 and 5.13, we get formulae for the approximate sequence matching with the unlimited overlapping using the Levenshtein and generalized Levenshtein distances respectively.

$$\begin{array}{ll} \textbf{if } j < m-k \textbf{ or } d_{j,i-1} < k & \\ \textbf{then } d_{j,i-1}, & /* \text{ self-loop labeled by } \Sigma */ \\ & 0 < j \leq m-k, 0 < i \leq n \end{array} \quad (5.15)$$

In Formula 5.14 and 5.15 the condition ($j < m-k$ **or** $d_{j,i-1} < k$) prevents the self-loop to be in a final state (the state of level $k+1$ and depth m).

5.5 String and Sequence Verification

As shown in Section 3.7, the exact string and sequence verification is so simple that the use of *FA*s is not efficient, therefore we discuss only the approximate string and sequence verification in this section. The *NFA* for the approximate string verification using the Hamming distance can be simulated by Formula 5.16.

$$\begin{array}{lll} d_{0,0} & \leftarrow & 0 \\ d_{j,0} & \leftarrow & k+1, \qquad 0 < j \leq m \\ d_{0,i} & \leftarrow & k+1, \qquad 0 < i \leq m \\ d_{j,i} & \leftarrow & \textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} \textbf{ else } d_{j-1,i-1}+1, \quad 0 < i \leq m, 0 < j \leq m \end{array} \quad (5.16)$$

A process of verification for the Hamming distance using Formula 5.16 is shown in Table 5.8.

D	-	a	d	a	a	c	a
-	**0**	3	3	3	3	3	3
a	3	**0**	4	3	3	4	3
d	3	4	**0**	5	4	4	5
b	3	4	5	**1**	6	5	5
b	3	4	5	6	**2**	7	6
c	3	4	5	6	7	**2**	8
a	3	3	5	5	6	8	**2**

Table 5.8: Matrix D for the approximate string verification using the Hamming distance (pattern $P = adbbca$, text $T = adaaca$, and $k = 2$).

Since in the Hamming distance the compared strings are of the same length and the only allowed operation is *replace*, we do not need to compute whole column of the matrix D but only one value[2] (i-th value in i-th step). If we then compute that one value in each step, we just only

[2]These important values are emphasized in Table 5.8.

count the number of mismatches when comparing the text and the pattern. For this counting we can use Algorithm 3.18, in which we replace statement "**return**('No.')" by a term counting the mismatches.

In the case of the Levenshtein and generalized Levenshtein distance the situation is not so simple. In this case states of not only one depth of *NFA* in each step can influence the result (like in the case of the Hamming distance), but states of up to $(2k+1)$ depths of *NFA* can influence the result, because the edit operations *insert* and *delete* can change the length of compared strings).

Since the *NFA*s for the approximate string matching and for the approximate string verification differs in the absence of the self-loop of the initial state, if we remove the simulation of this self-loop from Formulae 5.3 and 5.5, we get Formulae 5.17 and 5.18 for the approximate string verification using the Levenshtein and generalized Levenshtein distances respectively (Σ versions).

The *NFA*s for the approximate string matching and for the approximate string verification differ also in presence of states under ε-diagonal leading from the initial state. However, these states are also considered in the simulation—we consider the topmost active state in each depth of *NFA*.

$$
\begin{array}{lcll}
d_{j,0} & \leftarrow & j, & 0 \le j \le m \\
d_{0,i} & \leftarrow & k+1, & 0 < i \le m+k \\
d_{j,i} & \leftarrow & \min(\textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} \textbf{ else } d_{j-1,i-1}+1, & \\
 & & \quad \textbf{if } j < m \textbf{ then } d_{j,i-1}+1, & \\
 & & \quad d_{j-1,i}+1), & 0 < i \le m+k, 0 < j \le m
\end{array}
\tag{5.17}
$$

$$
\begin{array}{lcll}
d_{j,0} & \leftarrow & j, & 0 \le j \le m \\
d_{0,i} & \leftarrow & k+1, & 0 < i \le m+k \\
d_{j,i} & \leftarrow & \min(\textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} \textbf{ else } d_{j-1,i-1}+1, & \\
 & & \quad \textbf{if } j < m \textbf{ then } d_{j,i-1}+1, & \\
 & & \quad d_{j-1,i}+1, & \\
 & & \quad \textbf{if } i > 1 \textbf{ and } j > 1 \textbf{ and } t_{i-1} = p_j \textbf{ and } t_i = p_{j-1} & \\
 & & \quad \textbf{then } d_{j-2,i-2}+1), & 0 < i \le m+k, \\
 & & & 0 < j \le m
\end{array}
\tag{5.18}
$$

A process of verification for the Levenshtein distance using Formula 5.17 is shown in Table 5.9.

D	-	a	d	b	c	b	a	a	b
-	0	3	3	3	3	3	3	3	3
a	1	0	1	2	3	4	3	3	4
d	2	1	0	1	2	3	4	4	4
b	3	2	1	0	1	2	3	4	4
b	4	3	2	1	1	1	2	3	4
c	5	4	3	2	1	2	2	3	4
a	6	5	4	3	2	2	2	2	3

Table 5.9: Matrix D for the approximate string verification using the Levenshtein distance (pattern $P = adbbca$, text $T = adbcbaab$, and $k = 2$).

Theorem 5.11
The dynamic programming algorithm described by Formulae 5.17 and 5.18 simulate a run of

the *NFA* for the approximate string verification using the Levenshtein distance and the *NFA* for the approximate string verification using the generalized Levenshtein distance respectively (Σ version) described in Section 3.7.

Proof

The difference between the *NFA*s for the approximate string matching and the approximate string verification is the absence of the self-loop of the initial state. Therefore we have to provide that the initial state is active only at the beginning of process ($i = 0$). Thus $d_{0,0} = 0$ and $d_{0,i} = k+1$, $i > 0$. Whole process is limited by $(m + k)$ steps, since up to k symbols can be inserted in the pattern of length m.

The other difference between *NFA* for the approximate string matching and the approximate string verification using the Levenshtein and generalized Levenshtein distances is that the *NFA* for the approximate string verification contains also states under the ε-diagonal leading from the initial state. Such states were redundant in the *NFA* for the approximate string matching (See Theorem 3.14). We do not need to change anything in formulae for dynamic programming to simulate also these states, since they are already considered when computing the topmost active state of the corresponding depth. $\square$

We can make also modifications for $\overline{p}$ version of *NFA*s, reduced *NFA*s, as well as for the *NFA*s for the sequence matching and thus we get the simulations of other *NFA*s for verification.

5.6 Nonuniform Weights of Edit Operations

In the weighted approximate string matching each edit operation (*replace*, *insert*, *delete*, *transpose*) has its positive integer weight w (w_R, w_I, w_D, w_T respectively). These weights can be constants or they can vary according to the symbols involved in the corresponding edit operations or according to the position in the pattern (depth of *NFA*). In this section we discuss the most general case, when the weights are dependent on the symbols involved as well as on the position in the pattern ($w_R(a,b,j)$, $w_I(a,j)$, $w_D(a,j)$, $w_T(a,b,j)$, where $a,b \in \Sigma$, $0 < j \leq m$). Solutions for other cases can be easily obtained from this solution.

The formulae simulating the *NFA*s for the weighted approximate string matching using the Hamming, Levenshtein, and generalized Levenshtein distance (Σ versions) are Formulae 5.19, 5.20, and 5.21 respectively.

$$
\begin{array}{lcll}
d_{j,0} & \leftarrow & k+1, & 0 < j \leq m \\
d_{0,i} & \leftarrow & 0, & 0 \leq i \leq n \\
d_{j,i} & \leftarrow & \textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} & \\
 & & \quad \textbf{else } d_{j-1,i-1} + w_R(t_i,p_j,j), & 0 < i \leq n, 0 < j \leq m
\end{array}
\tag{5.19}
$$

$$
\begin{array}{lcll}
d_{j,0} & \leftarrow & j, & 0 \leq j \leq m \\
d_{0,i} & \leftarrow & 0, & 0 \leq i \leq n \\
d_{j,i} & \leftarrow & \textbf{min}(\textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} & \\
 & & \quad \textbf{else } d_{j-1,i-1} + w_R(t_i,p_j,j), & \\
 & & \quad \textbf{if } j < m \textbf{ then } d_{j,i-1} + w_I(t_i,j), & \\
 & & \quad d_{j-1,i} + w_D(p_j,j)), & 0 < i \leq n, 0 < j \leq m
\end{array}
\tag{5.20}
$$

$$
\begin{array}{lcll}
d_{j,0} & \leftarrow & j, & 0 \le j \le m \\
d_{0,i} & \leftarrow & 0, & 0 \le i \le n \\
d_{j,i} & \leftarrow & \mathbf{min}(\mathbf{if}\ t_i = p_j\ \mathbf{then}\ d_{j-1,i-1} & \\
 & & \qquad \mathbf{else}\ d_{j-1,i-1} + w_R(t_i, p_j, j), & \\
 & & \qquad \mathbf{if}\ j < m\ \mathbf{then}\ d_{j,i-1} + w_I(t_i, j), & \\
 & & \qquad d_{j-1,i} + w_D(p_j, j), & \\
 & & \qquad \mathbf{if}\ i > 1\ \mathbf{and}\ j > 1\ \mathbf{and}\ t_{i-1} = p_j\ \mathbf{and}\ t_i = p_{j-1} & \\
 & & \qquad \mathbf{then}\ d_{j-2,i-2} + w_T(p_{j-1}, p_j, j)), & 0 < i \le n, 0 < j \le m
\end{array}
\tag{5.21}
$$

These formulae differ from Formulae 5.2, 5.3, and 5.5 only in such a way that they increase value of $d_{j,i}$ not by 1 but by the corresponding weight.

5.7 Time and Space Complexity

In the algorithms presented in this chapter, matrix D of size mostly $(m+1) \times (n+1)$ (or $(m-k+1) \times (n+1)$ for reduced *NFA*s) is computed, but in practice one needs just only one (or two for the generalized Levenshtein distance) previous columns d_{i-1} (or d_{i-1}, d_{i-2} respectively) in order to compute column d_i. In the columns we do not need to store 0-th item, since this item contains always the same value (except step 0). Therefore the space complexity of this matrix is $\mathcal{O}(m)$ (or $\mathcal{O}(m-k)$ for reduced *NFA*s). In case of reduced *NFA*s one needs also auxiliary matrix G of size $(m+1) \times |\Sigma|$.

All operations shown in formulae for computing each element of matrix D can be performed in constant time, therefore the time complexity of the simulation of run of *NFA*s for the approximate string and sequence matching is $\mathcal{O}(mn)$ (or $\mathcal{O}((m-k)n)$ for reduced *NFA*s). In the case of the reduced *NFA*s one has also to consider the preprocessing time, which is needed for computing auxiliary matrix G. Matrix G can be computed by Algorithm 5.2, which runs in time $\mathcal{O}(m|\Sigma|)$.

In [GP89] they compacted the matrix D for the approximate string matching using the Levenshtein distance exploiting the property that $d_{j,i} - d_{j-1,i-1} \in \{0,1\}$, $0 < i \le n$, $0 < j \le m$. For each diagonal of the matrix D they store only the positions, in which the value increases.

Let us remark that the size of the input alphabet can be reduced as shown in Section 3.3 to reduced alphabet Σ', $\Sigma' \subseteq \Sigma$, $|\Sigma'| \le m+1$ (Σ' contains all the symbols used in pattern P and one special symbol for all the symbols not contained in P). So Σ is bounded by a length m of pattern P.

The time and space complexities are summarized in Table 5.10. For the reduced *NFA* there are also shown complexities with used alphabet reduction, where $|\sigma|$ ($|\sigma| \le |\Sigma|$) denotes space complexity of mapping $\sigma : \Sigma \mapsto \Sigma'$ (See Section 3.3).

5.8 General Rules

In the simulation of *NFA* using the dynamic programming described in this chapter we divide the set Q of states of *NFA* to subsets $Q_1, Q_2, \ldots, Q_l$ such that:

(1) $Q = \bigcup_{i=1}^{l} Q_i$,

(2) $Q_i \cap Q_j = \emptyset$, $\forall i,j$, $1 \le i \le l$, $1 \le j \le l$, $i \ne j$,

(3) in each situation there can be at most one active state in each subset Q_i, $1 \le i \le l$.

Algorithm 5.2 (Computing auxiliary matrix G)
Input: Pattern $P = p_1p_2 \dots p_m$, maximum number of differences allowed k.
Output: Matrix G of size $(m+1) \times (|\Sigma|)$.
Method:

```
for each a ∈ Σ do
  i ← k + 1
  g_{m+1,a} ← i
  for j ← m, m − 1, ..., 1 do
    if p_j = a then
      i ← 0
    else
      if i < k + 1 then
        i ← i + 1
      endif
    endif
    g_{j,a} ← i
  endfor
endfor
```

While dividing Q to subsets, we could get some subset Q_i, that can contain more than one active state. In order to satisfy condition (3) in such case, we have to choose one of these active states as a representative and other active states are not considered. The choice must always be performed without any change of *NFA* behavior:

(a) other active states belong to $\varepsilon CLOSURE()$ of the selected state (the topmost active state of ε-diagonal in Section 5.4) or

(b) the selected state is chosen with respect to the result of *NFA* (i.e., the minimum of errors considered in this thesis)—if the set of all accepting paths leading from the selected state of Q_i covers the set of all accepting paths leading from all active states of Q_i (i.e., if there is no string w such that a path labeled by w leads from an active state of Q_i to a final state and simultaneously there is no path labeled by w leading from the selected state to any of final states).

The states in each subset Q_i are numbered. Each subset Q_i is implemented by an integer variable that contains a number of active state in this subset or a special value if there is no active state in this subset. In the dynamic programming our goal is to minimize the number of these integer variables and to implement the transitions in the simplest way. Thus *NFA* is decomposed to a set of state-variables.

In [Ukk85] there is shown the inverted process. If we have a vector $\vec{d}$ (of limited length) of integer variables (with a limited number of values in each such variable) describing a stage of computing and all values of the vector depends just on the values of the vector in the previous step of computing, then we can construct the *DFA* for that computing. Such *DFA* construction is shown in Algorithm 5.3.

Since each configuration of the vector (vector setting) in the dynamic programming (with the above mentioned conditions) represents a *DFA* state, running the dynamic programming we

Algorithm 5.3 (*DFA* **Construction from the Dynamic Programming)**
Input: Initial setting $\vec{d_0}$ of vector $\vec{d}$, function *Next*() computing new value of $\vec{d}$ from the previous value of $\vec{d}$ and an input symbol $a \in \Sigma$, function *Final*() testing a final vector setting.
Output: *DFA* $M = (Q, \Sigma, \delta, q_0, F)$.
Method:

$Vectors \leftarrow \{\vec{d_0}\}$
$New \leftarrow \{\vec{d_0}\}$
$F \leftarrow \emptyset$
$\tau(\vec{d_0}) \leftarrow q_0$ /* mapping: vector setting to *DFA* state */
$Q \leftarrow \{q_0\}$
$i \leftarrow 1$
while $New \neq \emptyset$ **do**
  $S \leftarrow$ any element of New
  $New \leftarrow New \setminus \{S\}$
  for each $a \in \Sigma$ **do**
    $S' \leftarrow Next(S, a)$
    if $S' \notin Vectors$ **then** /* S' is new */
      $Vectors \leftarrow Vectors \cup \{S'\}$
      $Q \leftarrow Q \cup \{q_i\}$
      $\tau(S') \leftarrow q_i$
      $New \leftarrow New \cup \{S'\}$
      if $Final(S')$ **then**
        $F \leftarrow F \cup \{q_i\}$
      endif
      $i \leftarrow i + 1$
    endif
    $\delta(\tau(S), a) \leftarrow \tau(S')$
  endfor
endwhile

	approximate string	approximate sequence
matching: non-reduced *NFA*	time $\mathcal{O}(mn)$ space $\mathcal{O}(m)$	time $\mathcal{O}(mn)$ space $\mathcal{O}(m)$
matching: reduced *NFA*	time $\mathcal{O}(m\|\Sigma\| + (m-k)n)$ space $\mathcal{O}(m\|\Sigma\| + m - k)$ or time $\mathcal{O}(m^2 + (m-k)n)$ space $\mathcal{O}(m^2 + \|\sigma\| + m - k)$	time $\mathcal{O}(m\|\Sigma\| + (m-k)n)$ space $\mathcal{O}(m\|\Sigma\| + m - k)$ or time $\mathcal{O}(m^2 + (m-k)n)$ space $\mathcal{O}(m^2 + \|\sigma\| + m - k)$
verification	time $\mathcal{O}(m^2)$ or $\mathcal{O}(m(m+k))$ space $\mathcal{O}(m)$	time $\mathcal{O}(m^2)$ or $\mathcal{O}(m(m+k))$ space $\mathcal{O}(m)$

Table 5.10: The time and space complexities of the dynamic programming for the approximate string and sequence matching.

construct the equivalent *DFA* but we forget all computed states (we evaluate only the states used during the run). In Algorithm 5.3 we evaluate all possible states. Sometimes we can use a combination of the dynamic programming and the *DFA* construction[3] as mentioned in Section 4.1. It is useful if the *DFA* is too huge and cannot fit in memory.

[3]This idea is also discussed in [Nav97].

Chapter 6

Bit Parallelism

Bit parallelism is a method that uses bit vectors and benefits from the feature that the same bitwise operations (OR, AND, ADD, ... etc.) over groups of bits (or over individual bits) can be performed at once in parallel over the whole bit vector. The representatives of bit parallelism are Shift-Or, Shift-And, and Shift-Add algorithms.

Bit parallelism was used for the exact string matching (Shift-And in [Döm64]), the multiple exact string matching (Shift-And in [Shy76]), the approximate string matching using the Hamming distance (Shift-Add in [BYG92]), the approximate string matching using the Levenshtein distance (Shift-Or in [BYG92] and Shift-And in [WM92]) and for the generalized pattern matching (Shift-Or in [Abr87]), where the pattern consists not only from symbols but also from sets of symbols. In [BYG92, WM92] they consider also trivial occurrences.

In this chapter we will discuss only Shift-Or and Shift-Add algorithms. Shift-And algorithm is the same as Shift-Or algorithm but the meaning of 0 and 1 is exchanged as well as the use of bitwise operations AND and OR is exchanged.

6.1 Algorithm

The Shift-Or algorithm uses matrices R^l, $0 \le l \le k$ of size $m \times (n+1)$ and mask matrix D of size $m \times |\Sigma|$. Each element $r^l_{j,i}$, $0 < j \le m$, $0 \le i \le n$, contains 0, if the edit distance between string $p_1 \dots p_j$ and string ending at position i in text T is $\le l$, or 1, otherwise. Each element $d_{j,x}$, $0 < j \le m$, $x \in \Sigma$, contains 0, if $p_j = x$, or 1, otherwise. The matrices are implemented as tables of bit vectors as follows:

$$R^l_i = \begin{bmatrix} r^l_{1,i} \\ r^l_{2,i} \\ \vdots \\ r^l_{m,i} \end{bmatrix} \text{ and } D[x] = \begin{bmatrix} d_{1,x} \\ d_{2,x} \\ \vdots \\ d_{m,x} \end{bmatrix}, 0 \le i \le n, 0 \le l \le k, x \in \Sigma. \tag{6.1}$$

6.2 String Matching

6.2.1 Exact String Matching

In the exact string matching, vectors R^0_i, $0 \le i \le n$, are computed as follows [BYG92]:

$$
\begin{array}{lll}
r^0_{j,0} & \leftarrow 1, & 0 < j \leq m \\
R^0_i & \leftarrow \mathbf{shl}(R^0_{i-1}) \text{ OR } D[t_i], & 0 < i \leq n
\end{array}
\tag{6.2}
$$

In Formula 6.2 operation $\mathbf{shl}()$ is the bitwise operation `left shift` that inserts 0 at the beginning of vector and operation OR is the bitwise operation `or`. Term $\mathbf{shl}(R^0_{i-1})$ OR $D[t_i]$ represents matching—position i in text T is increased, position in pattern P is increased by operation $\mathbf{shl}()$, and the positions corresponding to the input symbol t_i are selected by term OR $D[t_i]$. Pattern P is found at position $t_{i-m+1} \ldots t_i$ if $r^0_{m,i} = 0$, $0 < i \leq n$.

An example of mask matrix D for pattern $P = adbbca$ is shown in Table 6.1 and an example of matrix R^0 for exact searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ is shown in Table 6.2.

D	a	b	c	d	$\Sigma \setminus \{a,b,c,d\}$
a	0	1	1	1	1
d	1	1	1	0	1
b	1	0	1	1	1
b	1	0	1	1	1
c	1	1	0	1	1
a	0	1	1	1	1

Table 6.1: Matrix D for pattern $P = adbbca$.

R^0	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
d	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**

Table 6.2: Matrix R^0 for the exact string matching (for pattern $P = adbbca$ and text $T = adcabcaabadbbca$).

Lemma 6.1
Given a pattern $P = p_1p_2 \ldots p_m$. The maximum number of configurations of bit vector $R^0 = r_1r_2 \ldots r_m$ in Shift-Or algorithm for the exact string matching defined by Formula 6.2 running over any input text T is $m + 1$.

Proof
The proof is given by the induction on the length of the found prefix.

At the beginning of the searching using the Shift-Or algorithm there is only one configuration $conf_0$ of vector R^0 that does not contain any 0 in the vector and which represents that no prefix has been found in the input text.

Let $conf_j$, $0 < j \leq m$, denote the configuration of R^0 with j-th bit reset, $r_j = 0$, $r_l \in \{0,1\}$, $1 \leq l < j$, $r_k = 1$, $\forall k$, $j < k \leq m$, (i.e., prefix $p_1p_2 \ldots p_j$ found).

To obtain the new configuration $conf_{i+1}$ (i.e., prefix $p_1p_2 \ldots p_ip_{i+1}$ found) we need to shift the configuration[1] $conf_i = r_1r_2 \ldots r_{i-1}01^{(m-i)}$ (i.e., prefix $p_1p_2 \ldots p_i$ found) by one position. After the shift we get the vector $0r_1r_2 \ldots r_{i-1}01^{(m-i-1)}$. Now, the bitwise operation OR must be performed with the mask vector corresponding to the input symbol in order to select the only bits corresponding to the input symbol.

If the input symbol is p_{i+1}, then we obtain our configuration $conf_{i+1} = r'_1r'_2 \ldots r'_i01^{(m-i-1)}$, where $r'_j = 0$, $1 < j \leq i$, if $r_{j-1} = 0 \wedge p_j = p_{i+1}$, (i.e., prefix $p_1p_2 \ldots p_j$ is a suffix of $p_1p_2 \ldots p_{i+1}$), or 1, otherwise, and $r'_1 = 0$, if $p_1 = p_{i+1}$, or 1, otherwise.

If the input symbol is not p_{i+1}, then we get the configuration $r'_1r'_2 \ldots r'_i11^{(m-i-1)}$, which already exists because for each configuration $conf_l$, $0 < l \leq m$, it holds that if we set the l-th bit, we obtain the configuration $conf_{l'}$, $0 \leq l' < l$.

It is true because $conf_l$ represents the prefix $p_1p_2 \ldots p_l$ that has been found. If all bits k, $1 \leq k < l$, are set, then by setting l-th bit in $conf_l$ we get already existing configuration $conf_0 = 1^{(m)}$ (no prefix found). If any bit k, $1 \leq k < l$, is reset, that means that not only prefix $p_1p_2 \ldots p_l$ has been found but also prefix $p_1p_2 \ldots p_k$ has been found. Moreover, this prefix $p_1p_2 \ldots p_k$ is also a suffix of $p_1p_2 \ldots p_l$ (i.e., $p_1p_2 \ldots p_k = p_{l-k+1}p_{l-k+2} \ldots p_l$). As $p_1p_2 \ldots p_k$ is a prefix of $p_1p_2 \ldots p_l$, then on the way from $conf_0$ to $conf_l$ the Shift-Or algorithm has to meet $conf_k$, therefore by setting l-th bit in $conf_l$ we get already existing configuration $conf_{l'}$, $0 \leq l' \leq l$.

Since each configuration $conf_i$ (corresponding to found prefix $p_1p_2 \ldots p_i$), $0 < i \leq m$, arises from the previous configuration $conf_{i-1}$, there are $m+1$ configurations—as many as the number of all prefixes of P (including prefix of length 0). □

Theorem 6.2
Shift-Or algorithm described by Formula 6.2 simulates a run of the *NFA* for the exact string matching constructed by Algorithm 3.4.

Proof
In Shift-Or algorithm for the exact string matching there is one bit vector R^0_i, $0 \leq i \leq n$, which represents the set of active states of *NFA*. In the vector, 0 represents active state and 1 represents non-active state of the simulated *NFA*. So in Shift-Or algorithm, the set of active states from Chapter 4 is implemented by bit vector.

In Formula 6.2, term $\mathbf{shl}(R^0_{i-1})$ OR $D[t_i]$ represents matching transition—each active state is moved to the next position in the right[2] in the same level. All active states are moved at once and only the transitions corresponding to the read symbol t_i are selected by mask vector $D[t_i]$, which changes 0 to 1 in each such state that its incoming matching transition is not labeled by t_i. The initial state of *NFA* is not in vector R^0 and is implemented by inserting 0 at the beginning of the vector in operation **shl**()—initial state is always active because of its self-loop.

At the beginning only the initial state is active therefore $R^0_0 = 1^{(m)}$.

If $r^0_{m,i} = 0$, $0 < i \leq n$, then we report that the final state is active and thus the pattern is found ending at position i in the text T. □

[1] Upper index in parenthesis denotes a repetition of the previous bit, i.e., $01^{(3)} = 0111$.

[2] In the Shift-Or algorithm, transitions from states in our figures are implemented by operation **shl**() (shift to the left) because of easier implementation in the case when number of states of *NFA* is greater than length of computer word and vectors R have to be divided into two or more bit vectors.

6.2.2 Approximate String Matching Using Hamming Distance

In the approximate string matching using the Hamming distance, vectors R_i^l, $0 \le l \le k$, $0 \le i \le n$, are computed as follows:

$$
\begin{array}{lcll}
r_{j,0}^l & \leftarrow & 1, & 0 < j \le m, 0 \le l \le k \\
R_i^0 & \leftarrow & \mathbf{shl}(R_{i-1}^0) \text{ OR } D[t_i], & 0 < i \le n \\
R_i^l & \leftarrow & (\mathbf{shl}(R_{i-1}^l) \text{ OR } D[t_i]) \text{ AND } \mathbf{shl}(R_{i-1}^{l-1}), & 0 < i \le n, 0 < l \le k
\end{array}
\tag{6.3}
$$

In Formula 6.3 operation AND is the bitwise operation `and`. Term $\mathbf{shl}(R_{i-1}^l)$ OR $D[t_i]$ represents matching and term $\mathbf{shl}(R_{i-1}^{l-1})$ represents edit operation *replace*—position i in text T is increased, position in pattern P is increased, and edit distance l is increased. Pattern P is found with at most k mismatches at position $t_{i-m+1} \dots t_i$ if $r_{m,i}^k = 0$, $0 < i \le n$. The maximum number of mismatches of the found string is $D_H(P, t_{i-m+1} \dots t_i) = l$, where l is the minimum number such that $r_{m,i}^l = 0$.

An example of matrices R^l for searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ with at most $k = 3$ mismatches is shown in Table 6.3.

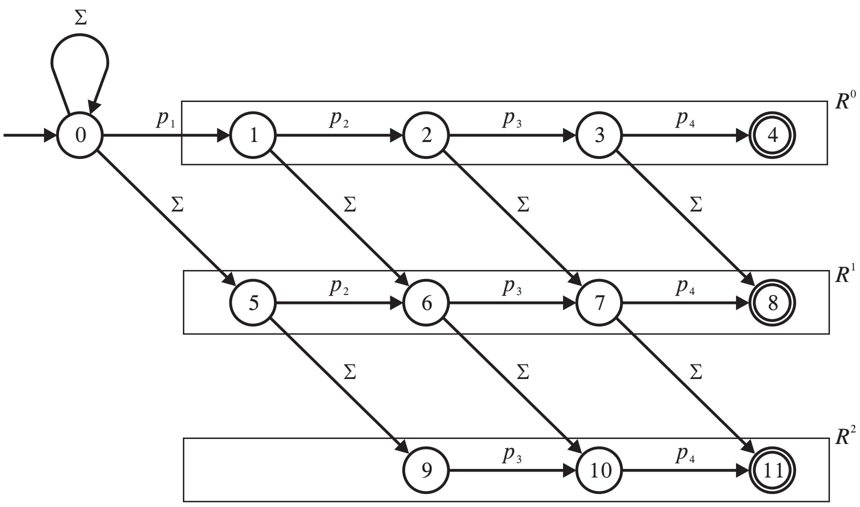


Figure 6.1: Bit parallelism uses one bit vector R for each level of states of *NFA*.

Theorem 6.3
Shift-Or algorithm described by Formula 6.3 simulates a run of the *NFA* for the approximate string matching using the Hamming distance constructed by Algorithm 3.9 (Σ version).

Proof
In Shift-Or algorithm for the approximate string matching there is for each level l, $0 \le l \le k$, of states of *NFA* one bit vector R_i^l, $0 \le i \le n$. So in Shift-Or algorithm, the set of active states from Chapter 4 is implemented by bit vectors—one vector for each level of states.

In Formula 6.3 term $\mathbf{shl}(R_{i-1}^l)$ OR $D[t_i]$ represents matching transition (See the proof of Theorem 6.2). Term $\mathbf{shl}(R_{i-1}^{l-1})$ represents transition *replace*—each active state of level $l-1$ is moved to the next depth in level l.

The self-loop of the initial state is implemented by inserting 0 at the beginning of vector R^0 within operation $\mathbf{shl}()$. 0 is inserted also at the beginning of each vector R^l, $0 < l \le k$. Since

R^0	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
d	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**
R^1	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	1	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1
b	1	1	1	**0**	1	1	1	1	1	**0**	1	1	**0**	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**
R^2	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	1	1	**0**	1	**0**	**0**	1	1	**0**	**0**	1	**0**	**0**	1	1
b	1	1	1	1	**0**	1	1	1	1	1	**0**	1	1	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**
R^3	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	1	1	1	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	1
c	1	1	1	1	1	**0**	**0**	1	1	1	1	**0**	1	1	**0**	1
a	1	1	1	1	1	1	1	**0**	1	1	1	1	1	1	1	**0**

Table 6.3: Matrices R^l for the approximate string matching using the Hamming distance (Σ version, $P = adbbca$, $k = 3$, and $T = adcabcaabadbbca$).

the first state of l-th level is connected with the initial state by the sequence of l transitions labeled by Σ, each of these first states is active from l-th step of the simulation respectively till the end. The impact of 0s inserted at the beginning of vectors R^l does not appear before l-th step, therefore also vectors R^l simulate correctly the *NFA*.

If $r^l_{m,i} = 0$, $0 < i \leq n$, the final state of l-th level is active and we can report that the pattern is found with at most l errors ending at position i in the text. In fact, we report just only the minimum l in each step. $\square$

For the simulation of $\overline{p}$ version of the *NFA* for the approximate string matching using the Hamming distance constructed by Algorithm 3.8 we can use Formula 6.4. To prevent inserting 0s into vectors R^l, $0 < l \leq k$, we use auxiliary vector E defined by Formula 6.5.

$$
\begin{array}{llll}
r_{j,0}^{l} & \leftarrow & 1, & 0 < j \leq m, 0 \leq l \leq k \\
R_i^0 & \leftarrow & \mathbf{shl}(R_{i-1}^0) \text{ OR } D[t_i], & 0 < i \leq n \\
R_i^1 & \leftarrow & (\mathbf{shl}(R_{i-1}^1) \text{ OR } D[t_i] \text{ OR } E) & \\
 & & \quad \text{AND } (\mathbf{shl}(R_{i-1}^0) \text{ OR NOT}(D[t_i])), & 0 < i \leq n \\
R_i^l & \leftarrow & (\mathbf{shl}(R_{i-1}^l) \text{ OR } D[t_i] \text{ OR } E) & \\
 & & \quad \text{AND } (\mathbf{shl}(R_{i-1}^{l-1}) \text{ OR NOT}(D[t_i]) \text{ OR } E), & 0 < i \leq n, 0 < l \leq k
\end{array}
\tag{6.4}
$$

$$
E = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{bmatrix}, \text{ where } e_1 = 1 \text{ and } e_j = 0, \forall j, 1 < j \leq m. \tag{6.5}
$$

In Formula 6.4 the bitwise operation NOT() exchanges all 0s and 1s in bit vector.

Simulation of $\overline{p}$ version of the the *NFA* for the approximate string matching using the Hamming distance for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ with at most $k = 3$ mismatches is shown in Table 6.4.

Theorem 6.4
Shift-Or algorithm described by Formula 6.4 simulates a run of the *NFA* for the approximate string matching using the Hamming distance constructed by Algorithm 3.8 ($\overline{p}$ version).

Proof
The simulation is the same like in the proof of Theorem 6.3 except *replace* transition. In this case, *replace* transition is labeled by $\overline{p}_j$, $0 < j \leq m$, therefore when shifting vector R_{i-1}^{l-1} only the *replace* transitions labeled by t_i must be selected by performing OR operation with NOT$(D[t_i])$.

The problem of inserting 0 at the beginning of vector R^l, $0 < l \leq k$, during **shl**() operation is solved by setting these bits by term OR E. □

6.2.3 Approximate String Matching Using Levenshtein Distance

In the approximate string matching using the Levenshtein distance, vectors R_i^l, $0 \leq l \leq k$, $0 \leq i \leq n$, are computed by Formula 6.6[3]. To prevent *insert* transitions leading into final states we use auxiliary vector V defined by Formula 6.7.

$$
\begin{array}{llll}
r_{j,0}^{l} & \leftarrow & 0, & 0 < j \leq l, 0 < l \leq k \\
r_{j,0}^{l} & \leftarrow & 1, & l < j \leq m, 0 \leq l \leq k \\
R_i^0 & \leftarrow & \mathbf{shl}(R_{i-1}^0) \text{ OR } D[t_i], & 0 < i \leq n \\
R_i^l & \leftarrow & (\mathbf{shl}(R_{i-1}^l) \text{ OR } D[t_i]) & \\
 & & \quad \text{AND } \mathbf{shl}(R_{i-1}^{l-1} \text{ AND } R_i^{l-1}) \text{ AND } (R_{i-1}^{l-1} \text{ OR } V), & 0 < i \leq n, 0 < l \leq k
\end{array}
\tag{6.6}
$$

$$
V = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix}, \text{ where } v_m = 1 \text{ and } v_j = 0, \forall j, 1 \leq j < m. \tag{6.7}
$$

[3]Similar formula for Shift-And simulation is shown in [WM92]. Since they consider also trivial occurrences, they do not use any term equivalent to term OR V.

R^0	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
d	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**

R^1	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	1	**0**	**0**	1	**0**	**0**	1	1	**0**	1	**0**	**0**	**0**	**0**	1
d	1	1	1	1	1	**0**	1	1	**0**	**0**	1	1	1	1	1	1
b	1	1	1	**0**	1	1	1	1	1	**0**	1	1	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

R^2	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
d	1	1	1	**0**	**0**	1	**0**	**0**	1	1	**0**	1	**0**	**0**	**0**	**0**
b	1	1	1	1	1	**0**	**0**	1	1	1	**0**	1	1	**0**	1	1
b	1	1	1	1	**0**	1	1	1	1	1	**0**	1	1	1	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

R^3	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
d	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
b	1	1	1	1	**0**	1	1	**0**	**0**	1	1	**0**	1	1	**0**	**0**
b	1	1	1	1	1	**0**	**0**	**0**	1	**0**	1	**0**	**0**	1	**0**	1
c	1	1	1	1	1	**0**	**0**	1	1	1	1	**0**	1	1	1	1
a	1	1	1	1	1	1	1	**0**	1	1	1	1	1	1	1	1

Table 6.4: Matrices R^l for the approximate string matching using the Hamming distance ($\overline{p}$ version, $P = adbbca$, $k = 3$, and $T = adcabcaabadbbca$).

In Formula 6.6 term $\mathbf{shl}(R^l_{i-1})$ OR $D[t_i]$ represents matching, term $\mathbf{shl}(R^{l-1}_{i-1})$ represents edit operation *replace*, term $\mathbf{shl}(R^{l-1}_i)$ represents edit operation *delete*—position in pattern P is increased, position in text T is not increased, and edit distance l is increased. Term R^{l-1}_{i-1} represents edit operation *insert*—position in pattern P is not increased, position in text T is increased, and edit distance l is increased. Term OR V provides that no *insert* transition leads from any final state.

Pattern P is found with at most k differences ending at position i if $r^k_{m,i} = 0$, $0 < i \leq n$. The maximum number of differences of the found string is l, where l is the minimum number such that $r^l_{m,i} = 0$.

An example of matrices R^l for searching for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ with at most $k = 3$ errors is shown in Table 6.5.

R^0	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
d	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**

R^1	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	1	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	1	1	**0**
b	1	1	**0**	**0**	1	**0**	1	1	1	**0**	1	**0**	**0**	**0**	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	**0**	**0**	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	**0**	**0**
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	**0**

R^2	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	1	**0**	**0**	**0**	**0**	**0**	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
c	1	1	1	**0**	1	1	**0**	1	1	1	1	1	**0**	**0**	**0**	**0**
a	1	1	1	1	**0**	1	1	**0**	1	1	1	1	1	**0**	**0**	**0**

R^3	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
c	1	1	**0**	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
a	1	1	1	**0**	**0**	1	**0**	**0**	**0**	1	**0**	1	**0**	**0**	**0**	**0**

Table 6.5: Matrices R^l for the approximate string matching using the Levenshtein distance (Σ version, $P = adbbca$, $k = 3$, and $T = adcabcaabadbbca$).

Theorem 6.5
Shift-Or algorithm described by Formula 6.6 simulates a run of the *NFA* for the approximate string matching using the Levenshtein distance constructed by Algorithm 3.11 (Σ version).

Proof
The proof is similar to the proof of Theorem 6.3, we only have to add the simulation of *insert* and *delete* transitions.

In Formula 6.6 term $\mathbf{shl}(R^l_{i-1})$ OR $D[t_i]$ represents matching transition and term $\mathbf{shl}(R^{l-1}_{i-1})$ represents transition *replace* (See the proof of Theorem 6.3). Term R^{l-1}_{i-1} represents transition *insert*—each active state of level $l-1$ is moved into level l within the same depth. Term $\mathbf{shl}(R^{l-1}_{i})$ represents transition *delete*—each active state of level $l-1$ is moved to the next depth in level l while no symbol is read from the input (position i in text T is the same).

At the beginning, only the initial state q_0 and all states located on the same ε-diagonal like q_0 are active (i.e., all states of $\varepsilon CLOSURE(\{q_0\})$ are active), therefore l-th bit of vector R^l,

$0 < l \leq k$, is 0 in the initial setting of the vector. The states in front of l-th bit in vector R^l can also be 0, since they have no impact (l-th bit is always 0, since all states of $\varepsilon CLOSURE(\{q_0\})$ are always active due to the self-loop of q_0). Therefore the bits behind l-th bit are set in the initial setting while the initial setting of the bits in front of l-th bit can be arbitrary.

In the case of the Levenshtein distance, the situation with inserting 0s at the beginning of vectors R^l, $0 < l \leq k$, during **shl**() operation is slightly different. Since all the states of the ε-diagonal leading from q_0 are always active, there is no impact of these 0 insertions. □

For the simulation of $\overline{p}$ version of the *NFA* for the approximate string matching using the Levenshtein distance constructed by Algorithm 3.10 we can use Formula 6.8.

$$
\begin{array}{lcll}
r^l_{j,0} & \leftarrow & 1, & 0 < j \leq m, 0 \leq l \leq k \\
R^0_i & \leftarrow & \mathbf{shl}(R^0_{i-1}) \text{ OR } D[t_i], & 0 < i \leq n \\
R^l_i & \leftarrow & (\mathbf{shl}(R^l_{i-1}) \text{ OR } D[t_i]) & \\
 & & \quad \text{AND } (\mathbf{shl}(R^{l-1}_{i-1}) \text{ OR NOT}(D[t_i])) & \\
 & & \quad \text{AND } \mathbf{shl}(R^{l-1}_i) & \\
 & & \quad \text{AND } (R^{l-1}_{i-1} \text{ OR NOT}(\mathbf{shr}(D[t_i])) \text{ OR } V), & 0 < i \leq n, 0 < l \leq k
\end{array}
\tag{6.8}
$$

In Formula 6.8 operation **shr**() is the bitwise operation `right shift`.

Simulation of $\overline{p}$ version of the *NFA* for the approximate string matching using the Levenshtein distance for pattern $P = adbbca$ in text $T = adcabcaabadbbca$ with at most $k = 3$ errors is shown in Table 6.6.

Theorem 6.6
Shift-Or algorithm described by Formula 6.8 simulates a run of the *NFA* for the approximate string matching using the Levenshtein distance constructed by Algorithm 3.10 ($\overline{p}$ version).

Proof
The proof is similar as the proof of Theorem 6.4. We have only to add the simulation of *insert* and *delete* transitions. The transition *delete* is simulated as shown in the proof of Theorem 6.5.

The simulation of *insert* transition differs from the simulation of the *insert* transition in the proof of Theorem 6.5 only in such a way, that in this case the *insert* transition is labeled by $\overline{p}$ (and not by Σ). Therefore we have to select the relevant *insert* transitions (according to the input symbol t_i) by term OR NOT(**shr**($D[t_i]$)): OR $D[t_i]$ selects the matching transitions labeled by t_i; OR NOT(**shr**($D[t_i]$)) selects the *insert* transitions labeled by t_i ($D[t_i]$ must be shifted to the right since *insert* transition leads only within its depth); *insert* transition is labeled by the symbols mismatching the symbol used as the label of the matching transition leading from the same state, from which the *insert* transition leads. □

6.2.4 Approximate String Matching Using Generalized Levenshtein Distance

In order to construct Shift-Or algorithm for the approximate string matching using the generalized Levenshtein distance [Hol97], we modify Formula 6.6 for the approximate string matching using the Levenshtein distance such that we add the term representing edit operation *transpose*. Since *NFA* for the approximate string matching using the generalized Levenshtein distance has on each transition *transpose* one auxiliary state (See Figure 3.11), we have to introduce new bit vectors S^l_i, $0 \leq l < k$, $0 \leq i < n$, as follows:

R^0	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
d	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**

R^1	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	1	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	1	1	1	**0**
b	1	1	**0**	**0**	1	**0**	1	1	1	**0**	1	**0**	**0**	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	**0**	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	**0**

R^2	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1	**0**
b	1	1	**0**	**0**	**0**	**0**	**0**	1	1	**0**	**0**	**0**	**0**	**0**	1	1
c	1	1	1	**0**	1	1	**0**	1	1	1	1	1	**0**	**0**	**0**	1
a	1	1	1	1	**0**	1	1	**0**	1	1	1	1	1	**0**	**0**	**0**

R^3	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
c	1	1	**0**	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	1
a	1	1	1	**0**	**0**	1	**0**	**0**	**0**	1	**0**	1	**0**	**0**	**0**	**0**

Table 6.6: Matrices R^l for the approximate string matching using the Levenshtein distance ($\overline{p}$ version, $P = adbbca$, $k = 3$, and $T = adcabcaabadbbca$).

$$S_i^l = \begin{bmatrix} s_{1,i}^l \\ s_{2,i}^l \\ \vdots \\ s_{m,i}^l \end{bmatrix}, 0 \leq l < k, 0 \leq i < n. \tag{6.9}$$

Vectors R_i^l, $0 \leq l \leq k$, $0 \leq i \leq n$, and S_i^l, $0 \leq l < k$, $0 \leq i < n$, are then computed as follows:

$$
\begin{array}{lcll}
r_{j,0}^{l} & \leftarrow & 0, & 0 < j \leq l, 0 < l \leq k \\
r_{j,0}^{l} & \leftarrow & 1, & l < j \leq m, 0 \leq l \leq k \\
R_i^0 & \leftarrow & \mathbf{shl}(R_{i-1}^0) \text{ OR } D[t_i], & 0 < i \leq n \\
R_i^l & \leftarrow & (\mathbf{shl}(R_{i-1}^l) \text{ OR } D[t_i]) & \\
 & & \quad \text{AND } \mathbf{shl}(R_{i-1}^{l-1} \text{ AND } R_i^{l-1} \text{ AND } (S_{i-1}^{l-1} \text{ OR } D[t_i])) & \\
 & & \quad \text{AND } (R_{i-1}^{l-1} \text{ OR } V), & 0 < i \leq n, 0 < l \leq k \\
s_{j,0}^{l} & \leftarrow & 1, & 0 < j \leq m, 0 \leq l < k \\
S_i^l & \leftarrow & \mathbf{shl}(R_{i-1}^l) \text{ OR } \mathbf{shr}(D[t_i]), & 0 < i < n, 0 \leq l < k
\end{array}
\tag{6.10}
$$

Term $\mathbf{shl}(R_{i-1}^l)$ OR $D[t_i]$ represents matching, term $\mathbf{shl}(R_{i-1}^{l-1})$ represents edit operation *replace*, term $\mathbf{shl}(R_i^{l-1})$ represents edit operation *delete*, and term R_{i-1}^{l-1} represents edit operation *insert*.

Term $(S_{i-1}^{l-1}$ OR $D[t_i])$ represents edit operation *transpose*—position in pattern P is increased by 2, position in text T is also increased by 2 but edit distance is increased just by 1. The increase of both positions by 2 is provided using vector S_i^l.

Pattern P is found with at most k differences ending at position i if $r_{m,i}^k = 0$, $0 < i \leq n$. The maximum number of differences of the found string is l, where l is the minimum number such that $r_{m,i}^l = 0$.

An example of matrices R^l and S^l for searching for pattern $P = adbbca$ in text $T = adbcbaabadbbca$ with at most $k = 3$ errors is shown in Table 6.7 (Σ version).

Theorem 6.7
Shift-Or algorithm described by Formula 6.10 simulates a run of the *NFA* for the approximate string matching using the generalized Levenshtein distance constructed by Algorithm 3.13 (Σ version).

Proof
In Formula 6.10 term $\mathbf{shl}(R_{i-1}^l)$ OR $D[t_i]$ represents matching transition and term $\mathbf{shl}(R_{i-1}^{l-1})$ represents transition *replace*, term R_{i-1}^{l-1} represents transition *insert*, and term $\mathbf{shl}(R_i^{l-1})$ represents transition *delete* (See the proof of Theorem 6.5).

Term $(S_{i-1}^{l-1}$ OR $D[t_i])$ represents edit operation *transpose*. In this transition all states of level l are moved to the next position in the right $(\mathbf{shl}(R_{i-1}^l))$ of auxiliary level l'. Then only the transitions corresponding to input symbol t_i have to be selected. It is provided by mask vector $D[t_i]$. Since each transition leading to state of depth j is labeled by symbol p_{j+1}, we have to shift the mask vector in opposite direction than in which vector R_{i-1}^l is shifted $(\mathbf{shr}(D[t_i]))$. All states of auxiliary level l' are stored in vector $S_i^l \leftarrow \mathbf{shl}(R_{i-1}^l)$ OR $\mathbf{shr}(D[t_i])$, $0 < i < n$, $0 \leq l < k$.

In next step $i+1$ of the computation all states of auxiliary level l' are moved to the next position in the right of level $l+1$ and only the transitions corresponding to input symbol t_{i+1} are selected by mask vector $D[t_{i+1}]$. Since each transition leading to state of depth $j+1$ is labeled by symbol p_j, we have to shift mask vector $D[t_{i+1}]$ in the same direction in which vector S_i^l is shifted. Therefore we insert term $\mathbf{shl}(S_{i-1}^{l-1}$ OR $D[t_i])$ in Formula 6.6 as well as we insert the formula for computation of vector S_i^l in Formula 6.6. □

For the simulation of $\overline{p}$ version of the *NFA* for the approximate string matching using the generalized Levenshtein distance constructed by Algorithm 3.12 we can use Formula 6.11.

R^0		-	a	d	b	c	b	a	a	b	a	d	b	b	c	a
	a	1	**0**	1	1	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
	d	1	1	**0**	1	1	1	1	1	1	1	**0**	1	1	1	1
	b	1	1	1	**0**	1	1	1	1	1	1	1	**0**	1	1	1
	b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
	c	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
	a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**
S^0	a	1	1	**0**	1	1	1	1	1	1	1	**0**	1	1	1	1
	d	1	1	1	1	1	1	1	1	**0**	1	1	1	1	1	1
	b	1	1	1	**0**	1	1	1	1	1	1	1	**0**	1	1	1
	b	1	1	1	1	**0**	1	1	1	1	1	1	1	1	1	1
	c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R^1	a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	d	1	**0**	**0**	**0**	1	1	**0**	**0**	**0**	**0**	**0**	**0**	1	1	**0**
	b	1	1	**0**	**0**	**0**	1	1	1	**0**	1	**0**	**0**	**0**	1	1
	b	1	1	1	**0**	**0**	**0**	1	1	1	1	1	**0**	**0**	**0**	1
	c	1	1	1	1	**0**	**0**	1	1	1	1	1	1	**0**	**0**	**0**
	a	1	1	1	1	1	1	**0**	1	1	1	1	1	1	**0**	**0**
S^1	a	1	1	**0**	1	1	1	1	1	1	1	**0**	1	1	1	1
	d	1	1	1	**0**	1	**0**	1	1	**0**	1	1	**0**	**0**	1	1
	b	1	1	1	**0**	1	1	1	1	**0**	1	1	**0**	**0**	1	1
	b	1	1	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1
	c	1	1	1	1	1	1	**0**	1	1	1	1	1	1	1	**0**
	a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R^2	a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	d	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	b	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	b	1	1	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	c	1	1	1	**0**	**0**	**0**	**0**	1	1	1	1	**0**	**0**	**0**	**0**
	a	1	1	1	1	**0**	**0**	**0**	**0**	1	1	1	1	**0**	**0**	**0**
S^2	a	1	1	**0**	1	1	1	1	1	1	1	**0**	1	1	1	1
	d	1	1	1	**0**	1	**0**	1	1	**0**	1	1	**0**	**0**	1	1
	b	1	1	1	**0**	1	**0**	1	1	**0**	1	1	**0**	**0**	1	1
	b	1	1	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1
	c	1	1	1	1	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
	a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R^3	a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	d	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	b	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	b	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	c	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	a	1	1	1	**0**	**0**	**0**	**0**	**0**	1	**0**	1	**0**	**0**	**0**	**0**

Table 6.7: Matrices R^l and S^l for the approximate string matching using the generalized Levenshtein distance (Σ version, $P = adbbca$, $k = 3$, and $T = adbcbaabadbbca$).

$$\begin{array}{lcll}
r_{j,0}^{l} & \leftarrow & 1, & 0 < j \le m, 0 \le l \le k \\
R_i^0 & \leftarrow & \mathbf{shl}(R_{i-1}^0) \text{ OR } D[t_i], & 0 < i \le n \\
R_i^l & \leftarrow & (\mathbf{shl}(R_{i-1}^l) \text{ OR } D[t_i]) & \\
 & & \text{AND } (\mathbf{shl}(R_{i-1}^{l-1}) \text{ OR } \text{NOT}(D[t_i])) & \\
 & & \text{AND } \mathbf{shl}(R_i^{l-1} \text{ AND } (S_{i-1}^{l-1} \text{ OR } D[t_i])) & \\
 & & \text{AND } (R_{i-1}^{l-1} \text{ OR } \text{NOT}(\mathbf{shr}(D[t_i])) \text{ OR } V), & 0 < i \le n, 0 < l \le k \\
s_{j,0}^{l} & \leftarrow & 1, & 0 < j \le m, 0 \le l < k \\
S_i^l & \leftarrow & \mathbf{shl}(R_{i-1}^l) \text{ OR } \mathbf{shr}(D[t_i]), & 0 < i < n, 0 \le l < k
\end{array} \tag{6.11}$$

An example of matrices R^l and S^l for searching for pattern $P = adbbca$ in text $T = adbcbaabadbbca$ with at most $k = 3$ errors is shown in Table 6.8 ($\overline{p}$ version).

Theorem 6.8
Shift-Or algorithm described by Formula 6.11 simulates a run of the *NFA* for the approximate string matching using the generalized Levenshtein distance constructed by Algorithm 3.12 ($\overline{p}$ version).

Proof
The simulation is similar as in the proof of the Theorem 6.6. The only difference is the simulation of *transpose* transition, which is shown in the proof of the Theorem 6.7. □

6.3 Approximate Sequence Matching

In the previous sections we have presented Shift-Or algorithm for the string matching. Since *NFA* for the sequence matching is very similar to *NFA* for the string matching, we can modify this algorithm for the sequence matching. To get *NFA* for the sequence matching we have to add self-loop to each non-final state of *NFA* for the string matching [Hol97].

6.3.1 Unlimited Overlapping

If we insert term AND $(R_{i-1}^l \text{ OR } V)$ simulating the self-loops labeled by Σ into Formulae 6.2, 6.3, 6.6, and 6.10 for the string matching, we get respectively Formula 6.12 for the exact sequence matching, Formula 6.13 for the approximate sequence matching using the Hamming distance, Formula 6.14 for the approximate sequence matching using the Levenshtein distance, and Formula 6.15 for the approximate sequence matching using the generalized Levenshtein distance (Σ versions).

$$\begin{array}{lcll}
r_{j,0}^{l} & \leftarrow & 1, & 0 < j \le m, 0 \le l \le k \\
R_i^0 & \leftarrow & (\mathbf{shl}(R_{i-1}^0) \text{ OR } D[t_i]) \text{ AND } (R_{i-1}^0 \text{ OR } V), & 0 < i \le n
\end{array} \tag{6.12}$$

$$\begin{array}{lcll}
r_{j,0}^{l} & \leftarrow & 1, & 0 < j \le m, 0 \le l \le k \\
R_i^0 & \leftarrow & (\mathbf{shl}(R_{i-1}^0) \text{ OR } D[t_i]) \text{ AND } (R_{i-1}^0 \text{ OR } V), & 0 < i \le n \\
R_i^l & \leftarrow & (\mathbf{shl}(R_{i-1}^l) \text{ OR } D[t_i]) & \\
 & & \text{AND } \mathbf{shl}(R_{i-1}^{l-1}) \text{ AND } (R_{i-1}^l \text{ OR } V), & 0 < i \le n, 0 < l \le k
\end{array} \tag{6.13}$$

R^0		-	a	d	b	c	b	a	a	b	a	d	b	b	c	a
	a	1	0	1	1	1	1	0	0	1	0	1	1	1	1	0
	d	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1
	b	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1
	b	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
	c	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
	a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
S^0	a	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1
	d	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1
	b	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1
	b	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
	c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R^1	a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	d	1	0	0	1	1	1	0	0	0	0	0	1	1	1	0
	b	1	1	0	0	0	1	1	1	0	1	0	0	1	1	1
	b	1	1	1	0	0	0	1	1	1	1	1	0	0	1	1
	c	1	1	1	1	0	0	1	1	1	1	1	1	0	0	1
	a	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0
S^1	a	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1
	d	1	1	1	0	1	0	1	1	0	1	1	0	0	1	1
	b	1	1	1	0	1	1	1	1	0	1	1	0	1	1	1
	b	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
	c	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
	a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R^2	a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	d	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	b	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	b	1	1	0	0	0	0	0	1	0	0	0	0	0	1	1
	c	1	1	1	0	0	0	0	1	1	1	1	0	0	0	1
	a	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0
S^2	a	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1
	d	1	1	1	0	1	0	1	1	0	1	1	0	0	1	1
	b	1	1	1	0	1	0	1	1	0	1	1	0	0	1	1
	b	1	1	1	1	0	1	1	1	1	1	1	1	1	0	1
	c	1	1	1	1	1	1	0	0	1	0	1	1	1	1	1
	a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R^3	a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	d	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	b	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	b	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	c	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1
	a	1	1	1	0	0	0	0	0	1	0	1	0	0	0	0

Table 6.8: Matrices R^l and S^l for the approximate string matching using the generalized Levenshtein distance ($\overline{p}$ version, $P = adbbca$, $k = 3$, and $T = adbcbaabadbbca$).

$$
\begin{array}{lcll}
r^l_{j,0} & \leftarrow & 0, & 0 < j \le l, 0 < l \le k \\
r^l_{j,0} & \leftarrow & 1, & l < j \le m, 0 \le l \le k \\
R^0_i & \leftarrow & (\mathbf{shl}(R^0_{i-1}) \text{ OR } D[t_i]) \text{ AND } (R^0_{i-1} \text{ OR } V), & 0 < i \le n \\
R^l_i & \leftarrow & (\mathbf{shl}(R^l_{i-1}) \text{ OR } D[t_i]) & \\
 & & \quad \text{AND } \mathbf{shl}(R^{l-1}_{i-1} \text{ AND } R^{l-1}_i) & \\
 & & \quad \text{AND } (R^{l-1}_{i-1} \text{ OR } V) \text{ AND } (R^l_{i-1} \text{ OR } V), & 0 < i \le n, 0 < l \le k
\end{array}
\tag{6.14}
$$

$$
\begin{array}{lcll}
r^l_{j,0} & \leftarrow & 0, & 0 < j \le l, 0 < l \le k \\
r^l_{j,0} & \leftarrow & 1, & l < j \le m, 0 \le l \le k \\
R^0_i & \leftarrow & (\mathbf{shl}(R^0_{i-1}) \text{ OR } D[t_i]) \text{ AND } (R^0_{i-1} \text{ OR } V), & 0 < i \le n \\
R^l_i & \leftarrow & (\mathbf{shl}(R^l_{i-1}) \text{ OR } D[t_i]) & \\
 & & \quad \text{AND } \mathbf{shl}(R^{l-1}_{i-1} \text{ AND } R^{l-1}_i \text{ AND } (S^{l-1}_{i-1} \text{ OR } D[t_i])) & \\
 & & \quad \text{AND } (R^{l-1}_{i-1} \text{ OR } V) \text{ AND } (R^l_{i-1} \text{ OR } V), & 0 < i \le n, 0 < l \le k \\
s^l_{j,0} & \leftarrow & 1, & 0 < j \le m, 0 \le l < k \\
S^l_i & \leftarrow & \mathbf{shl}(R^l_{i-1}) \text{ OR } \mathbf{shr}(D[t_i]) \text{ AND } S^l_{i-1}, & 0 < i < n, 0 \le l < k
\end{array}
\tag{6.15}
$$

In the same way we can modify Formulae 6.4, 6.8, and 6.11 simulating $\overline{p}$ version of *NFA*s.

Lemma 6.9
Term AND $(R^l_{i-1}$ OR $V)$, $0 \le l \le k$, in Formulae 6.12, 6.13, 6.14, and 6.15 and term AND S^l_{i-1} in Formula 6.15 simulate self-loops labeled by Σ in the *NFA*s for the exact and approximate sequence matching with the unlimited overlapping.

Proof
Term AND $(R^l_{i-1}$ OR $V)$, $0 \le l \le k$, provides that old value of vector R^l is always considered, therefore if a state become active (the corresponding bit is reset), it remains active until the end of the simulation—self-loop labeled by Σ of active state is used in each step. Term OR V prevents the self-loop to be used also in final state. Term AND S^l_{i-1} provides the self-loop of the transition *transpose*. □

6.3.2 Limited Overlapping

For the approximate sequence matching with the limited overlapping we can also insert some term simulating the self-loop, which is labeled by $\overline{p}_j$, $0 < j \le m$, in this case. If we do so, we get following formula for the exact sequence matching with the limited overlapping.

$$
\begin{array}{lcll}
r^l_{j,0} & \leftarrow & 1, & 0 < j \le m, 0 \le l \le k \\
R^0_i & \leftarrow & (\mathbf{shl}(R^0_{i-1}) \text{ OR } D[t_i]) & \\
 & & \quad \text{AND } (R^0_{i-1} \text{ OR } \mathbf{shr}(\text{NOT}(D[t_i])) \text{ OR } V), & 0 < i \le n
\end{array}
\tag{6.16}
$$

Lemma 6.10
Term AND $(R^0_{i-1}$ OR $\mathbf{shr}(\text{NOT}(D[t_i]))$ OR $V)$, in Formula 6.16 simulates self-loops labeled by $\overline{p}_j$, $0 < j \le m$, in the *NFA* for the exact sequence matching with the limited overlapping.

Proof
Term R^0_{i-1} has the same meaning as in the proof of Lemma 6.9. But here, the self-loops are labeled by $\overline{p}_j$, $0 < j \leq m$, therefore we have to select only the relevant self-loops (with respect to input symbol t_i). This selection is provided by mask vector $\text{NOT}(D[t_i])$, which selects the transitions not labeled by t_i. This mask vector has to be shifted in opposite direction than in which matching transitions are shifted. This shift has to be done, since $D[t_i]$ selects the relevant incoming matching transitions (coming from the previous depth of *NFA*), but self-loop comes to the same state from which it leads here. Term OR V also prevents the self-loop to be used in final state. □

Using term $\text{AND } (R^0_{i-1} \text{ OR } \mathbf{shr}(\text{NOT}(D[t_i])) \text{ OR } V)$ simulating the self-loop labeled by $\overline{p}_j$, $0 < j \leq m$ as shown above, we can construct formulae simulating other *NFA*s for the approximate sequence matching with the limited overlapping. In the case of *NFA* for the generalized Levenshtein distance, we have also to insert term $\text{AND } (S^l_{i-1} \text{ OR } \text{NOT}(D[t_i]))$ in formula for computing vector S^l_i.

6.4 Reduced Finite Automata

In the reduced *NFA*s for the approximate string matching each level contains only $(m - k + 1)$ states, therefore it is not needed to have a bit vector of length m for each level of states. For the reduced *NFA*s we use bit vectors R^l of length $(m-k+1)$ for the Hamming distance and of length $(m - k)$ for the Levenshtein and generalized Levenshtein distances. In the latter two cases we do not need a bit for the leftmost state of each level of *NFA*, since each such state is connected with the initial state by a sequence of ε-transitions. The initial state is always active (because of its self-loop), therefore the leftmost state in each level is also always active.

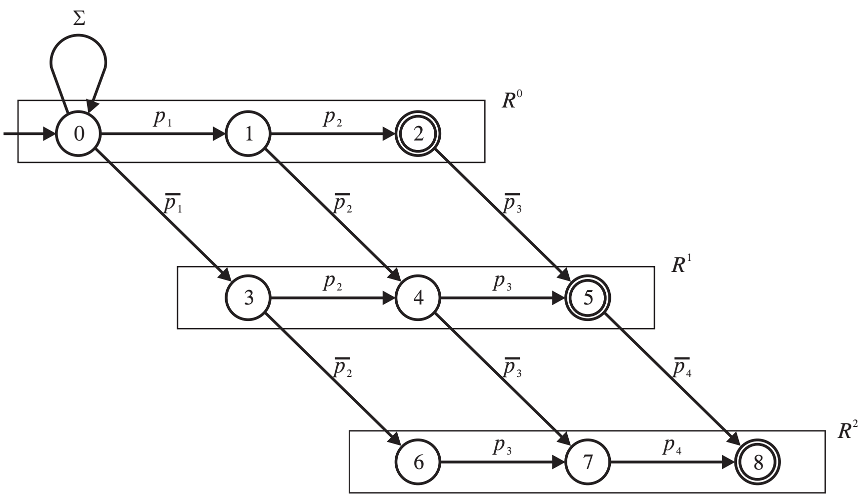


Figure 6.2: Bit parallelism for reduced *NFA*s for the approximate string matching using the Hamming distance uses for each level of states of *NFA* one shortened bit vector R of length $(m - k + 1)$ bits.

Formula 6.17 simulates the reduced *NFA* for the approximate string matching using the Hamming distance ($\overline{p}$ version), in which there is a final state at the end of each level (See Figure 6.2).

In case of simulation of *NFA* for the approximate string matching using the Hamming distance we number bits from 0 in vectors, but in all other cases we number bits from 1.

$$
\begin{array}{llll}
r^0_{0,0} & \leftarrow & 0 & \\
r^0_{j,0} & \leftarrow & 1, & 0 < j \le m-k \\
r^l_{j,0} & \leftarrow & 1, & 0 \le j \le m-k, 0 < l \le k \\
R^0_i & \leftarrow & \mathbf{shl}(R^0_{i-1}) \text{ OR } D^0[t_i], & 0 < i \le n \\
R^l_i & \leftarrow & (\mathbf{shl}(R^l_{i-1}) \text{ OR } D^l[t_i] \text{ OR } E') & \\
 & & \quad \text{AND } (R^{l-1}_{i-1} \text{ OR NOT}(D^l[t_i])), & 0 < i \le n, 0 < l \le k
\end{array}
\tag{6.17}
$$

Vector E' is defined in Formula 6.18.

$$
E' = \begin{bmatrix} e'_1 \\ e'_2 \\ \vdots \\ e'_{m-k} \end{bmatrix}, \text{ where } e'_1 = 1 \text{ and } e'_j = 0, \forall j, 1 < j \le m-k. \tag{6.18}
$$

Bit vectors $D^l[x]$ are defined as follows:

$$
D^l[x] = \begin{bmatrix} d_{1+l,x} \\ d_{2+l,x} \\ \vdots \\ d_{m-k+l,x} \end{bmatrix}, \quad 0 \le l \le k, x \in \Sigma, \tag{6.19}
$$

where bits $d_{j,x}$, $1 \le j \le m$, are the bits of the original vectors of D (Formula 6.1).

Simulation of the reduced *NFA* for the approximate string matching using the Hamming distance ($\overline{p}$ version) is shown in Table 6.9.

Formula 6.20 simulates the reduced *NFA* for the approximate string matching using the Hamming distance ($\overline{p}$ version), in which there is only one final state (it is in level k) and the last state of each level l is connected with this final state by a sequence of $(k-l)$ transitions labeled by Σ (See Figure 3.20). Vector Z is defined by Formula 6.21.

$$
\begin{array}{llll}
r^0_{0,0} & \leftarrow & 0 & \\
r^0_{j,0} & \leftarrow & 1, & 0 < j \le m-k \\
r^l_{j,0} & \leftarrow & 1, & 0 \le j \le m-k, 0 < l \le k \\
R^0_i & \leftarrow & \mathbf{shl}(R^0_{i-1}) \text{ OR } D^0[t_i], & 0 < i \le n \\
R^l_i & \leftarrow & (\mathbf{shl}(R^l_{i-1}) \text{ OR } D^l[t_i] \text{ OR } E') & \\
 & & \quad \text{AND } (R^{l-1}_{i-1} \text{ OR } (\text{NOT}(D^l[t_i]) \text{ AND } Z)), & 0 < i \le n, 0 < l \le k
\end{array}
\tag{6.20}
$$

$$
Z = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_{m-k} \end{bmatrix}, \text{ where } z_{m-k} = 0 \text{ and } z_j = 1, \forall j, 1 \le j < m-k. \tag{6.21}
$$

If $r^k_{m-k,i} = 0$, $i \ge m$, then pattern P is found with at most k mismatches at position $t_{i-m+1} \dots t_i$. In this case we also have to check, whether the found occurrence does not start in front of the beginning of the input text.

R^0	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
a	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
d	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1	1
R^1	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	1	1	**0**	**0**	1	**0**	**0**	1	1	**0**	1	**0**	**0**	**0**	**0**	1
d	1	1	1	1	1	**0**	1	1	**0**	**0**	1	1	1	1	1	1
b	1	1	1	**0**	1	1	1	1	1	**0**	1	1	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R^2	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	1	1	1	**0**	**0**	1	**0**	**0**	1	1	**0**	1	**0**	**0**	**0**	**0**
b	1	1	1	1	1	**0**	**0**	1	1	1	**0**	1	1	**0**	1	1
b	1	1	1	1	**0**	1	1	1	1	1	**0**	1	1	1	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R^3	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	1	1	1	1	**0**	1	1	**0**	**0**	1	1	**0**	1	1	**0**	**0**
b	1	1	1	1	1	**0**	**0**	**0**	1	**0**	1	**0**	**0**	1	**0**	1
c	1	1	1	1	1	**0**	**0**	1	1	1	1	**0**	1	1	1	1
a	1	1	1	1	1	1	1	**0**	1	1	1	1	1	1	1	1

Table 6.9: Matrices R^l for the approximate string matching using the Hamming distance (reduced *NFA*, $\overline{p}$ version, $P = adbbca$, $k = 3$, and $T = adcabcaabadbbca$).

Simulation of the reduced *NFA* for the approximate string matching using the Hamming distance ($\overline{p}$ version) with one final state is shown in Table 6.10.

Theorem 6.11
Shift-Or algorithm described by Formula 6.17 as well as that one described by Formula 6.20 simulates a run of the reduced *NFA* for the approximate string matching using the Hamming distance ($\overline{p}$ version) described in Section 3.6.

Proof
In Formulae 6.17 and 6.20 a state of depth j and level l has the same position in bit vector R^l as a position of a state of depth $j-1$ and level $l+1$ in vector R^{l+1}. Therefore when simulating *replace* transition we do not need **shl**() operation. This is also other main difference between Formula 6.4 for non-reduced *NFA* and Formula 6.17 for reduced *NFA*. The other difference is the use of shortened mask vectors.

If $r^l_{m-k,i} = 0$, it means that the final state at level l is active, therefore after reading $k - l$ symbols we reach the end of an occurrence of the pattern P with at most k errors. It also implies that the found occurrence has less than l errors.

The *NFA* with only one final state differs from the previous *NFA* only in *replace* transition connecting the last states of levels. In this case these transitions are labeled by Σ, therefore we insert term AND Z, which provides using the *replace* transition for all symbols of input alphabet Σ. In this case we also check only one final state (of level k). □

R^0	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
a	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
d	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1	1
R^1	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	1	1	**0**	**0**	1	**0**	**0**	1	1	**0**	1	**0**	**0**	**0**	**0**	1
d	1	1	1	1	1	**0**	1	1	**0**	**0**	1	1	1	1	1	1
b	1	1	1	**0**	1	1	1	1	1	**0**	1	1	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
R^2	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	1	1	1	**0**	**0**	1	**0**	**0**	1	1	**0**	1	**0**	**0**	**0**	**0**
b	1	1	1	1	1	**0**	**0**	1	1	1	**0**	1	1	**0**	1	1
b	1	1	1	1	**0**	1	1	1	1	1	**0**	1	1	1	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
R^3	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	1	1	1	1	**0**	1	1	**0**	**0**	1	1	**0**	1	1	**0**	**0**
b	1	1	1	1	1	**0**	**0**	**0**	1	**0**	1	**0**	**0**	1	**0**	1
c	1	1	1	1	1	**0**	**0**	1	1	1	1	**0**	1	1	1	1
a	1	1	1	1	1	1	1	**0**	1	1	1	1	1	1	1	**0**

Table 6.10: Matrices R^l for the approximate string matching using the Hamming distance (reduced *NFA* with one final state, $\overline{p}$ version, $P = adbbca$, $k = 3$, and $T = adcabcaabadbbca$).

In Σ version of *NFA* for the approximate string matching using the Hamming distance we have only one final state and the *replace* transitions are labeled by Σ. Formula 6.22 simulates such *NFA*.

$$\begin{array}{rcll}
r^0_{0,0} & \leftarrow & 0 & \\
r^0_{j,0} & \leftarrow & 1, & 0 < j \leq m-k \\
r^l_{j,0} & \leftarrow & 1, & 0 \leq j \leq m-k, 0 < l \leq k \\
R^0_i & \leftarrow & \mathbf{shl}(R^0_{i-1}) \text{ OR } D^0[t_i], & 0 < i \leq n \\
R^l_i & \leftarrow & (\mathbf{shl}(R^l_{i-1}) \text{ OR } D^l[t_i]) \text{ AND } R^{l-1}_{i-1}, & 0 < i \leq n, 0 < l \leq k
\end{array} \tag{6.22}$$

Since first l bits of each vector R^l, $0 \leq l \leq k$, are removed, a new problem appears. Because of inserting 0s at the beginning of each vector R^l by operation $\mathbf{shl}()$, a bit for state of depth $l+1$ and level l is 0 whenever $p_{l+1} = t_i$. It represents that there are l mismatching symbols in input text T followed by one matching symbol $p_{l+1} = t_i$. It is true except the case that $i < l+1$—pattern cannot start in front of the first symbol of text T. Therefore we can say that if $r^l_{m-k,i} = 0$, $0 \leq l \leq k$, $m-k+l \leq i \leq n-k+l$, then pattern P is found with at most k mismatches at position $t_{i+k-l-m+1} \ldots t_{i+k-l}$. This problem does not appear in the simulation of $\overline{p}$ version of *NFA* because of term OR E'.

Simulation of the reduced *NFA* for the approximate string matching using the Hamming distance (Σ version) is shown in Table 6.11.

R^0	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
a	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
d	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1	1
R^1	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	1	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1
b	1	1	1	**0**	1	1	1	1	1	**0**	1	1	**0**	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
R^2	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	1	1	**0**	1	**0**	**0**	1	1	**0**	**0**	1	**0**	**0**	1	1
b	1	1	1	1	**0**	1	1	1	1	1	**0**	1	1	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
R^3	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	1	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	1	1	1	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	1
c	1	1	1	1	1	**0**	**0**	1	1	1	1	**0**	1	1	**0**	1
a	1	1	1	1	1	1	1	**0**	1	1	1	1	1	1	1	**0**

Table 6.11: Matrices R^l for the approximate string matching using the Hamming distance (reduced *NFA*, Σ version, $P = adbbca$, $k = 3$, and $T = adcabcaabadbbca$).

Theorem 6.12
Shift-Or algorithm described by Formula 6.22 simulates a run of the reduced *NFA* for the approximate string matching using the Hamming distance (Σ version) described in Section 3.6.

Proof
The simulation of this *NFA* differs from the simulation of the equivalent non-reduced *NFA* only in the direction of *replace* transitions as described at the beginning of the proof of Theorem 6.11. Therefore the difference between Formula 6.3 (for non-reduced *NFA*) and Formula 6.22 (for reduced *NFA*) is only omitting **shl**() operation in the simulation of *replace* transition. □

In the approximate string matching using the Hamming distance the reduction of states shortens vectors R^l and simplifies the formula for their computing—one operation **shl**() is omitted. On the other hand a new operation appears and it is shift of mask vector $D[t_i]$ by l positions in order to get vector $D^l[t_i]$. This operation is not needed in the case that we accept higher space complexity and store k matrices D^l, $0 \leq l \leq k$, each shifted by l positions.

In the case of reduced *NFA* for the approximate string matching using the Levenshtein distance we do not need change final states (in both Σ and $\overline{p}$ version). Formula 6.23 simulates the reduced *NFA* for the approximate string matching using the Levenshtein distance (Σ version).

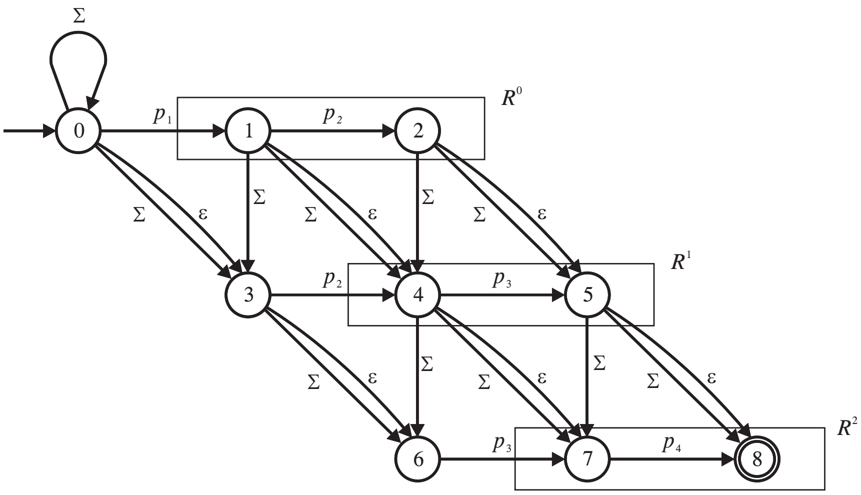


Figure 6.3: Bit parallelism for reduced *NFA*s for the approximate string matching using the Levenshtein distance uses for each level of states of *NFA* one shortened bit vector R of length $(m-k)$ bits.

$$\begin{array}{rcll} r^l_{j,0} & \leftarrow & 1, & 0 < j \leq m-k, 0 \leq l \leq k \\ R^0_i & \leftarrow & \mathbf{shl}(R^0_{i-1}) \text{ OR } D^0[t_i], & 0 < i \leq n \\ R^l_i & \leftarrow & (\mathbf{shl}(R^l_{i-1}) \text{ OR } D^l[t_i]) & \\ & & \text{AND } R^{l-1}_{i-1} \text{ AND } R^{l-1}_i \text{ AND } (\mathbf{shr}(R^{l-1}_{i-1}) \text{ OR } V'), & 0 < i \leq n, 0 < l \leq k \end{array} \quad (6.23)$$

$$V' = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_{m-k} \end{bmatrix}, \text{ where } v_{m-k} = 1 \text{ and } v_j = 0, \forall j, 1 \leq j < m-k. \quad (6.24)$$

If $r^k_{m-k,i} = 0$, $0 < i \leq n$, then pattern P is found with at most k differences ending at position i.

Simulation of the reduced *NFA* for the approximate string matching using the Levenshtein distance (Σ version) is shown in Table 6.12.

Theorem 6.13

Shift-Or algorithm described by Formula 6.23 simulates a run of the reduced *NFA* for the approximate string matching using the Levenshtein distance (Σ version) described in Section 3.6.

Proof

The simulation is similar to the one described in the proof of Theorem 6.12 but in this case we have to add the simulation of edit operation *insert* and *delete*.

Transition *insert* leads only within the same depth but while depth j, $l \leq j \leq l+m-k$, in vector R^l is represented by $(j-l)$-th bit, in vector R^{l+1} it is represented by $(j-l-1)$-th bit. Therefore term R^{l-1}_{i-1} (from Formula 6.3) simulating *insert* transition must be shifted by operation $\mathbf{shr}()$.

Transition *delete* leads to the next depth of the following level, so in this case term R^{l-1}_i is not needed to be shifted by $\mathbf{shl}()$. $\square$

R^0	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
d	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1	1
R^1	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
d	1	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	1	1	**0**
b	1	1	**0**	**0**	1	**0**	1	1	1	**0**	1	**0**	**0**	**0**	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	**0**	1	1
R^2	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
b	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	1	**0**	**0**	**0**	**0**	**0**	1	1	**0**	**0**	**0**	**0**	**0**	**0**	1
c	1	1	1	**0**	1	1	**0**	1	1	1	1	1	**0**	**0**	**0**	1
R^3	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
b	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
c	1	1	**0**	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
a	1	1	1	**0**	**0**	1	**0**	**0**	**0**	1	**0**	1	**0**	**0**	**0**	**0**

Table 6.12: Matrices R^l for the approximate string matching using the Levenshtein distance (reduced *NFA*, Σ version, $P = adbbca$, $k = 3$, and $T = adcabcaabadbbca$).

In the case of the approximate string matching using the Levenshtein distance the reduction of states of *NFA* only shortens vectors R^l, but does not simplify the formula for the Shift-Or algorithm.

Σ version of reduced *NFA* for the approximate string matching using the generalized Levenshtein distance can be simulated by Formula 6.25.

$$
\begin{array}{lcll}
r^l_{j,0} & \leftarrow & 1, & 0 < j \leq m-k, 0 \leq l \leq k \\
R^0_i & \leftarrow & \mathbf{shl}(R^0_{i-1}) \text{ OR } D^0[t_i], & 0 < i \leq n \\
R^l_i & \leftarrow & (\mathbf{shl}(R^l_{i-1}) \text{ OR } D^l[t_i]) & \\
 & & \quad \text{AND } R^{l-1}_{i-1} \text{ AND } R^{l-1}_i \text{ AND } (\mathbf{shr}(R^{l-1}_{i-1}) \text{ OR } V') & \\
 & & \quad \text{AND } (S^{l-1}_{i-1} \text{ OR } D^{l-1}[t_i]), & 0 < i \leq n, 0 < l \leq k \\
s^l_{j,0} & \leftarrow & 1, & 0 < j \leq m-k, 0 \leq l \leq k \\
S^l_i & \leftarrow & \mathbf{shl}(R^l_{i-1}) \text{ OR } D^{l+1}[t_i], & 0 < i < n, 0 \leq l < k
\end{array}
\qquad (6.25)
$$

Theorem 6.14
Shift-Or algorithm described by Formula 6.25 simulates a run of the reduced *NFA* for the approximate string matching using the generalized Levenshtein distance (Σ version) described in Section 3.6.

Proof
The simulation is the same like in the proof of Theorem 6.13, but we have to add simulation of transition *transpose*. Transition *transpose* leads from state of j-th depth, $0 < j < m-1$, and level l, $0 \leq l < k$, via immediate state into a state of depth $j+2$ and level $l+1$. Since i-th bit, $1 < i \leq m-k$, of vector R^l, $0 \leq l < k$, represents a state of the same depth as a state represented by $(i-1)$-th bit of vector R^{l+1}, we have to remove one $\mathbf{shl}()$ operation from the simulation of transition *transpose* in Formula 6.10. $\square$

In the same way as shown above we can also construct formulae for Shift-Or algorithm simulation of $\overline{p}$ versions of reduced *NFA*s of the approximate string matching using the Levenshtein distance and using the generalized Levenshtein distance as well as the approximate sequence matching.

6.5 String and Sequence Verification

*NFA*s for the string and sequence matching and *NFA*s for the string and sequence verification differs in the absence of self-loop of the initial state. This self-loop is implemented by inserting 0s at the beginning of vectors R^l in **shl**() operation of Shift-Or algorithm. If, in the formulae of this chapter, we use a different left-shift operation, which does not insert 0s but 1s, we get the formulae for the verification. If only **shl**() operation inserting 0s is available, we can use Shift-And algorithm. The last possibility is to set bits at the beginning of vector R^0 in each step i, $1 < i < n$ (the bits of vectors R^l, $0 < l \leq k$, should be set in each step i, $1 \leq i < n$). That can be provided by term OR E, where vector E is defined by Formula 6.5.

*NFA*s for the approximate string matching using the Levenshtein and generalized Levenshtein distances differ from *NFA*s for the approximate string verification using the Levenshtein and generalized Levenshtein distances also in the states under 0-th diagonal. Except for simulation of reduced *NFA*s, the simulation methods have also bits for these states and therefore they can be simulated without any change of formulae.

6.6 Nonuniform Weights of Edit Operations

The weighted approximate string matching inserts positive integer weights (w_R, w_I, w_D, and w_T) for edit operations (*replace*, *insert*, *delete*, and *transpose* respectively). In this section we consider only the weights independent on the symbols involved in the corresponding operations. Therefore formula for the approximate string matching using the Hamming distance is the same as in the ordinary approximate string matching, but each level l represents cost $(l * w_R)$.

For the weighted approximate string matching using the Levenshtein and generalized Levenshtein distance we must provide that transition *replace* leads from level l into level $(l + w_R)$, transition *insert* into level $(l + w_I)$, transition *delete* into level $(l + w_D)$, and transition *transpose* into level $(l + w_T)$. Formulae 6.26 and 6.27 show Shift-Or simulation of Σ versions of the weighted approximate string matching using the Levenshtein and generalized Levenshtein distance respectively.

$$\begin{array}{lcll}
r^l_{j,0} & \leftarrow & 0, & 0 < j \leq l, 0 < l \leq k \\
r^l_{j,0} & \leftarrow & 1, & l < j \leq m, 0 \leq l \leq k \\
R^0_i & \leftarrow & \mathbf{shl}(R^0_{i-1}) \text{ OR } D[t_i], & 0 < i \leq n \\
R^l_i & \leftarrow & (\mathbf{shl}(R^l_{i-1}) \text{ OR } D[t_i]) & \\
 & & \text{AND } \mathbf{shl}(R^{l-w_R}_{i-1} \text{ AND } R^{l-w_D}_i) & \\
 & & \text{AND } (R^{l-w_I}_{i-1} \text{ OR } V), & 0 < i \leq n, 0 < l \leq k
\end{array} \tag{6.26}$$

$$
\begin{array}{lcll}
r^l_{j,0} & \leftarrow & 0, & 0 < j \le l, 0 < l \le k \\
r^l_{j,0} & \leftarrow & 1, & l < j \le m, 0 \le l \le k \\
R^0_i & \leftarrow & \mathbf{shl}(R^0_{i-1}) \text{ OR } D[t_i], & 0 < i \le n \\
R^l_i & \leftarrow & (\mathbf{shl}(R^l_{i-1}) \text{ OR } D[t_i]) & \\
 & & \quad \text{AND } \mathbf{shl}(R^{l-w_R}_{i-1} \text{ AND } R^{l-w_D}_i \text{ AND } (S^{l-w_T}_{i-1} \text{ OR } D[t_i])) & \\
 & & \quad \text{AND } (R^{l-w_I}_{i-1} \text{ OR } V), & 0 < i \le n, 0 < l \le k \\
s^l_{j,0} & \leftarrow & 1, & 0 < j \le m, 0 \le l < k \\
S^l_i & \leftarrow & \mathbf{shl}(R^l_{i-1}) \text{ OR } \mathbf{shr}(D[t_i]), & 0 < i < n, 0 \le l < k
\end{array}
\tag{6.27}
$$

6.7 Other Methods of Bit Parallelism

In the previous sections we described only Shift-Or algorithm. If we exchange the meaning of 0s and 1s and the usage of ANDs and ORs in the formulae presented, we get Shift-And algorithm [WM92].

Other method of bit parallelism is Shift-Add algorithm [BYG92]. In this algorithm we have one bit-vector, which contains m blocks of bits of size $b = \lceil \log_2 m \rceil$ (one block for each depth of *NFA* for the approximate string matching using the Hamming distance). The formula for computing such vector then consists of shifting the vector by b bits and adding with the mask vector for current input symbol t. This vector contains 1 in $(b.j)$-th position, if $t \neq p_{j+1}$, or 0, elsewhere. This algorithm runs in time $\mathcal{O}(\lceil \frac{mb}{w} \rceil n)$.

We can also use Shift-Add algorithm for the weighted approximate string matching using the Hamming distance. In such case each block of bits in mask vector contains binary representation of weight of the corresponding edit operation *replace*. We have also to enlarge the length of block of bits to prevent a carry to the next block of bits. Note, that Shift-Add algorithm can also be considered as an implementation of dynamic programming.

In [BYN96b, BYN96a] they show a simulation of reduced *NFA* for the approximate string matching using the Levenshtein distance by ε-diagonals in time $\mathcal{O}(\lceil \frac{km}{w} \rceil n)$. They use one bit-vector with $(m - k)$ blocks of size $(k + 1)$. Each ε-diagonal is represented by one block.

In [BYN96a] they improve the approximate string matching using the Levenshtein distance in such a way, that they search for any of the first $(k + 1)$ symbols of the pattern. If they find any of them, they start *NFA* simulation and if the simulation then reaches the initial situation (i.e., only the states located in ε-diagonal leading from the initial state are active), they again start searching for any of the first $(k + 1)$ symbols of the pattern, which is faster than the simulation.

Shift-Or algorithm can also be used for the exact string matching with don't care symbols, classes of symbols, and complements as shown in [BYG92]. In [WM92] they extend this method to unlimited wild cards and apply the above features for the approximate string matching. They also use bit parallelism for regular expressions, for the weighted approximate string matching, for set of patterns, and for the situations, when errors are not allowed in some parts of pattern (another mask vector is used).

All the previous cases have good time complexity if the *NFA* simulation fits in one computer word (each vector in one computer word). If *NFA* is larger, it is necessary to partition *NFA* or pattern as described in [BYN96b, BYN96a, BYN97, BYN99, NBY98, WM92].

Shift-Or algorithm can also be used for multiple pattern matching [BYG92, BYN97] or for the distributed pattern matching [HIMM99].

6.8 Time and Space Complexity

The time and space analysis of the Shift-Or algorithm is as follows [WM92]. Denote the computer word size by w. The preprocessing requires $\mathcal{O}(m|\Sigma|)$ time plus $\mathcal{O}(k\lceil\frac{m}{w}\rceil)$ to initialize the k vectors. The running time is $\mathcal{O}(nk\lceil\frac{m}{w}\rceil)$. The space complexity is $\mathcal{O}(|\Sigma|\lceil\frac{m}{w}\rceil)$ for mask matrix D plus $\mathcal{O}(k\lceil\frac{m}{w}\rceil)$ for the k vectors.

6.9 General Rules

The bit parallelism is very similar to the naive method implemented by the bit vectors but in this method we do not need the transition table. The set Q of states of *NFA* is divided to subsets $Q_1, Q_2, \ldots, Q_l$ such that:

(1) $Q = \bigcup_{i=1}^{l} Q_i$,

(2) $Q_i \cap Q_j = \emptyset$, $\forall i, j$, $1 \leq i \leq l$, $1 \leq j \leq l$, $i \neq j$.

Each such subset Q_i is implemented by the bit vector, in which each bit represents one[4] state $q \in Q_i$—0 represents that state q is active. The transition table in this technique is represented by the formula for the computation of the vectors, in which groups of transitions are computed at once (in parallel—therefore this technique is called the bit parallelism).

To make bit parallelism as effective as possible, we have to minimize the number of bitwise operations for computing bit vectors (i.e., to maximize exploiting parallelism within computer word).

[4]Only in Shift-Add simulation of *NFA* for the approximate string matching using the Hamming distance a block of bits represents binary number of level of the topmost active state in the corresponding depth of *NFA*.

Chapter 7

Conclusions and Future Work

7.1 Summary of Results

In this thesis we have presented nondeterministic finite automata (*NFA*s) for the exact and approximate pattern (string and sequence) matching and their variants (Hamming, Levenshtein, or generalized Levenshtein distance, weighted approximate pattern matching, reduced *NFA*s, ... etc.).

These *NFA*s cannot be directly used because of their nondeterminism. We can transform them to their equivalent *DFA*s. In this thesis we estimate numbers of states of such *DFA*s. Then we show other method for use of *NFA*s, which is simulation of run of *NFA*. We present uniform treatment of several approximate string matching algorithms [BYG92, BYN96b, Ukk85, WM92] by the way of simulation of *NFA*.

At the beginning of the thesis we present *NFA*s for pattern matching. *NFA* for the exact string matching is well known. *NFA* for approximate string matching using the Hamming distance and the initial version of *NFA* for approximate string matching using the Levenshtein distance (both $\overline{p}$ versions) were published in [Dar92, Mel95, Mel96b]. We have constructed *NFA*s for other problems of exact and approximate pattern matching. Some of them were published in [Hol96b, Hol97, MH97], other are completely new. We distinguish so called $\overline{p}$ versions and Σ versions of *NFA*s, both of them skip trivial occurrences (caused by appending mismatching symbols behind a previous occurrence). For all *NFA*s we have estimated numbers of states of their equivalent *DFA*s.

Then we present the basic simulation method, which is the base for the other simulation methods described in this thesis. This simulation can be used for general *NFA*s and it runs in time $\mathcal{O}(n|Q|\lceil\frac{|Q|}{w}\rceil)$ and space $\mathcal{O}(|\Sigma||Q|\lceil\frac{|Q|}{w}\rceil)$, where n is a length of input text, Q is the set of states of *NFA*, Σ is input alphabet, and w is number of bits used in computer word ($\mathcal{O}(\log_2 n)$). The goal of other simulation methods is to improve the complexities of the basic simulation method. They differ in representation of set of active states, which is the foundation stone of the basic simulation method.

The following simulation method presented is the dynamic programming, in which the set of active states is represented by a vector of integers. This algorithm was directly used for the approximate string matching using the Levenshtein distance in [Sel80, Ukk85], but their algorithm searches also for trivial occurrences. We present the dynamic programming as a simulation method. We show, how to use this simulation method for all other *NFA*s presented in this thesis.

The last simulation method presented in this thesis is bit parallelism, in which the set of active states is represented by bit vectors. Bit parallelism covers Shift-Or algorithm (for the exact string matching used in [BYG92] and for the approximate string matching using the Levenshtein distance used in [BYN96b]), Shift-And algorithm (for the approximate string matching using the Levenshtein distance used in [WM92]), and Shift-Add algorithm (for the approximate string matching using the Hamming distance used in [BYG92]). All of them were designed to search also for trivial occurrences. We show the simulation of all *NFA*s presented in this thesis by bit parallelism.

The simulations are shown not only for the approximate string matching using the Hamming and Levenshtein distances, but also for the approximate string matching using the generalized Levenshtein distance, for the exact and approximate sequence matching, for the reduced *NFA*s for the approximate string and sequence matching, for the pattern verification, and for the weighted approximate pattern matching.

If we have a pattern, a number of errors, and size of an input text, we can decide, what method (transformation to *DFA* or some of the simulation methods) is the best choice for the searching. The longer input text we have, the longer preprocessing time (preprocessing of simulation or *DFA* construction) is acceptable.

If we compare bit parallelism and the dynamic programming, we have to tell that the dynamic programming has no preprocessing (except for the reduced *NFA*s). Therefore it is more suitable for matching, where the pattern changes during searching process as used in [CIY98].

Dynamic programming is suitable for *NFA* that can be decomposed to set of states, each of them having at most one active state in each step of simulation. On the other hand, bit parallelism is suitable for such *NFA* that can be sorted into bit vectors in such a way, that the transitions are then represented by only several bit operations over these vectors.

The main contribution of this thesis is the presentation and analysis of *NFA* simulators.

7.2 Suggestions for Further Research

In this thesis the simulation methods (bit parallelism and dynamic programming) require human analysis of structure of simulated *NFA*. The simulation is then based on that structure. The future research could make this analysis automated, such that some program decides optimal representation of active states for simulation. Future research could also find other simulation methods.

Bibliography

[Abr87] K. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987.

[AC75] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[BC92] H. Bunke and J. Csirik. Parametric string matching and its application to pattern recognition. Technical Report IAM-92-007, University of Bern, 1992.

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

[BYG92] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[BYN96a] R. A. Baeza-Yates and G. Navarro. A fast heuristic for approximate string matching. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proceedings of the 3rd South American Workshop on String Processing*, pages 47–63, Recife, Brazil, 1996. Carleton University Press.

[BYN96b] R. A. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In D. S. Hirschberg and E. W. Myers, editors, *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, number 1075 in Lecture Notes in Computer Science, pages 1–23, Laguna Beach, CA, 1996. Springer-Verlag, Berlin.

[BYN97] R. A. Baeza-Yates and G. Navarro. Multiple approximate string matching. In F. K. H. A. Dehne, A. Rau-Chaplin, J.-R. Sack, and R. Tamassia, editors, *Proceedings of the 5th Workshop on Algorithms and Data Structures*, number 1272 in Lecture Notes in Computer Science, pages 174–184, Halifax, Nova Scotia, Canada, 1997. Springer-Verlag, Berlin.

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

[CIY98] M. Crochemore, C. S. Iliopoulos, and H. Yu. Algorithms for computing evolutionary chains in molecular and musical sequences. In C. S. Iliopoulos, editor, *Proceedings of the 9th Australasian Workshop On Combinatorial Algorithms*, pages 172–184, Perth, WA, Australia, 1998.

[Cro97] M. Crochemore. Off-line serial exact string searching. In A. Apostolico and Z. Galil, editors, *Pattern matching algorithms*, chapter 1, pages 1–53. Oxford University Press, 1997.

[CW79] B. Commentz-Walter. A string matching algorithm fast on the average. In H. A. Maurer, editor, *Proceedings of the 6th International Colloquium on Automata, Languages and Programming*, number 71 in Lecture Notes in Computer Science, pages 118–132, Graz, Austria, 1979. Springer-Verlag, Berlin.

[Dar92] T. Darabont. Approximate string matching with k mismatches. Master's thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic, 1992.

[Döm64] B. Dömölki. An algorithm for syntactical analysis. *Computational Linguistics*, (3):29–46, 1964.

[FL96] A. L. N. Fred and J. M. N. Leitão. A minimum code length technique for clustering of syntactic patterns. In *Proceedings of the 13th International Conference on Pattern Recognition*, volume II., Vienna, Austria, 1996. IEEE Computer Society Press.

[Fre74] H. Freeman. Computer processing of linear-drawing images. *ACM Comp. Surv.*, 6(1):57–97, 1974.

[GG88] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *J. Complexity*, 4(1):33–72, 1988.

[GL89] R. Grossi and F. Luccio. Simple and efficient string matching with k mismatches. *Inf. Process. Lett.*, 33(3):113–120, 1989.

[GP] G. Golinelli and A. Poggi. Image retrieval by content from databases of fashion sketches.

[GP89] Z. Galil and K. Park. An improved algorithm for approximate string matching. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, number 372 in Lecture Notes in Computer Science, pages 394–404, Stresa, Italy, 1989. Springer-Verlag, Berlin.

[HIMM99] J. Holub, C. S. Iliopoulos, B. Melichar, and L. Mouchard. Distributed string matching using finite automata. In R. Raman and J. Simpson, editors, *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms*, pages 114–128, Perth, WA, Australia, 1999.

[Hol96a] J. Holub. Approximate string matching in text. Master's thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic, 1996.

[Hol96b] J. Holub. Reduced nondeterministic finite automata for approximate string matching. In J. Holub, editor, *Proceedings of the Prague Stringologic Club Workshop '96*, pages 19–27, Czech Technical University, Prague, Czech Republic, 1996. Collaborative Report DC–96–10.

[Hol97] J. Holub. Simulation of NFA in approximate string and sequence matching. In J. Holub, editor, *Proceedings of the Prague Stringology Club Workshop '97*, pages 39–46, Czech Technical University, Prague, Czech Republic, 1997. Collaborative Report DC–97–03.

[Hol98] J. Holub. Dynamic programming for reduced NFAs for approximate string and sequence matching. In J. Holub and M. Šimánek, editors, *Proceedings of the Prague Stringology Club Workshop '98*, pages 73–82, Czech Technical University, Prague, Czech Republic, 1998. Collaborative Report DC–98–06.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to automata, languages and computations.* Addison-Wesley, Reading, MA, 1979.

[Huf74] D. A. Huffman. The synthesis of sequential switching circuits. *J. Franklin Institute*, 257:3–4, 161–190, 275–303, 1974.

[JTU96] P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. *Softw. Pract. Exp.*, 26(12):1439–1458, 1996.

[Kle56] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, pages 3–42, 1956.

[KMP77] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.

[Koz97] D. C. Kozen. *Automata and Computability.* Springer-Verlag, Berlin, 1997.

[Kur96] S. Kurtz. Approximate string searching under weighted edit distance. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proceedings of the 3rd South American Workshop on String Processing*, pages 156–170, Recife, Brazil, 1996. Carleton University Press.

[LV88] G. M. Landau and U. Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37(1):63–78, 1988.

[LW75] R. Lowrance and R. A. Wagner. An extension of the string-to-string correction problem. *J. Assoc. Comput. Mach.*, 22(2):177–183, 1975.

[MC97] C. Moore and J. P. Crutchfield. Quantum automata and quantum grammars. 1997. `http://xxx.lanl.gov/abs/quant-ph/9707031`.

[Mea55] G. H. Mealy. A method for synthetizing sequential circuits. *Bell System Technical J.*, 34(5):1045–1079, 1955.

[Mel95] B. Melichar. Approximate string matching by finite automata. In V. Hlaváč and R. Šára, editors, *Computer Analysis of Images and Patterns*, number 970 in Lecture Notes in Computer Science, pages 342–349. Springer-Verlag, Berlin, 1995.

[Mel96a] B. Melichar. Space complexity of linear time approximate string matching. In J. Holub, editor, *Proceedings of the Prague Stringologic Club Workshop '96*, pages 28–36, Czech Technical University, Prague, Czech Republic, 1996. Collaborative Report DC–96–10.

[Mel96b] B. Melichar. String matching with k differences by finite automata. In *Proceedings of the 13th International Conference on Pattern Recognition*, volume II., pages 256–260, Vienna, Austria, 1996. IEEE Computer Society Press.

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

[Moo56] E. F. Moore. Gedanken experiments on sequential machines. *Automata Studies*, pages 129–153, 1956.

[MP43] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophysics*, 5:115–133, 1943.

[MP80] W. J. Masek and M. S. Paterson. A faster algorithm for computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

[Nav97] G. Navarro. A partial deterministic automaton for approximate string matching. In R. Baeza-Yates, editor, *Proceedings of the 4th South American Workshop on String Processing*, pages 95–111, Valparaiso, Chile, 1997. Carleton University Press.

[NBY98] G. Navarro and R. Baeza-Yates. Improving an algorithm for approximate pattern matching. Technical Report TR/DCC-98-5, Dept. of Computer Science, University of Chile, 1998.

[RS59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3:114–125, 1959.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

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

[Shy76] R. K. Shyamasundar. A simple string matching algorithm. Technical report, Tata Institute of Fundamental Research, 1976.

[Sol64] R. J. Solomonoff. A formal theory of inductive inference (part I. and II.). *Inf. Control*, 7:1–22, 224–254, 1964.

[TKY+96] A. Takashu, N. Katayama, M. Yamaoka, O. Iwaki, K. Oyama, and J. Adachi. Approximate matching for OCR-processed bibliographic data. In *Proceedings of the 13th International Conference on Pattern Recognition*, volume III., Vienna, Austria, 1996. IEEE Computer Society Press.

[Ukk85] E. Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1–3):132–137, 1985.

[Wat95] B. W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. Ph. D. Thesis, Eindhoven University of Technology, 1995.

[WF74] R. A. Wagner and M. Fischer. The string-to-string correction problem. *J. Assoc. Comput. Mach.*, 21(1):168–173, 1974.

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