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

TEXT SEARCHING ALGORITHMS

CASE STUDIES

Bořivoj Melichar, Miroslav Balík, Jan Holub, Jan Lahoda,
Michal Voráček and Jan Žďárek

December 2005
(version December 2, 2005)

Contents

1 Definitions

This chapter provides the essential definitions needed for understanding this text. These definitions were selected from *Athens' Tutorial* and they were incorporated here for reader's more comfortable reading.

1.1 Basics

Definition 1.1 (Alphabet)
An *alphabet* A is a finite non-empty set of *symbols*.

Definition 1.2 (Complement of symbol)
A *complement* of symbol a over A, where $a \in A$, is a set $A \setminus \{a\}$ and is denoted $\overline{a}$.

Definition 1.3 (String)
A *string* over A is any sequence of symbols from A.

Definition 1.4 (Set of all strings)
The *set of all strings* over A is denoted A^*.

Definition 1.5 (Set of all non-empty strings)
The *set of all non-empty strings* over A is denoted A^+.

Definition 1.6 (Length of string)
The *length of string* x is the number of symbols in string $x \in A^*$ and is denoted $|x|$.

Definition 1.7 (Empty string)
An *empty string* is a string of length 0 and is denoted ε.

Remark 1.8
It holds $A^* = A^+ \cup \{\varepsilon\}$.

Notation 1.9
Exponents for string with repetitions will be used: $a^0 = \varepsilon$, $a^1 = a$, $a^2 = aa$, $a^3 = aaa$, $\ldots$, for $a \in A$ and $x^0 = \varepsilon, x^1 = x, x^2 = xx, x^3 = xxx, \ldots$, for $x \in A^*$.

Definition 1.10 (Concatenation)
The operation *concatenation* is defined over the set of strings A^* as follows: if x and y are strings over the alphabet A, then by appending the string y to the string x we obtain the string xy.

1.2 Finite automata

Definition 1.11 (Deterministic finite automaton)
A deterministic finite automaton (*DFA*) is a quintuple $M = (Q, A, \delta, q_0, F)$, where
Q is a finite set of states,
A is a finite input alphabet,
δ is a mapping from $Q \times A$ to Q, $(Q \times A \mapsto Q)$
$q_0 \in Q$ is an initial state,
$F \subset Q$ is the set of final states.

Definition 1.12 (Configuration of FA)
Let $M = (Q, A, \delta, q_0, F)$ be a finite automaton. A pair $(q, w) \in Q \times A^*$ is *a configuration of the finite automaton M.* A configuration (q_0, w) is called *an initial configuration*, a configuration (q, ε), where $q \in F$, is called *a final (accepting) configuration* of the finite automaton M.

Definition 1.13 (Transition in DFA)
Let $M = (Q, A, \delta, q_0, F)$ be a deterministic finite automaton. A relation $\vdash_M \in (Q \times A^*) \times (Q \times A^*)$ is called *a transition* in the automaton M. If $\delta(q, a) = p$, then $(q, aw) \vdash_M (p, w)$ for each $w \in A^*$. The k-power of the relation $\vdash_M$ will be denoted by $\vdash_M^k$. The symbols $\vdash_M^+$ and $\vdash_M^*$ denote a transitive and a transitive reflexive closure of the relation $\vdash_M$, respectively.

Definition 1.14 (Language accepted by DFA)
We will say that an input string $w \in A^*$ *is accepted* by a finite deterministic automaton
$M = (Q, A, \delta, q_0, F)$ if $(q_0, w) \vdash_M^* (q, \varepsilon)$ for some $q \in F$.
The language $L(M) = \{w : w \in T^*, (q_0, w) \vdash^* (q, \varepsilon), q \in F\}$ is *the language accepted* by a finite automaton M. A string $w \in L(M)$ if it consists only of symbols from the input alphabet and there is a sequence of transitions such that it leads from the initial configuration (q_0, w) to the final configuration (q, ε), $q \in F$.

Definition 1.15 (Complete DFA)
A finite deterministic automaton $M = (Q, A, \delta, q_0, F)$ is said to be *complete* if the mapping $\delta(q, a)$ is defined for each pair of states $q \in Q$ and input symbols $a \in A$.

Definition 1.16 (Nondeterministic finite automaton)
A nondeterministic finite automaton (NFA) is a quintuple $M = (Q, A, \delta, q_0, F)$, where
Q is a finite set of states,
A is a finite input alphabet,
δ is a mapping from $Q \times A$ into the set of subsets of Q,
$q_0 \in Q$ is an initial state,
$F \subset Q$ is the set of final states.

Definition 1.17 (Transition in NFA)
Let $M = (Q, A, \delta, q_0, F)$ be a nondeterministic finite automaton. A relation
$\vdash_M \subset (Q \times A^*) \times (Q \times A^*)$ will be called *a transition* in the automaton M if $p \in \delta(q_,a)$ then $(q, aw) \vdash_M (p, w)$, for each $w \in A^*$.

Definition 1.18 (Language accepted by NFA)
A string $w \in A^*$ is said to be *accepted by a nondeterministic finite automaton*
$M = (Q, A, \delta, q_0, F)$, if there exists a sequence of transitions $(q_0, w) \vdash^* (q, \varepsilon)$ for some $q \in F$.
The language $L(M) = \{w : w \in A^*, (q_0, w) \vdash^* (q, \varepsilon)$ for some $q \in F\}$ is then *the language accepted by a nondeterministic finite automaton M.*

Definition 1.19 (NFA with ε-transitions)
A nondeterministic finite automaton with ε-transitions is a quintuple $M = (Q, A, \delta, q_0, F)$, where
Q is a finite set of states,
A is a finite input alphabet,
δ is a mapping from $Q \times (A \cup \{\varepsilon\})$ into the set of subsets of Q,
$q_0 \in Q$ is an initial state,
$F \subset Q$ is the set of final states.

Definition 1.20 (Transition in NFA with ε-transitions)
Let $M = (Q, A, \delta, q_0, F)$ be a nondeterministic finite automaton with ε-transitions. A relation $\vdash_M \subset (Q \times A^*) \times (Q \times A^*)$ will be called *a transition* in the automaton M if $p \in \delta(q,a)$, $a \in A \cup \{\varepsilon\}$, then $(q, aw) \vdash_M (p, w)$, for each $w \in A^*$.

Definition 1.21 (ε-*CLOSURE*)
Function ε-*CLOSURE* for finite automaton $M = (Q, A, \delta, q_0, F)$ is defined as:

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

Definition 1.22 (NFA with set of initial states)
A nondeterministic finite automaton M with set of initial states I is a quintuple
$M = (Q, A, \delta, I, F)$, where:
Q is a finite set of states,
A is a finite input alphabet,
δ is a mapping from $Q \times A$ into the set of subsets of Q,
$I \subset Q$ is the non-empty set of initial states,
$F \subset Q$ is the set of final states.

Definition 1.23 (Accessible state)
Let $M = (Q, A, \delta, q_0, F)$ be a finite automaton. A state $q \in Q$ is called *accessible* if there exists a string $w \in A^*$ such that there exists a sequence of transitions from the initial state q_0 into the state q:

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

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

Definition 1.24 (Useful state)
Let $M = (Q, A, \delta, q_0, F)$ be a finite automaton. A state $q \in Q$ is called *useful* if there exists a string $w \in A^*$ such that there exists a sequence of transitions from the state q into some final state:
$(q, w) \vdash_M (p, \varepsilon)$, $p \in F$.

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

Definition 1.25 (Finite automaton)
Finite automaton (*FA*) is DFA or NFA.

Definition 1.26 (Equivalence of finite automata)
Finite automata M_1 and M_2 are said to be *equivalent* if they accept the same language, it means $L(M_1) = L(M_2)$.

Definition 1.27 (Sets of states)
Let $M = (Q, A, \delta, q_0, F)$ be a finite automaton. Let us define for arbitrary $a \in A$ the set $Q(a) \subset Q$ as follows:
$Q(a) = \{q : q \in \delta(p, a), a \in A, p, q \in Q\}$.

Definition 1.28 (Homogenous automaton)
Let $M = (Q, A, \delta, q_0, F)$ be a finite automaton and $Q(a)$ be sets of states for all symbols $a \in T$. If for all pairs of symbols $a, b \in A$, $a \neq b$, it holds $Q(a) \cap Q(b) = \emptyset$, then the automaton M is called homogenous. The collection of sets $\{Q(a) : a \in A\}$ is for the homogenous finite automaton the decomposition on classes having one of these two forms:

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

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

1.3 Text searching

Definition 1.29 (Replace)
Edit operation *replace* is an operation which converts string wav to string wbv, where $w, v \in A^*$, $a, b \in A$ (one symbol is replaced by another).

Definition 1.30 (Insert)
Edit operation *insert* is an operation which converts string wv to string wav, where $w, v \in A^*$, $a \in A$ (a symbol is inserted into a string).

Definition 1.31 (Delete)
Edit operation *delete* is an operation which converts string wav to string wv, where $w, v \in A^*$, $a \in A$ (a symbol is removed from a string).

Definition 1.32 (Transpose)
Edit operation *transpose* is an operation which converts string $wabv$ to string $wbav$, where $w, v \in A^*$, $a, b \in A$ (two adjacent symbols are exchanged).

Definition 1.33 (Distances of strings)
Three variants of distances between two strings u and v, where $u, v \in A^*$, are defined as minimal number of editing operations:

1. *replace* (Hamming distance, R-distance),
2. *delete, insert* and *replace* (Levenshtein distance, DIR-distance),
3. *delete, insert, replace* and *transpose* (generalized Levenshtein distance, $DIR\ T$-distance),

needed to convert string u into string v.

Definition 1.34 ("Don't care" symbol)
"Don't care" symbol is a special universal symbol $\circ$ that matches any other symbol including itself.

Definition 1.35 (Set of all prefixes)
The set $Pref(x)$, $x \in A^*$, is a set of all prefixes of string x:
$Pref(x) = \{y : x = yu,\ x, y, u \in A^*\}$.

Definition 1.36 (Set of all suffixes)
The set $Suff(x)$, $x \in A^*$, is a set of all suffixes of string x:
$Suff(x) = \{y : x = uy,\ x, y, u \in A^*\}$.

Definition 1.37 (Set of all factors)
The set $Fact(x),\ x \in A^*$, is a set of all factors of the string x:
$Fact(x) = \{y : x = uyv,\ x, y, u, v \in A^*\}$.

Definition 1.38 (Set of all subsequences)
The set $Sub(x),\ x \in A^*$, is a set of all subsequences of the string x:
$$\begin{aligned} Sub(x) = \{\, & a_1a_2 \dots a_m : x = y_0a_1y_1a_2 \dots a_my_m, \\ & y_i \in A^*, i = 0, 1, 2, \dots, m, a_j \in A, \\ & j = 1, 2, \dots, m,\ m \geq 0\} \end{aligned}$$

Definition 1.39 (Set of approximate prefixes)
The set of approximate prefixes $APref$ of the string x is a set:
$APref(x) = \{u : v \in Pref(x),\ D(u, v) \leq k\}$.

Definition 1.40 (Set of approximate suffixes)
The set of approximate suffixes $ASuff$ of the string x is a set:
$ASuff(x) = \{u : v \in Suff(x),\ D(u, v) \leq k\}$.

Definition 1.41 (Set of approximate factors)
The set of approximate factors $AFact$ of the string x is a set:
$AFact(x) = \{u : v \in Fact(x),\ D(u, v) \leq k\}$.

Definition 1.42 (Set of approximate subsequences)
The set of approximate subsequences $ASub$ of the string x is a set:
$ASub(x) = \{u : v \in Sub(x),\ D(u, v) \leq k\}$.

Definition 1.43 (Prefix automaton)
The *prefix automaton* for string u is a finite automaton accepting the language $Pref(u)$.

Definition 1.44 (Suffix automaton)
The *suffix automaton* for string u is a finite automaton accepting the language $Suff(u)$.

Definition 1.45 (Factor automaton)
The *factor automaton* for string u is a finite automaton accepting the language $Fact(u)$.

Definition 1.46 (Subsequence automaton)
The *subsequence automaton* for string u is a finite automaton accepting the language $Sub(u)$.

Definition 1.47 (Approximate prefix automaton)
The *approximate prefix automaton* for string u is a finite automaton accepting the language $APref(u)$.

Definition 1.48 (Approximate suffix automaton)
The *approximate suffix automaton* for string u is a finite automaton accepting the language $ASuff(u)$.

Definition 1.49 (Approximate factor automaton)
The *approximate factor automaton* for string u is a finite automaton accepting the language $AFact(u)$.

Definition 1.50 (Approximate subsequence automaton)
The *approximate subsequence automaton* for string u is a finite automaton accepting the language $ASub(u)$.

Definition 1.51 (Basic pattern matching problems)
Given a text string $T = t_1t_2 \cdots t_n$ and a pattern $P = p_1p_2 \cdots p_m$. Then we may define:

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

Definition 1.52 (Matching a sequence of patterns)
Given a text string $T = t_1 t_2 \cdots t_n$ and a sequence of patterns (strings or sequences) $P_1, P_2, \ldots,$ P_s. Matching of sequence of patterns $P_1, P_2, \ldots, P_s$ is a verification whether an occurrence of pattern P_i in text T is followed by an occurrence of P_{i+1}, $1 \leq i < s$.

2 Implementation of Suffix Automaton Miroslav Balík

2.1 Motivation

This Section shows an effective implementation of *suffix automaton.* Suffix automaton for text T is a minimal automaton that accepts all suffixes of the text T and represents a complete index of this input text T. While all usual implementations of suffix automaton need about 30 times larger storage space than the size of the input text, here we show an implementation that decreases this requirement down to four times the size of the input text. This is a better result than any known implementation of *suffix trees*, structures used for the same kind of tasks.

The method uses data compression of suffix automaton elements, i.e. states, transitions and labels. The construction time of this implementation is linear with respect to the size of the input text. This implementation does not increase the time to search for a pattern, which is proportional to the length of the pattern.

2.2 Introduction

String matching is one of the most frequently used tasks in text processing. With the increased volume of processed data, which usually consists of unstructured texts, the importance of data qualification technologies is increasing. This is the reason why indexing structures are constructed for static texts that support pattern matching in a linear time with respect to the length of the pattern.

Although some indexing structures have a linear size with respect to the length of the text, this size is high enough to prevent practical implementation and usage. This size depends on implementation details, on the type of the text and on the type of the automaton used. For the suffix tree the size is rarely smaller than $10n$ bytes, where n is the length of the text. Other structures are suffix automaton called Directed Acyclic Word Graph (*DAWG*) automaton (size about $30n$ bytes) and its compact version *CDAWG* (size about $10n$ bytes). Stefan Kurtz in [1] shows a number of known implementations of these automata together with experimental evaluations and a number of references.

Other types of indexing structures are usually smaller, *suffix arrays* [2] (size $5n$ bytes), *level compressed tries* [3] (size about $11n$ bytes), *suffix cactuses* - a combination of suffix trees and suffix arrays [4] (size $9n$ bytes), and *suffix binary search trees*[5] (size about $10n$ bytes).

An automaton is usually stored as a graph, with vertices represent states of the automaton and edges represent transitions. A state is then represented by an index into a transition table or as a memory position referenced by edges. Edges are stored as a part of the vertex they start from. Thus it is possible to locate an edge with a specific label in a constant time with respect to the number of vertices and edges. Each edge contains information about the vertex it leads to, and its label, which is one symbol in the case of suffix automaton (*DAWG*), or a sequence of symbols in the case of *CDAWG* and suffix trees. Every sequence used as a label is a substring of a text, so it can be represented by a starting and ending positions of this substring.

An implementation presented in this paper uses a compression of elements of the graph representing the automaton to decrease space requirements. The "compression" is not a compression of the whole data structure, which would mean performing decompression to be able to work with it, but it is a compression of individual elements, so it is necessary to decompress only those elements that are necessary during a specific search. This method is applicable for all homogenous automata[1] and it can be generalized to all automata accepting a finite set of strings and to all structures, which can be drawn as an acyclic graph.

The whole graph is a sequence of bits in a memory that can be referenced by pointers. A vertex is a position in the bit stream where a sequence of edges originating from the vertex begins. These edges are pointers into a bit stream, they point to places where the corresponding terminal vertices are located. Vertices are stored in a topological ordering[2], which ensures that a search for a pattern is a one-way pass through the implementation of the graph structure.

Each vertex contains an information about labels of all edges leading to it and the number of edges that start from it. Since it is possible to construct a statistical distribution of all symbols in the text, we can store edge labels using a Huffman code [6]. We can use it also to encode the number of edges starting from a vertex and, because the most frequent case is when only one edge starts from a vertex, it is dealt with as a special case.

The approach presented here creates a suffix automaton structure in three phases. The first phase is the construction of the usual suffix automaton (Page 10), the second phase is the topological ordering (or re-ordering) of vertices (Subsection Topological Ordering, page 12), which ensures that no edge has a negative "length", where length is measured as a difference of vertex numbers. The final phase is the encoding and storing the resulting structure. Encoding of each element is described on page 13.

We have used a set of 31 files in the experiments. 17 files are taken from the Calgary Corpus and 14 files from the Canterbury Corpus [7]. The Canterbury Corpus consists of 18 files, but the file pic is identical to the file ptt5 of the Calgary Corpus. Both Corpora are widely used to compare lossless data compression programs. We use these files to test performance for matching our results with Kurz's results [1].

2.3 Suffix automaton properties

SA(T) is a minimal automaton that accepts all suffixes of a text T.

The major advantages of suffix automaton are:

- it has a linear size limited by the number of states, which is less than $2|n| - 2$; the number of transitions is less than $3|n| - 4$, where $n > 1$ is the length of the text [8],
- it can be constructed in $\mathcal{O}(n)$ time [8],
- it allows to check whether a pattern occurs in a text in $\mathcal{O}(m)$ time, where m is the length of the pattern. Pattern matching algorithm is shown in Fig. 2.1.

2.4 Construction of suffix automaton

There are many ways of constructing suffix automaton from text, more details can be found for example in [8]. The method used here is the on-line construction algorithm. An example of suffix automaton constructed using this algorithm for an input text $T = acagac$ is shown in Fig. 2.8.

[1]Homogenous automata have all transitions to a specific state labeled with the same symbol.

[2]A topological ordering of a graph is a numbering of vertices that ensures that each edge starts from a vertex with a lower number and ends at a vertex with a higher number.

Input: $SA(T)$, pattern $P = p_1p_2 \dots p_m$.
Output: YES, if P occurs in T, NO otherwise.

```
1. State $q := q_0$; $i := 1$;
2. ***while***$((i < m + 1)$ ***and*** $(q \neq NULL))$
3.     ***do*** $q := Successor(q, P[i])$; $i := i + 1$; ***end***
4. ***if***$(i < m + 1)$ ***then*** **NO** // Pattern does not occur in Text
   ***else*** **YES** // Pattern occurs in Text
```

Figure 2.1: Matching algorithm

When drawing the state graph corresponding to a finite automaton, we adopt the following conventions:

- All states are drawn as circles (vertices).
- Transition are drawn as labeled (with alphabet symbol $x \in A$) directed edges between states.
- Start states have an in-transition with no source.
- Final states are drawn as two concentric circles.

Index i points to the processed symbol of the pattern, function $Successor(q, P[i])$ evaluates the successor of state q using the edge labeled by symbol $P[i]$. If there is no edge from state q labeled by this symbol, the function holds the $NULL$ value.

The whole algorithm consists of the main loop. The number of iterations is m in the worst case. In each iteration the appropriate edge is searched. This search depends only on the size of the input alphabet, it is independent on the number of edges in the whole graph. The resulting time complexity of the algorithm is $\mathcal{O}(m)$.

During the construction phase a statistical distribution of symbols in the text is created. A statistical distribution of the number of edges at respective vertices is also created.

Table 2.1 shows general properties of the suffix automaton for testing files. The first column shows the name of the testing file, the second column shows its source, CL – Calgary Corpus, CN – Canterbury Corpus. The third column shows the size of the text, the next one shows the size of the input alphabet. The ratio between the size of the text and the size of the corresponding suffix automaton is shown in sixth column, and the last column describes the average number of transitions that start from one state.

The size of an automaton is the number of its states. It is obvious that the number of states together with the number of transitions directly influence the size of the implementation of the automaton.

The average number of transitions that start from one state can be calculated as the ratio of the number of states and the number of transitions. If this ratio is low, we can transfer transition labels to the states the transitions lead to. This will also influence the behaviour of the matching algorithm.

File	Source	Length	$\lvert A \rvert$	States/symbol	Trans./state
book1	CL	768771	81	1.51	1.47
book2	CL	610856	96	1.54	1.37
paper1	CL	53161	95	1.55	1.37
paper2	CL	82199	91	1.52	1.41
paper3	CL	46526	84	1.51	1.43
paper4	CL	13286	80	1.52	1.44
paper5	CL	11954	91	1.52	1.43
paper6	CL	38105	93	1.57	1.37
alice29	CN	152089	74	1.54	1.41
lcet10	CN	426754	84	1.54	1.37
plrabn12	CN	481861	81	1.50	1.46
bible	CN	4047392	63	1.56	1.31
world192	CN	2473400	94	1.53	1.23
bib	CL	111261	81	1.52	1.30
news	CL	377109	98	1.52	1.38
progc	CL	39611	92	1.55	1.36
progl	CL	71646	87	1.64	1.23
progp	CL	49379	89	1.66	1.23
trans	CL	93695	99	1.71	1.18
fields	CN	11150	90	1.61	1.27
cp	CN	24603	86	1.53	1.32
grammar	CN	3721	76	1.60	1.30
xargs	CN	4227	74	1.54	1.40
asyoulik	CN	125179	68	1.50	1.45
geo	CL	102400	256	1.30	1.57
obj1	CL	21504	256	1.35	1.45
obj2	CL	246814	256	1.46	1.30
ptt5	CN	513216	159	1.53	1.23
kennedy	CN	1029744	256	1.05	1.52
sum	CN	38240	255	1.44	1.37
ecoli	CN	4638690	4	1.64	1.54

Table 2.1: General properties of the suffix automata

2.5 Topological ordering

The suffix automaton structure is a directed acyclic graph. This means that its vertices can be ordered according to their interconnection by edges. Such an implementation that keeps all the information about edges starting from a vertex only in the vertex concerned while storing the vertices in a given order guarantees that every pattern matching event will result in a single one-way pass through this structure.

The problem of such topological ordering can be solved in linear time. At first, for each vertex its input degree (the number of edges ending at the vertex) is computed, next a list of vertices having an input degree equal to zero (the list of roots) is constructed. At first, this list will contain only the initial vertex. One vertex is chosen from the list. This vertex gets the next number in the ordering and for all vertices accessible by an edge starting at this vertex their input degree is decreased by one. Then vertices that have a zero input degree are inserted into the list. This goes on until the list is empty. The order of the vertices, which determines the quality of the final implementation, obtained this way depends on the strategy of choosing a vertex from the list. Several strategies were tested and the best results were obtained using the LIFO (last in - first out) strategy, because using this strategy for vertices with only one outgoing edge gets its successor next number (if this successor has been inserted into the roots list). Cases where the vertices have only one edge, and this edge points to the successor in topological ordering, are dealt as a special case (see Subection on Encoding on page 13).

2.6 Encoding

The graph of suffix automaton is encoded element by element (elements are described later in this Subsection). It starts with the last vertex according to the topological order (as described above) and progresses in the reverse order, ending with the first vertex of the order. This ensures that a vertex position can be defined by the first bit of its representation and that all edges starting at the current vertex can be stored because all ending vertices have already been processed and their addresses are known.

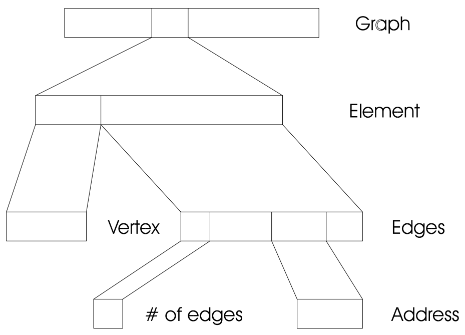


Figure 2.2: Implementation - Data Structures

The highest building block is a *graph.* It is further divided into single *elements.* Each *element* consists of two parts: a *vertex* and an *edge*. A *vertex* carries information on a label of all edges ending at it. A Huffman code is used for coding symbols of the alphabet. An *edge* is further split into a *header* and an *address order*. A *header* carries information on the *number of addresses* - edges belonging to a respective vertex. A distribution of edge counts for all vertices can be obtained during the construction of suffix automaton. This makes it possible to use a Huffman code for header encoding, but Fibonacci encoding([9]) is sufficient as well, though one must expect a substantial amount of small numbers. An *address* is the address of the first bit of the element being pointed to by an appropriate edge. It is further split into two parts, one describing the length of the other part, which is a binary encoded address.

2.6.1 Symbol encoding

A code of an *element* (vertex and corresponding edges) starts with a code of the symbol for which it is possible to enter the vertex. The best code is the Huffman code. The Huffman code for DNA sequence is the code of constant length, two bits. This is the most space efficient code for storing four different values with the same probability, stored without context.

Fig. 2.3 shows an algorithm of Huffman tree construction. This algorithm uses Huffman nodes. Each Huffman node consists of symbol labeling, number of occurrences, left and right

Input: Symbols $x_i \in A, i \in \{1, 2, \dots k\}$ and their number of occurrences n_i in text.
Output: Huffman tree.

```
1.  Foreach  $x \in A$ do
2.      begin
3.        $newl$ :=($x_i$, $n_i$, $NULL$, $NULL$);
4.        $Insert(L, newl)$; //Put this node into sorted list $L$
5.      end
6.  $j := k$;
7.  while ($j > 2$) do
8.      begin
9.        $l_1 := Pop(L)$, $l_2 := Pop(L)$; //Pop first two records from the list
10.       $newl$ :=(—,$n_{l_1} + n_{l_2}$, $l_1$, $l_2$); //Create new node and put it in $L$
11.       $Insert(L, newl)$;
12.     end
```

Figure 2.3: Huffman tree construction

sons. Leaves are initialized with symbols, their number of occurrences and two NULL pointers. New nodes have both sons defined and their symbols undefined. The edge to the left son is labeled by symbol 0, the right edge by symbol 1. The algorithm uses sorted list structure. The order of the record in the list depends on the value of the number of occurrences. The record with the lower number of occurrences precedes records with higher number of occurrences. The function $Insert(L, newl)$ inserts new record $newl$ into the sorted list L at its appropriate place.

Code words are obtained from Huffman tree. The code word for a symbol x_i is string on the path from the root to the leaf initialized by symbol x_i. The Huffman tree is also used for bit stream decoding.

Table 2.2 shows experimental results. The second column shows the size of text. The third column shows the size of alphabet and the fourth column shows the resulting average length of Huffman code. This average size as is calculated using this formula:

$$as = \frac{\Sigma_{i=1}^{k} n_i * l_i}{\Sigma_{i=1}^{k} n_i},$$

where symbol n_i denotes the number of $i-th$ symbol in the text. Symbol l_i denotes the number of bits necessary to store one $i-th$ symbol. Index i varies between 1 and k, where k is the size of the input alphabet.

The last column on Table 2.2 shows the average length of Huffman code for storing the number of edges going out from the vertex. This field will be discussed in the subsection Encoding of Number of Edges on page 15.

It can be observed that we obtain better results for text in natural languages than for text with constant probability for each symbol. Text files contain a lot of redundancy. This is similar to the case of files with an alphabet containing fewer symbols than 256. A special case are texts storing DNA sequences. DNA sequences contains only four symbols (usually a, c, g, t), but eight bits are used for each symbol, because the textual form is used. Huffman encoding

File	Source	Length	$\lvert A \rvert$	Bits/Symbol	Bits/# of Edges
book1	CL	768771	81	4.56	1.48
book2	CL	610856	96	4.82	1.39
paper1	CL	53161	95	5.01	1.40
paper2	CL	82199	91	4.63	1.43
paper3	CL	46526	84	4.69	1.46
paper4	CL	13286	80	4.73	1.43
paper5	CL	11954	91	4.97	1.46
paper6	CL	38105	93	5.04	1.39
alice29	CN	152089	74	4.61	1.43
lcet10	CN	426754	84	4.70	1.38
plrabn12	CN	481861	81	4.57	1.47
bible	CN	4047392	63	4.39	1.32
world192	CN	2473400	94	5.04	1.24
bib	CL	111261	81	5.23	1.31
news	CL	377109	98	5.23	1.37
progc	CL	39611	92	5.23	1.37
progl	CL	71646	87	4.80	1.26
progp	CL	49379	89	4.89	1.25
trans	CL	93695	99	5.57	1.19
fields	CN	11150	90	5.04	1.29
cp	CN	24603	86	4.84	1.47
geo	CL	102400	256	5.67	1.34
obj1	CL	21504	256	5.97	1.42
obj2	CL	246814	256	6.29	1.28
ptt5	CN	513216	159	1.66	1.22
kennedy	CN	1029744	256	3.59	1.08
sum	CN	38240	255	5.37	1.31
ecoli	CN	4638690	4	2.00	1.57

Table 2.2: Encoding of symbols and number of edges from a vertex

produces a code two bits long[3], six bits are saved for each stored symbol.

2.6.2 Encoding of number of edges

The code of the *number of edges* is another item. This value can also be obtained prior to encoding. A typical example of a distribution of the number of edges for three input text files is shown in Fig. 2.4. Text from the file ecoli represents text over the DNA alphabet. The number of symbols is very small and the probability of all symbols are similar. The second text is bible, a natural language text. This text contains a lot of redundancy, the probabilities of symbols are various and depend on the neighbour symbols in the text. For example the probability that the bible contains string *qqq* is much smaller than the probability for string *the*. The last text is file kennedy over the alphabet with 256 symbols.

For an input alphabet of size k, are $k + 1$ possible values for the number of successors, because we have added a special case for only one outgoing edge pointed to the successor.

[3]The length of Huffman code depends on the size of alphabet and number of occurrences of each symbol. DNA sequences have the similar number of occurrences of each symbol. Huffman code of constant length is produced for this type of text.

That is why ecoli has defined values for only five numbers. The other two texts have values defined for $k + 1$ symbols. Values of number of states for number of transition greater than 6 are not shown on the figure, because these values are very small.

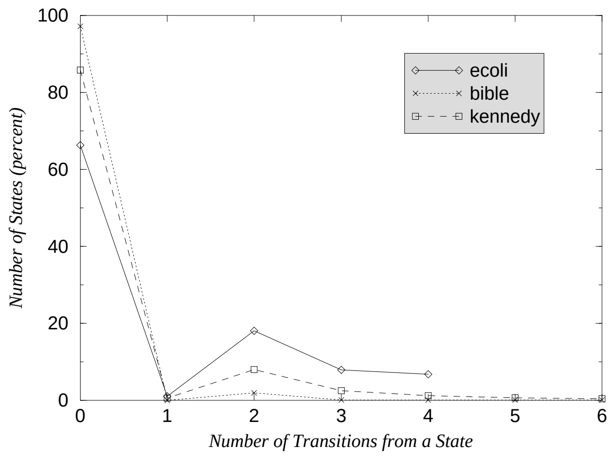


Figure 2.4: Number of outgoing edges distribution

In Fig. 2.4 vertices with just one edge leading from them were further divided into two groups: the first group is formed by vertices having just one edge leading to the next vertex according to given vertex ordering (included in the group Edge count = 0), and the second group is formed by vertices having just one edge leading anywhere else (Edge count = 1). The first group can be easily encoded by the value of Edge count.

The figure also shows that more than 97% of all vertices belong to the first group for natural text, 84% for the files over large alphabet and 62% for ecoli. This means that the codeword describing this fact should be very short. It will be only one bit long using Huffman coding. Other values of edge counts are represented by more bits according to the structure of the input text.

The smallest element of suffix automaton represents a vertex with just one edge ending at the next vertex. For ecoli it is 3 (2 per symbol + 1 bit per edges) bits on average. The fact that suffix automaton consists mainly of such elements was used in the construction of the *Compact DAWG structure (CDAWG)* derived from the general suffix automaton, more details can be found in [10].

2.6.3 Edge encoding

The last part of the graph element contains references to vertices that can be accessed from the current vertex. These references are realized as relative addresses with respect to the beginning of the next element. The valid values are non-negative numbers. To evaluate them it is necessary to know the ending positions of corresponding edges. This is why the code file is created by analysing *DAWG* from the last vertex towards the root in an order that excludes negative edges. If we wanted to work with these edges, we would have to reserve an address space to be filled in later when the position of the ending vertex is known.

The address space for a given edge depends on the number of bits representing the elements (vertices) lying between the starting and ending vertices. As the size of these elements is not fixed (the size of the dynamic part depends mainly on element addresses), it is impossible to obtain an exact statistical distribution of values of these addresses, which we obtained for symbols and edges. A poor implementation of these addresses will result in the situation where elements will be more distant and the value range broader.

Yet it is possible to make an estimation based on the distribution of edge lengths (measured by the number of vertices between the starting and ending vertices). In this case the real address value might be only $q - times$ higher on average, where q is an average length of one suffix automaton element. The first estimation of optimal address encoding is based on the fact that the number of addresses covered by k bits is the same for $k = 1, 2, ..., t$, where t is the number of bits of the maximum address. We will use an address consisting of two parts: the first part will determine the number of bits of the second part, the second part will determine the distance of the ending vertex in bits. The simplest case is when the addresses are of a fixed length, then the length of an average address field is $r = s + t$, where $s = 0$, which means that $r = t$. Another significant case is a situation when the number of categories is t, then $s = \lceil \log_2 t \rceil$.

When s is chosen from an interval $s \in \langle 0, \lceil \log_2 t \rceil \rangle$, the number of categories is 2^s, the number of address bits of the $i - th$ category is $\frac{t*i}{2^s}$. An average address field length is then

$$r = s + \sum_{i=1}^{2^s} \frac{t * i}{2^s}.$$

When we rearrange this formula, we obtain

$$r = s + t\frac{2^s + 1}{2^{s+1}}.$$

When the address length is fixed and the number of categories varies, this function has a local minimum for

$$2^s = \frac{t \ln 2}{2}.$$

If we know t, we can calculate s as

$$s = \log_2(t \ln 2) - 1$$

Tab. 2.6.3 shows optimal values of t for given values of s as well as address limits when it does not matter if we use a code for s or $s + 1$ categories. The estimation of the input file length assumes that the code file is three times greater than is the length of the input text, and that the code file contains the longest possible edge, which connects the initial and the last vertices. This observation is based on experimental evaluation.

It can be seen that the value $s = 3$ is sufficient for a wide range of input text file lengths, which guarantees a simple implementation, yet it leaves some space for doubts about the quality of the approach used or the edge lengths are not spread uniformly in the whole range of possible edge lengths (1 to the maximum length). The answer to these doubts can be found in Fig. 2.5.

Fig. 2.5 does not contain edges ending at the next vertex (with respect to the actual vertex) as they are dealt with in a different way. It can clearly be observed that the assumption of uniformity of the distribution is not quite fulfilled. Nevertheless categories can be constructed in a way that supports the requirement of the minimal average code word length. Fig. 2.6 depicts the real distribution of address lengths for one way of encoding.

s	**t**	**Optimal $|\mathbf{X}|$**
1	6	3B
1 and 2	8	11B
2	12	171B
2 and 3	16	2.7kB
3	23	350kB
3 and 4	32	180MB
4	46	2.9TB
4 and 5	64	$8 \cdot 10^{17}$B
5	92	$2 \cdot 10^{26}$B

Table 2.3: Address encoding

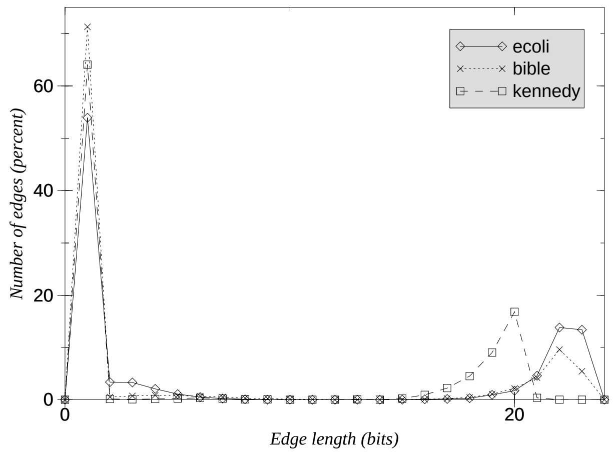


Figure 2.5: Edge Length Distribution

This encoding regularly divides address codes into eight categories by four bits. The longest address is 32 bits long. It corresponds to the maximal size of the text of about 100 MB long. The relevancy with respect to the statistical distribution of edges is obvious, the peaks being shifted by three or four bits to the right.

Edge decoding depends on the number of categories used for encoding. When eight categories are used, three bits are used for symbol length code ($s = 3$). We read these three bits as an integer n. Then we calculate the number of bits that represent an edge address as $t = (n + 1) * 3$. Finally, we read n bits from $CodeFile$ as an integer, and this number is the address.

2.7 Matching algorithm over the implementation

The matching algorithm over the implementation is shown in Fig.2.7. Each element (symbol labeling, number of edges encoding, edge encoding) is decoded at the moment of its use.

Variable Ptr points to the implementation and each bit of the implementation is addressable using Ptr. The beginning of the implementation has the value $Ptr = 0$. Function

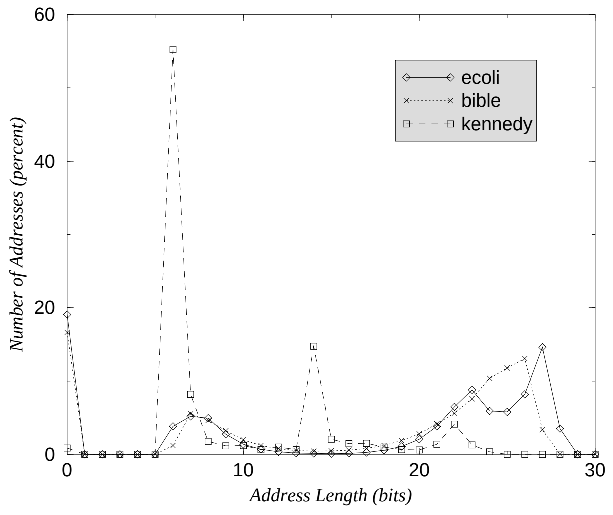


Figure 2.6: Address length distribution

$DecodeNum(Ptr)$ decodes number of edges. The decoding process starts from the root of the Huffman tree, the input bit stream is taken from the position Ptr. The number of used bits is stored for the function $Update(Ptr)$. The executing process in function $DecodeLabel(Ptr)$ is analogical, but for decoding we use a Huffman tree constructed for edge labeling instead of a Huffman tree constructed for the number of edges.

If the appropriate edge from the currently processed vertex is searched then all possible vertices are visited. Each symbol stored by the visited vertex is compared with the labeling of the appropriate edge. The maximal number of visited vertices equals the maximal number of edges from a vertex, the size of the input alphabet. The time complexity of execution of this function is $\mathcal{O}(m)$, where m is the length of the pattern.

2.8 Example

An example of suffix automaton is shown in Fig. 2.8. This automaton is build for the text $T = acagac$ over the alphabet for DNA sequences, i.e. $\{a, c, g, t\}$. The pattern $P_1 = aga$ is a substring of text T, the pattern $P_2 = cga$ does not occur in text T, pattern P_2 is not a substring of text T.

String matching over an automaton is shown in Fig.2.1. String matching for the pattern $P_1 = aga$ begins in the initial state number 0. The first symbol of the pattern (a) is matched with the transition labeled by the same symbol, while the actual state is set to the number 1. Then the second symbol from the pattern is processed and actual state is set to state 4. If no transition from the actual state has the same labeling, the matching process terminates and the pattern does not occur in the text. In our case the last symbol is matched with the transition labeled by the same symbol, and the state number 5 is accessed. This state is not the final state, so pattern $P_1 = aga$ is not a suffix of text $T = acagac$, but this pattern is a substring of the text.

Input: Implementation of $SA(T)$, pattern $P = p_1p_2 \dots p_m$.
Output: YES if P occurs in T, NO otherwise.

```
1. $Build\ decoding\ trees\ from\ the\ implementation$;
2. $Ptr := 0$; //$Ptr$ ...pointer into the implementation
3. $i := 1$;
4. $initialize\ Stack$;
5. while$((i < m + 1)$ and $(Ptr \neq NULL))$ do
6.  begin $num := DecodeNum(Ptr); Update(Ptr)$;
7.    if$(num =$ "$Only\ one\ edge\ to\ the\ next\ vertex$"$)$ then $Push(Stack, 0)$;
8.      else while$(num > 0)$ do
9.        begin $Push(Stack, DecodeEdgeLength(Ptr))$;
10.         $Update(Ptr)$;
11.         $num := num - 1$;
12.       end; //end while
13.   $found := FALSE$;
14.   while$((Empty(Stack) \neq TRUE)$ and $(found \neq TRUE))$ do
15.     begin $Ptr := Ptr + Pop(Stack)$;
16.       if$(DecodeLabel(Ptr) = P[i])$ then
17.        begin $Update(Ptr)$;
18.          $i := i + 1$;
19.          $found := TRUE$;
20.          $initialize\ Stack$;
21.        end; //end if
22.     end; //end while
23.    if$(found = FALSE)$then $Ptr := NULL$;
24.   end; //end while
25. if$(i < m + 1)$ then NO // Pattern does not occur in Text
26.         else YES // Pattern occurs in Text
```

Figure 2.7: Matching algorithm over the implementation

2.8.1 Implementation structure

Suffix automaton is created for text $T = acagac$ (see Fig. 2.8). The structure of implementation is shown in Fig. 2.9. The first element (state) corresponds to vertex number 0, i.e. the initial state. This state has no incoming edge, so the corresponding element consists only of part storing edges. This part starts with the code for the number of outgoing edges (here number 3). Three addresses follow. Each address refers to the end vertex of one outgoing edge. These vertices are pointed as the number of bits necessary to shift for pointing to end vertex. The first address points to the element number 1. It corresponds to the transition labeled by symbol a. The length of this address is 0, because element number 1 follows stored element. The lengths of the next transitions depend on the size of elements leading between stored and referred elements. That is why we store elements from the last element to the first one in numbering of topological ordering.

The second element (number 1) consist of two parts, vertex and edges. The vertex carries information about the labeling of incoming edge, here symbol a. Edges starts with the number

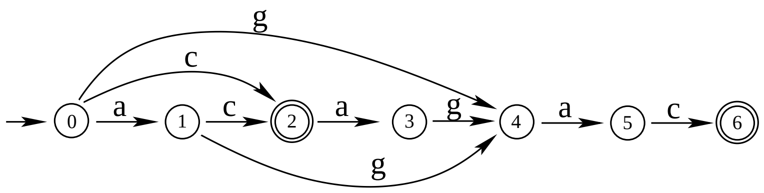


Figure 2.8: An automaton $SA(acagac)$

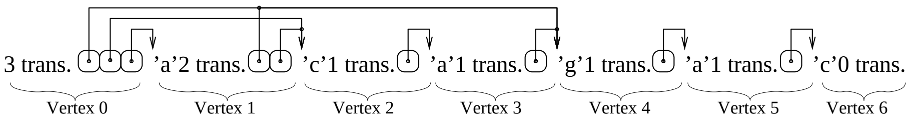


Figure 2.9: Implementation of $SA(acagac)$

of outgoing edges, here two edges. The first points to the third element (number 2) the second points to the fifth element (number 4). The third element starts with symbol labeling (c). The element contains only one edge and this edge points to the next element. The special category is added for this case in the part of number of edges. Therefore the part of addresses is not present because all needed information is saved in previous code.

The meaning of individual parts of next elements are similar. The last element has no outgoing edge, so we create a new category for the number of outgoing edges; this value is 0. In order to save the space for implementation of large texts it is better to save this element in another way. The element for vertices without any outgoing edge consists of a label of the incoming edge, and the number of edges is set to 1. This vertex contains only one edge, and this edge does not point the next vertex (this situation is encoded in another way). The address is set to 0. This is an inadmissible value for an address that points to the non next vertex, so this combination does not occur in another place in the stream. This detail is for decreasing the storage requirement, and is not used in the example.

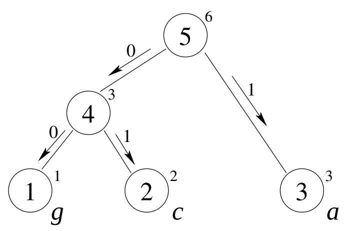
Addresses are saved relatively. Three addresses are saved for element number 0, the first points to the element 1, the second points to the element 2 and the last one to the element 4. All addresses in our example are in bits from the end position of element 0, but in a real implementation the first address points from element 0 to element 1, the second from 1 to 2 and the last one from 2 to 4.

2.8.2 Huffman trees

An algorithm for constructing a Huffman tree is shown in Fig. 2.3. Huffman code is used for storing symbol labellings and the number of edges outgoing from a state. The statistical distribution is prepared during the construction phase.

Table in the Fig. 2.10 shows the statistical distribution (the second column) for each symbol (the first column) and the resulting Huffman code (the last column). The Huffman code for the number of edges is shown in the Fig. 2.11. In the example we do not use the special coding for storing final state, so we have six categories for the number of outgoing edges. The category of four successors and one successor to the not next vertex are not present; they have no code. The table and Huffman tree is in Fig. 2.11 and have the same meaning as in Fig. 2.10.

The decoding of an input stream begins at the *root* of the coding tree and follows a left



Symbol	f	code
a	3	1
c	2	01
g	1	00
t	0	–

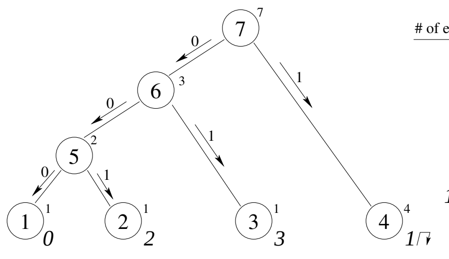
Figure 2.10: Example: Huffman code for symbols

edge when a '0' is read or a right edge when a '1' is read. When a leaf is reached, the corresponding symbol is output. For example, we read the stream on the position, where it is the following sequence of bits: 01001.... We want to decode one Huffman code for the symbol of input alphabet. We want to get this symbol and the new position in the stream (the length of read Huffman code). We use the Huffman tree (see Fig. 2.10). We read the first bit from the stream - 0 and we follow from the root - vertex number 5 to the vertex 4. We read the next bit 1 and follow to the vertex 2. This is a leaf, the corresponding symbol c is decoded. Two bits were read, and the pointer in the stream is updated.

Table 2.2 shows experimental results for a set of texts. The fifth column shows average size (in bits) used for storing one symbol. This average size as for text $T = acagac$ is calculated using formula:

$$as = \frac{\Sigma_{i=1}^{k} n_i * l_i}{\Sigma_{i=1}^{k} n_i} = \frac{(3*1)_a + (2*2)_c + (1*2)_g + (0*0)_t}{3+2+1} = 1.5$$

This number means that 1.5 bits are used for storing one input symbol in the average.



# of edges	f	code
4	0	–
3	1	01
2	1	001
1	0	–
1↴	4	1
0	1	000

Figure 2.11: Example: Huffman code for symbols

2.8.3 Addresses

The last part of the vertex contains references to vertices that can be accessed from the current vertex. These references are realized as the relative addresses with respect to the beginning of the next element. The valid values are non-negative numbers. The longest address can point to the last vertex from the first one. We assume that the space requirements for storing text is three times greater than the original text. The longest edge for text $T = acagac$ connects the first and the last state. Appropriate address points to the last vertex from the end of the first vertex. That means that this address shifts over four vertices. We suppose that each vertex is saved using three bytes. The resulting address is $4 * 3 * 3 = 36$ bits long in the worst case. Six bits are used for storing this address.

Table 2.6.3 shows that one category for addresses is the best for text T of the length of 6 symbols. We use two categories, because we want to demonstrate the use of categories. The first category (encoded by 0) determines the address three bits long, the second category (encoded by 1) determines address six bits long.

2.8.4 Construction of the implementation

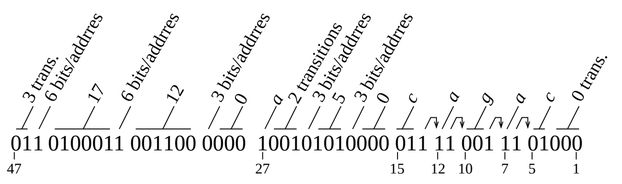


Figure 2.12: Implementation of $SA(acagac)$

The resulting implementation for our example is shown in Fig. 2.12. The creation of this bit stream starts with the terminal state of the suffix automaton. Therefore the bit bit stream is indexed from the last bit of the implementation (number 1) to the first bit (number 47 in our case). The special importance have indexes pointing to the beginnings of individual vertices. These values are used for evaluating lengths of addresses.

The whole information in each step about one state is saved and the corresponding part of bit stream is denoted as a vertex. The last state is processed in the first step. This state has no outgoing transition, so 000 is saved. This code corresponds to the Huffman tree for the number of edges. In front of this bits are saved 01, the Huffman code for symbol $'c'$. The last vertex is created, so the starting position of this is saved for computing the destination position of some edge, here 5. The next vertex (vertex number 5) consists of two parts, the symbol code (here 1 for symbol a) and the number of edges (only one edge and this edge points to the next vertex – 1). This is the smallest vertex in the implementation, consisting only of two bits. Its starting position is 7 bits from the end.

Vertices 4, 3 and 2 are created in the same way, the corresponding positions are 10, 12 and 15. Two transitions go out from the state 1. Therefore vertex 1 contains two addresses, one points to vertex 2 and the second address points to vertex 4. The length of the address is the difference between the bit leading behind vertex actually saved and the position of the starting bit of the accessed vertex. The first address points from the vertex 1. Its starting position is 15. It corresponds to the beginning of the next vertex. The destination of the first address is vertex 2, its position is 15. The difference is 0, so the value of the first address is 0. This value is less than 7 ($7_{10} = 111_2$), the category is the first, the code is 0. The whole address is encoded as 0 000. The second address is obtained in the same way and the resulting value is $15 - 10 = 5$ and code 0 101. Vertex number 0 is saved in the same way. The resulting code is 47 bits long. It is less than one byte per input symbol. The lengths of storage space for Huffman trees must be added. This space does not increase the ratio between implementation and the text for larger text.

2.8.5 Complexity

Suffix automaton can be created using the on-line construction algorithm in $\mathcal{O}(n)$ time [8]. Vertex re-ordering can be also done in $\mathcal{O}(n)$ time, encoding of suffix automaton elements as described above can also be done in $\mathcal{O}(n)$ time. Moreover, vertex re-ordering can be done during the first or third phase. This means that the described suffix automaton construction can be performed in $\mathcal{O}(n)$ time.

The time complexity of searching in such an encoded suffix automaton is $\mathcal{O}(m)$, see [11].

2.9 Results

File	Source	Length	$\|A\|$	*DAWG—SA*	*CDAWG*	*Suff. tree*	*SA.1*	*SA.2*
book1	CL	768771	81	30.35	15.75	9.83	3.75	3.66
book2	CL	610856	96	29.78	12.71	9.67	3.25	3.17
paper1	CL	53161	95	30.02	12.72	9.82	3.09	2.98
paper2	CL	82199	91	29.85	13.68	9.82	3.19	3.06
paper3	CL	46526	84	30.00	14.40	9.80	3.24	3.12
paper4	CL	13286	80	30.34	14.76	9.91	3.21	3.04
paper5	CL	11954	91	30.00	14.04	9.80	3.14	2.97
paper6	CL	38105	93	30.29	12.80	9.89	3.08	2.96
alice29	CN	152089	74	30.27	14.14	9.84	3.34	3.20
lcet10	CN	426754	84	29.75	12.70	9.66	3.21	3.12
plrabn12	CN	481861	81	29.98	15.13	9.74	3.60	3.52
bible	CN	4047392	63	29.28	10.87	7.27	3.01	2.94
world192	CN	2473400	94	27.98	7.87	9.22	2.58	2.53
bib	CL	111261	81	28.53	9.94	9.46	2.76	2.68
news	CL	377109	98	29.48	12.10	9.54	3.25	3.15
progc	CL	39611	92	29.73	11.87	9.59	2.98	2.87
progl	CL	71646	87	29.96	8.71	10.22	2.48	2.40
progp	CL	49379	89	30.21	8.28	10.31	2.44	2.35
trans	CL	93695	99	30.47	6.69	10.49	2.41	2.35
fields	CN	11150	90	29.86	9.40	9.78	2.54	2.43
cp	CN	24603	86	29.04	10.44	9.34	2.75	2.64
grammar	CN	3721	76	29.96	10.60	10.14	2.48	2.36
xargs	CN	4227	74	30.02	13.10	9.63	2.90	2.75
asyoulik	CN	125179	68	29.97	14.93	9.77	3.49	3.34
geo	CL	102400	256	26.97	13.10	7.49	3.27	3.18
obj1	CL	21504	256	27.51	13.20	7.69	3.12	2.98
obj2	CL	246814	256	27.22	8.66	9.30	2.75	2.67
ptt5	CN	513216	159	27.86	8.08	8.94	1.70	1.63
kennedy	CN	1029744	256	21.18	7.29	4.64	1.65	1.57
sum	CN	38240	255	27.79	10.26	8.92	2.62	2.53
ecoli	CN	4638690	4	34.01	23.55	12.56	4.59	4.46

Table 2.4: Relative space requirements (in bytes per input symbol) of suffix automaton and *suffix tree*

The table 2.4 shows the results for the set of test files. The first four columns have the same meaning as on previous tables. The fifth, sixth and seventh columns (*DAWG*—suffix automaton, *CDAWG* and *Suff. Tree*) cite the results of implementation published by Kurtz in [1] and show the relative space requirement (in bytes per input symbol). Standard methods are used for storing this structures for string matching. These are suffix automaton ($DAWG$), $CompactDAWG$ ($CDAWG$ for short) and the *suffix tree*. The best method from [1] is cited for storing the *suffix tree*. $CDAWG$ and *suffix tree* use the original text for string matching, but the length of the text is not increased to the shown ratio.

The last two columns show the results of the implementation presented in this paper. These values do not include the space requirements for storing Huffman trees. The space requirement of the Huffman tree depends on the size of the input alphabet and statistical distribution of input symbols. The simplest way to store a Huffman tree is to store the number of occurrences (for each symbol). It is necessary to build a Huffman tree for each match in this case. Brodnik and Carlson in [12] show how to implement a Huffman tree. Their method permits sublinear decoding and the size of the extra storage except for the storage of the symbols of input alphabet is $\mathcal{O}(l \log A)$ bits, where l is the longest Huffman code and A is the number of symbols in the alphabet. The size of the resulting structure is very small with respect to the size of the text even for small texts. This size does not depend on the size of the text.That is why this size is not added to the implementation.

The first values ($SA.1$) correspond to the storing addresses using eight categories (three bits per category code). These categories regularly divide possible lengths from the length 0 bits to the maximal size, 32 bits. The first category corresponds to addresses 4 bits long, the second corresponds to addresses 8 bits long, etc. This way of address encoding is sufficient for texts less than approximately 100 MB. Some categories are not used if the suffix automaton created for a small text.

The second set of values is obtained for a code with three categories, where the first category denotes the address of length zero, i.e no address is stored and the referred element leads next to processed element. The next categories regularly divide possible lengths from the length 4 bits to the maximal size. The maximal size of address depends on the size of text, if we assume that the space requirements for storing text is five times greater than the original text, the suffix automaton corresponding to this text can contain a long edge from first element to the last one. This method is useful for all texts, but four categories are better for texts over 180 MB, see Table 2.6.3.

2.10 Conclusion

A new method of suffix automaton implementation is presented. The results show that the ratio of code file size to the input file size is about 3:1. This number changes very little as the input file size increases with small detriment of the code file. If the ratio rose as high as 4:1, a CD-ROM with the capacity of 600MB could contain one code file for a text of the maximal size up to 150MB, which is a more than seven-times better than the result obtained by the classical approach.

The time to match a pattern using our implementation of suffix automaton depends on the size of the pattern and does not depend on the size of text.

The principle of our implementation can be used for all homogenous automata, i.e. automata having all edges to one state labeled by the same symbol. It is possible to transfer edge labeling from the destination state to the source state. This will increase the space requirement, because labeling of each edge is stored in a unique place. In our implementation labeling of all edges incoming to one state is stored in this place. Using this transformation it is possible to extend the principle of our implementation to all automata, that accept finite language, i.e. all acyclic graphs. Storage space does not increase rapidly. Table 2.1 shows that the number of outgoing transition of a state is about 1.5 in the average, and the storage of edge labeling increases 1.5 times, i.e larger by 50 percent. The number of bits necessary to store one label is about 5 bits (see Table 2.2). The labels are stored by each state, and the number of states versus the number of symbols is about 1.6 (see Table 2.1). The number of bits necessary to encode edge labeling in our implementation is $1.6 * 5 = 8$ bits per input symbol and $1.6 * 5 * 1.5$ bits per input symbol after transferring labels from states. The ratio between the size of the implementation and the size of the text is 0.5 byte larger. The transfer of edge labeling from incoming state to the outgoing states accelerate the matching process, because the appropriate edge can be determined in actually processed state, without visiting all following states.

References

[1] Kurtz S. *Reducing the Space Requirement of Suffix Trees.* Software–Practice and Experience, 29(13), 1999; 1149-1171.

[2] Gonnet G.H, Baeza-Yates R. *Handbook of Algorithms and Data Structures - In Pascal and C.* Addison - Wesley, Wokingham, UK, 1991.

[3] Anderson A, Nilson S. *Efficient implementation of suffix trees.* Software-Practice and Experience, 25(1995); 129–141.

[4] Kärkkäinen J. *Suffix cactus: A cross between suffix tree and suffix array.* in Proc. 6th Symposium on combinatorial Pattern Matching, CPM95, 1995; 191-204.

[5] Irving R.W. *Suffix binary search trees.* Technical report TR-1995-7, Computing science Department, University of Glasgow, Apr.95.

[6] Huffman, D.A. *A method for construction of minimum redundancy codes.* Proceedings of IRE, Vol.40, No.9, Sept.1952; 1098-1101.

[7] http://corpus.canterbury.ac.nz/.

[8] Crochemore M, Rytter W. *Text Algorithms.* Oxford University Press, New York, 1994.

[9] Melichar B. *Fulltext Systems.* Publishing House of CTU, Prague, 1996, in Czech.

[10] Crochemore M, Vérin R. *Direct Construction Of Compact Directed Acyclic Word Graphs.* CPM97, A. Apostolico and J. Hein, eds., LNCS 1264, Springer–Verlag, 1997; 116-129.

[11] Balík M. *String Matching in a Text.* Diploma Thesis, CTU, Dept. of Computer Science & Engineering, Prague, 1998.

[12] Brodnik A, Carlsson S. *Sub–linear Decoding of Huffman Codes Almost In–Place.* 1998.

3 String Matching in a Compressed Text
Jan Lahoda

3.1 Motivation

Lets imagine that we are given a text T_U (for example $T_U = aacabaab$) and a pattern P (for example $P = ab$). Using one of the algorithms described in Chapter 2, it is quite trivial task to find all occurrences of P in T_U. The problem arises if we are not given the plain text (T_U), but the same text compressed using some compression method (this text is denoted T_C and in our example it is $T_C = 00110100010$). If the task is to find all occurrences of P in T_U using only P and T_C, the solution is not so trivial.

To solve such a problem, two approaches can be used. The first is to decompress the text and than do the pattern matching (search for the occurrences). The second is to develop a brand new algorithm for pattern matching in text compressed by a particular compression method.

In this chapter, the mentioned problem will be solved using the second approach. We will use finite automata to solve pattern matching in compressed text, in particular in text coded by Huffman coding.

3.2 Static Huffman code

Huffman coding is a statistical compression method. It achieves compression by assigning shorter codes to more probable symbols and longer codes to less probable symbols. The resulting code is over alphabet $\{0, 1\}$. The static Huffman code assigns always the same code to the same symbol. The probability of the symbol may be given as probability distribution $p(a_I)$ for all $a_I \in A_I$.

The Huffman tree t is one possibility of computing the codes for each symbol using the probability distribution. It is a binary tree and has one leaf for each symbol $a_I \in A_I$. Each node of the tree holds a probability. Leaves hold probability of the respective input symbols and each of the inner nodes holds the sum of probabilities of both its children. The tree is constructed in the bottom-up manner starting from leaves, and each time a new inner node is created, two existing nodes holding the two smallest probabilities are used as its children. The process of course ends when the root of the tree is created. After that we assign a value of $\{0, 1\}$ to each edge, the left child of the particular node gets value 0 and the right child gets value 1. The Huffman code is composed from the values of edges from the root node into the node corresponding to the symbol being coded.

The example of Huffman tree for $A_I = \{a, b, c\}$, $p(a) = 0.5$, $p(b) = 0.25$, $p(c) = 0.25$ is in Figure 3.2.

3.3 Pattern matching in Huffman coded text

In this section we will demonstrate the algorithm of pattern matching in the Huffman coded text. An algorithm for finding all occurrences of a pattern P in the uncompressed text T_U using only compressed text T_C will be shown.

The pattern matching in the compressed text will be based on an automaton $M_C = (Q_C, \{0, 1\}, \delta_C, q_{C0}, F_C)$. This automaton will be constructed using the Huffman tree t (the Huffman tree used to compress the text) and the pattern matching automaton M solving the particular pattern matching problem $\mathcal{P}$ for set of patterns P.

Definition 3.1
The deterministic finite automaton $M_H = (Q_H, \{0,1\}, \delta_H, q_{H0}, F_H)$ is an automaton whose transition diagram is the tree t, starting state q_{H0} is the root of t and the final states F_H are the leaves of t. □

Theorem 3.2
The state and space complexity of the automaton M_H is $\mathcal{O}(|A_I|)$ and it can be constructed in $\mathcal{O}(|A_I|)$ time.

Proof
The number of nodes in Huffman tree is $\mathcal{O}(|A_I|)$. That means state complexity is also $\mathcal{O}(|A_I|)$ and size complexity is $\mathcal{O}(|A_I| \cdot |\{0,1\}|)$, that is $\mathcal{O}(|A_I|)$.

The transition table is of size $\mathcal{O}(|A_I|)$. For each state $q_H \in Q_H$ and symbol $b \in \{0,1\}$ we can compute the target state q'_H of the transition in $\mathcal{O}(1)$ time, and therefore we need only $\mathcal{O}(|A_I|)$ steps to fill this table, and as the transition table is the biggest structure in the automaton M_H we need $\mathcal{O}(|A_I|)$ time to construct the automaton. □

To create an automaton which performs pattern matching in compressed text, the automata M and M_H are combined and the result is denoted $M'_C = (Q'_C, \{0,1\}, \delta'_C, q'_{C0}, F'_C)$. The combining method is as follows: first each state from automaton M is replaced by the whole automaton M_H. After that each transition from M is replaced by an ε-transition, starting in the state that corresponds to the symbol of the original transition, and ending in the root of the tree that corresponds to the original target state.

The automaton M'_C is not deterministic because it contains ε-transitions. The deterministic version of this automaton could be created using Algorithm 3.4 by removing the ε-transitions. This automaton is denoted $M_C = (Q_C, \{0,1\}, \delta_C, q_{C0}, F_C)$.

Theorem 3.3
Each state $q \in Q'_C$ that is a source state of an ε-transition is the target state of exactly one transition, and this transition is not an ε-transition. Each state is source state of at most one ε-transition.

Proof
Both these facts lead from the way of constructing automaton M'_C.

Each node that is the starting state of an ε-transition is a leaf node of the original tree t. Each leaf of a tree has exactly one edge, and therefore state q is the target state of only one transition. This transition is not an ε-transition.

As the automaton M is deterministic, for each state $q \in Q$ and each symbol $a \in A_I$ holds that at most one transition leads from state q for symbol a. Exactly one state $q_H \in Q_H$ corresponds to each symbol $a \in A_I$. Therefore there is only one transition beginning in q_H, and it is an ε-transition. □

Algorithm 3.4
Construction of automaton M_C.
Input: Automaton M'_C.
Output: Automaton M_C.
Method:

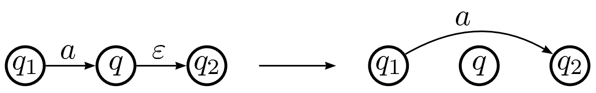


Figure 3.1: A figure for Theorem 3.5

1. For each state $q \in Q'_C$ such that exist $q_1, q_2 \in Q'_C$ and $a \in \{0,1\}$ such that $\delta'_C(q_1, a) = q$ and $\delta'_C(q, \varepsilon) = q_2$ do:
   (a) create transition $\delta'_C(q_1, a) = q_2$
   (b) remove state q from set Q'_C (including all transitions starting from or leading to this state).

Theorem 3.5
For each automaton M'_C constructed as described above a deterministic automaton M_C can be constructed using Algorithm 3.4 so that $L(M'_C) = L(M_C)$ and $|Q| \leq |Q'|$.

Proof
The automaton M'_C is deterministic, except the ε-transitions. (Proof: the automaton M_H is deterministic and M'_C is based on this automaton. The only added transitions are ε-transitions.) Therefore, we need to eliminate these transitions in a way that will not introduce any new cause of nondeterminism.

According to the assumptions and Theorem 3.3, a transformation depicted in Figure 3.1. As the ε-transition is the only transition beginning in state q_1, the new transition is also the only transition beginning in state q_2, and therefore no new nondeterminism is introduced.

If we perform this elimination for all $q \in Q'$, we clearly remove all ε-transitions and introduce no new nondeterminism. Therefore, if the automaton M'_C was deterministic except ε-transitions, it is deterministic after these eliminations.

As we do not create any new states, it is clear that $|Q| \leq |Q'|$. □

It is even possible to construct directly the automaton M_C using automata M and M_H without constructing automaton M'_C.

Algorithm 3.6
Direct construction of automaton M_C.
Input: Automata M and M_H.
Output: Deterministic automaton M_C
Method:

1. Construct $Q_C = Q \times (Q_H \backslash F_H)$. The states from Q_C will therefore be pairs (q, q_H), $q \in Q$, $q_H \in (Q_H \backslash F_H)$.
2.
$$\delta_C((q, q_H), b) = \begin{cases} (q, \delta(q_H, b)) & \text{if } \delta_H(q_H, b) \notin F_H \\ (\delta(q, a), q_{H0}) & \text{otherwise} \end{cases}$$
   $a \in A_I$ is the symbol corresponding to the state $\delta_H(q_H, b)$ (the state $\delta_H(q_H, b)$ is an image of a node in the Huffman tree t, a is the symbol corresponding to this node).

3. Set $q_{C0} = (q_0, q_{H0})$.
4. $F_C = \{(q, q_{H0}) : q \in F\}$.

Theorem 3.7
The pattern matching automaton M_C has at most $\mathcal{O}(|Q| \cdot |A_I|)$ states, its space complexity is $\mathcal{O}(|Q| \cdot |A_I|)$ and it is constructed in $\mathcal{O}(|Q| \cdot |A_I|)$ time.

Proof
The set of states of M_C is constructed as a Carthesian product of two sets: the state set of the automaton M and the set of nodes of the Huffman tree (we can omit subtraction of F_H for worst case complexity). The resulting state complexity is a simple product of sizes of these two sets, that is $\mathcal{O}(|Q| \cdot |A_I|)$.

The space complexity of automaton M_C is $\mathcal{O}(|Q_C| \cdot |\{0, 1\}|)$, that is $\mathcal{O}(|Q| \cdot |A_I|)$.

Steps 1 and 4 takes at most $\mathcal{O}(|Q| \cdot |A_I|)$ time. Each sub-step of Step 2 takes $\mathcal{O}(1)$ time and they are repeated at most $\mathcal{O}(|Q| \cdot |A_I|)$ times. Step 3 takes $\mathcal{O}(1)$. Therefore Algorithm 3.6 takes $\mathcal{O}(|Q| \cdot |A_I|)$ time. $\square$

Using automaton M_C the compressed pattern matching is quite simple and is very similar to pattern matching in uncompressed text. The only problem that needs to be solved is that one symbol in the compressed text does not correspond to one symbol in the uncompressed text. Therefore the position in the uncompressed text cannot be computed from position in the compressed text. In order to solve this problem it is necessary to count how many times the execution passes through a state corresponding to the root of the Huffman tree. This number is precisely the position in the uncompressed text. The complete compressed pattern matching algorithm is given as Algorithm 3.8.

Algorithm 3.8
Pattern matching in the Huffman coded text.
Input: Pattern matching problem $\mathcal{P}$, set of patterns P, compressed text $T_C = b_1 b_2 \dots b_{n_c}$.
Output: Position(s) of occurrence(s) of a pattern from P in the uncompressed text T_U.
Method:

1. Construct automaton M_H using tree t.
2. Construct a pattern matching automaton M for problem $\mathcal{P}$ and set of patterns P.
3. Using automata M_H and M construct automaton M_C (algorithm 3.6).
4. $q_c = (q_0, q_{H0})$.
5. If $q_c \in F_C$, mark occurrence of a pattern from P on position i.
6. Read a new input symbol b. If end of the input text, then finish.
7. If $q_c = (q, q_{H0})$ for any $q \in Q$, then $i = i + 1$.
8. $q_c = \delta_C(q_c, b)$.
9. Repeat from 5.

Theorem 3.9
Let E be time of constructing pattern matching automaton M for pattern matching problem $\mathcal{P}$ and set of patterns P. Then Algorithm 3.8 runs in $\mathcal{O}(n_C + |Q| \cdot |A_I| + E)$ worst case time

using $\mathcal{O}(|Q| \cdot |A_I|)$ extra space.

Proof
Step 3 supersedes Step 1 and it takes at most $\mathcal{O}(|Q| \cdot |A_I|)$ time (see Theorem 3.7). Step 2 takes by assumption of this theorem $\mathcal{O}(E)$. Steps 4—8 take $\mathcal{O}(1)$ time and are repeated n_C times. The worst time complexity of the whole algorithm therefore is $\mathcal{O}(|A_I| + E + |Q| \cdot |A_I| + n_C)$ and that is $\mathcal{O}(n_C + |Q| \cdot |A_I| + E)$.

The biggest created structure is automaton M_C and therefore the extra space is equivalent to the size of this automaton. The space complexity of automaton M_C is, according to Theorem 3.7, $\mathcal{O}(|Q| \cdot |A_I|)$. □

3.4 Optimized pattern matching algorithm

An important problem of Algorithm 3.8 is that the input alphabet is binary $\{0, 1\}$. Usually, the work with bits is not very efficient. The program reading data by bytes, has to "divide" the byte into bits, and to perform one transition for each bit. Although the reading is quite efficient, dividing byte into bits and transitions for bits are quite inefficient. Therefore we would rather like to use whole bytes instead of bits.

The solution is so-called alphabet extension. Two or more transitions are collapsed into one transition, effectively extending alphabet. Using this approach, one transition per byte can be performed at cost of bigger automaton (the size of alphabet increases, so does the automaton). The number of transitions collapsed into one is denoted $d \in \mathcal{N}$ ($\mathcal{N}$ is the set of natural numbers), the extended alphabet is denoted A_O ($|A_O| = |\{0, 1\}|^d = 2^d$) and the compressed text over the extended alphabet is denoted T'_C.

A pattern matching automaton $M_O = (Q_O, A_O, \delta_O, q_{O0}, \emptyset)$ is created. This automaton is constructed from M_C using alphabet extension as shown above.

In case the alphabet extension is used in the pattern matching in compressed text, there is one more problem: reading of one symbol $a_O \in A_O$ from the text T'_C may produce more than one symbol after decompressing the text[4]. Similarly more than one symbol from the text T'_C may produce only one symbol after they are decompressed (and therefore some input symbols may produce no output symbols).

The solution of these problems is to define two auxiliary functions (represented as tables) output function $N : Q_O \times A_O \to \mathcal{N}^*$ ($\mathcal{N}^*$ is in fact a string of numbers from $\mathcal{N}$) and index function $I : Q_O \times A_O \to \mathcal{N}$. The pattern matching using M_O, N and I is quite simple — we read one symbol from the text T'_C, look into the N table using current state of the automaton M_O and the read symbol. The record in this table defines the occurrences of patterns from P in the uncompressed text T_U. After that we use the function I to move imaginary pointer in the uncompressed text, and perform a transition in the automaton M_O. This is shown in Algorithm 3.12.

Algorithm 3.10
Construction of the new automaton M_O, output function N and index function I.
Input: Pattern matching automaton M_C for pattern P.
Output: New pattern matching automaton M_O, output function N and index function I both represented as tables.
Method:

[4]Using example from Section 3.2, eight symbols "a" are encoded as "00000000_2", that is one byte "00_{16}".

1. $Q_O = Q_C$, $q_{O0} = q_{C0}$.
2. For each state $q_O \in Q_O$ and each symbol s of the output alphabet A_O do
   (a) $q_t = q_O$, $i = 0$, $N[q, s] = \emptyset$.
   (b) For each symbol $b \in \{0, 1\}$ from s do
      i. If $q_t = (q, q_{H0})$ then $i = i + 1$.
      ii. If $q_t \in F_C$ then $N[q, s] = N[q, s] \cup \{i\}$.
      iii. $q_t = \delta_C(q_t, b)$.
   (c) $\delta_O(q, s) = q_t$
   (d) $I[q, s] = i$.

Theorem 3.11
The number of states of the new automaton M_O is the same as the number of states of the automaton M_C (that means $\mathcal{O}(|Q| \cdot |A_I|)$). The space complexity of the new automaton M_O is $\mathcal{O}(|Q| \cdot |A_I| \cdot |A_O|)$. The worst case time complexity of Algorithm 3.10 is $\mathcal{O}(|Q| \cdot |A_I| \cdot |A_O| \cdot \lceil \log |A_O| \rceil + E)$.

Proof
According to the step 1, the set of states of automaton M_O equals to the set of states of automaton M_C and therefore these sets have equal size.

The space complexity is therefore $\mathcal{O}(|Q| \cdot |A_I| \cdot |A_O|)$.

The innermost loop contains three steps (2.b.i-2.b.iii). Each of these steps takes $\mathcal{O}(1)$ time. Step 2 requires repeating $\mathcal{O}(|Q| \cdot |A_I| \cdot |A_O|)$ times. Step 2.b requires repeating d times, that is $\lceil \log |A_O| \rceil$.

Therefore Algorithm 3.10 runs in $\mathcal{O}(|Q| \cdot |A_I| \cdot |A_O| \cdot \lceil \log |A_O| \rceil)$ time. □

Algorithm 3.12
Pattern matching in the Huffman coded text.
Input: Input text $T_C = s_1 s_2 \ldots s_c$ ($s_i \in A_O$), pattern matching problem $\mathcal{P}$, set of patterns P, compressed text $T'_C = s_1 s_2 \ldots s_{n'_c}$ ($s_i \in A_O$).
Output: Position(s) of occurrence(s) of a pattern from P in the uncompressed text T_U.
Method:

1. Using tree t, construct automaton M_H.
2. Construct pattern matching automaton M for problem $\mathcal{P}$ and set of patterns P.
3. Using automata M_H and M construct automaton M_C.
4. Using Algorithm 3.10 construct automaton M_O, output function N and index function I.
5. $q_O = q_{O0}$.
6. Read a new input symbol $s \in A_O$. If end of the input finish.
7. $i = i + I(q_c, s)$.
8. For each record $r \in N(q_c, s)$ report occurrence of a pattern from P in text T_U on offset $i - r$.

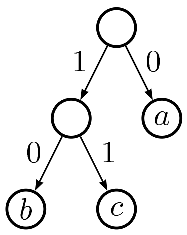


Figure 3.2: Example of Huffman tree t

9. $q_O = \delta_O(q_c, s)$.
10. Repeat from step 6.

Theorem 3.13
Let E be time of constructing pattern matching automaton M for pattern matching problem $\mathcal{P}$ and set of patterns P. Let r be the number of occurrences of P in T_U. Then Algorithm 3.12 runs in $\mathcal{O}(n'_C + r + |Q| \cdot |A_I| \cdot |A_O| \cdot \lceil \log |A_O| \rceil + E)$ worst case time using $\mathcal{O}(|Q| \cdot |A_I| \cdot |A_O|)$ extra space.

Proof
Step 3 supersedes Step 1, and it take at most $\mathcal{O}(|Q| \cdot |A_I|)$ time (see Theorem 3.7). Step 2 takes by the assumptions of this theorem $\mathcal{O}(E)$ time. Steps 5,6,7,9 take $\mathcal{O}(1)$ time and are repeated n'_C times. Step 8 takes $\mathcal{O}(r)$ time for the whole text. The worst time complexity of the whole algorithm therefore is $\mathcal{O}(n'_C + r + |Q| \cdot |A_I| \cdot |A_O| \cdot \lceil \log |A_O| \rceil + E)$.

The biggest structure created during the algorithm is automaton M_O. The space complexity of this algorithm is therefore the same as space complexity of this automaton, and it is $\mathcal{O}(|Q| \cdot |A_I| \cdot |A_O|)$. □

3.5 Case analysis

A new algorithm for pattern matching in the Huffman coded text has been shown in Sections 3.3 and 3.4. Particular pattern matching problems are analysed in this section.

Algorithms 3.6, 3.8, 3.10 and 3.12 are quite general. If a pattern matching problem $\mathcal{P}$ is being solved, the automaton M is created for this task and appropriate set of patterns P and Algorithms 3.8 and 3.12 are solving pattern matching problem $\mathcal{P}$ in the Huffman coded text. If we make use of Theorems 3.7, 3.9, 3.11 and 3.13, it is only necessary to compute the time E of constructing automaton M and its state size ($|Q|$) and the time complexity of solving the particular matching problem can be determined.

All the examples in this section are based on alphabet $A_I = \{a, b, c\}$ and probability distribution $p(a) = 0.5$, $p(b) = 0.25$, $p(c) = 0.25$. The appropriate Huffman tree is shown in Figure 3.2 and the corresponding automaton M_H is shown in Figure 3.3.

3.5.1 Exact pattern matching of one pattern

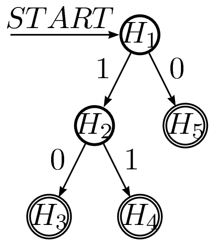


Figure 3.3: Automaton M_H corresponding to tree in Figure 3.2

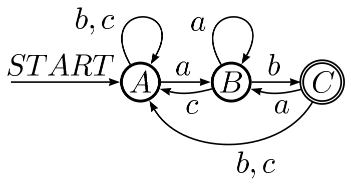


Figure 3.4: Pattern matching automaton M for pattern “ab” over alphabet $\{a, b, c\}$

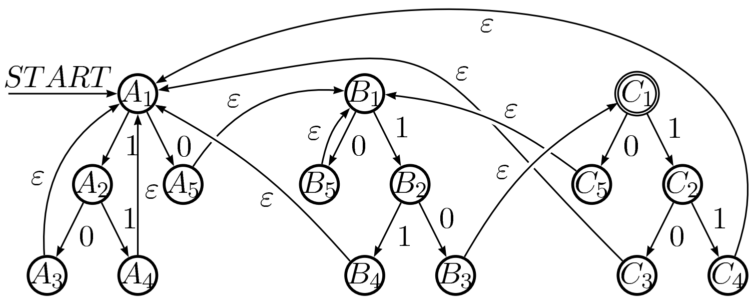


Figure 3.5: Nondeterministic automaton M'_C for one pattern matching in the Huffman coded text

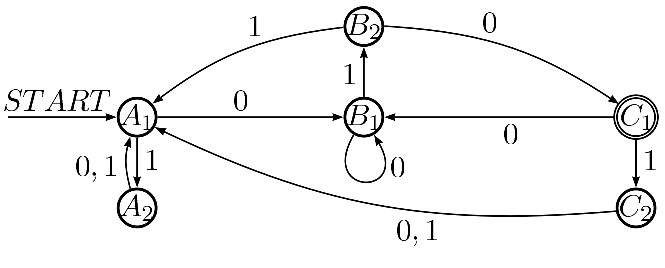


Figure 3.6: Deterministic automaton M_C for one pattern matching in the Huffman coded text

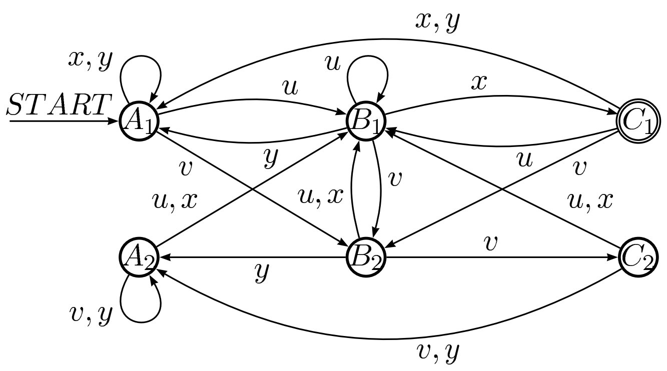


Figure 3.7: Optimized automaton M_O for one pattern matching in the Huffman coded text

The basic pattern matching problem is exact one pattern matching problem. A pattern $P = a_1 a_2 \ldots a_m$ of length m is given, and all occurrences of this pattern in the uncompressed text T_U should be found. The number of states of a pattern matching automaton for exact pattern matching of pattern of length m is $m + 1$ and it can be constructed in $\mathcal{O}(m)$ time, as described in [2].

Let $\mathcal{P}$ be exact pattern matching problem of one pattern. Then the set of patterns contains only one pattern of length m, for example: $P =$"ab". The pattern matching automaton $M = (Q, A_I, \delta, q_0, F)$ constructed for this problem and pattern P is depicted in Figure 3.4.

The automata M'_C and M_C are depicted in Figures 3.5 and 3.6, respectively. The automaton M_O constructed for $d = 2$, $A_O = \{u, v, x, y\}$ ($00 = u$, $01 = v$, $10 = x$, $11 = y$) is depicted in Figure 3.7.

According to Theorem 3.8 the exact pattern matching of one pattern in the Huffman coded text runs in $\mathcal{O}(n_C + |m| \cdot |A_I|)$ worst case time using $\mathcal{O}(|m| \cdot |A_I|)$ extra space.

According to Theorem 3.12, the "optimized" exact pattern matching of one pattern in the Huffman coded text runs in $\mathcal{O}(n'_C + r + m \cdot |A_I| \cdot |A_O| \cdot \lceil \log |A_O| \rceil)$ worst case time using $\mathcal{O}(m \cdot |A_I| \cdot |A_O|)$ extra space.

3.5.2 Approximate pattern matching

Another important pattern matching problems is approximate pattern matching with k mismatches. A pattern $P = a_1 a_2 \ldots a_m$ of length m is given and all occurrences of the pattern P in the uncompressed text T_U with at most k mismatches should be found. There are three metrics to measure the distance: Hamming distance, Levenshtein distance and generalised Levenshtein distance. According to Polcar [8], for all three distance metrics hold that the number of states of a particular pattern matching automaton is $\mathcal{O}(|A_I|^k \cdot m^{k+1})$ and therefore the space complexity of this automaton is $\mathcal{O}(|A_I|^{k+1} \cdot m^{k+1})$ and the time complexity of constructing such automata (approximately) $\mathcal{O}(|A_I|^{k+1} \cdot m^{k+1})$.

Let $\mathcal{P}$ be approximate pattern matching with k mismatches. Than the set of patterns contains only one pattern and a natural number $k \leq m$ is specified, for example: $P =$"ab", $k = 1$. The pattern matching automaton $M = (Q, A_I, \delta, q_0, F)$ constructed for this problem, pattern and distance is depicted in Figure 3.8.

The automaton M_C is depicted in Figure 3.9.

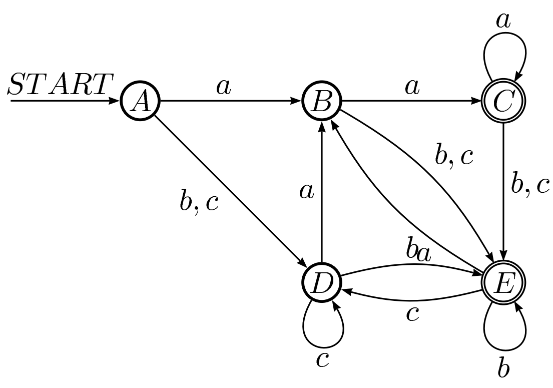


Figure 3.8: Pattern matching automaton M for approximate pattern matching of pattern "ab" over alphabet $\{a, b, c\}$

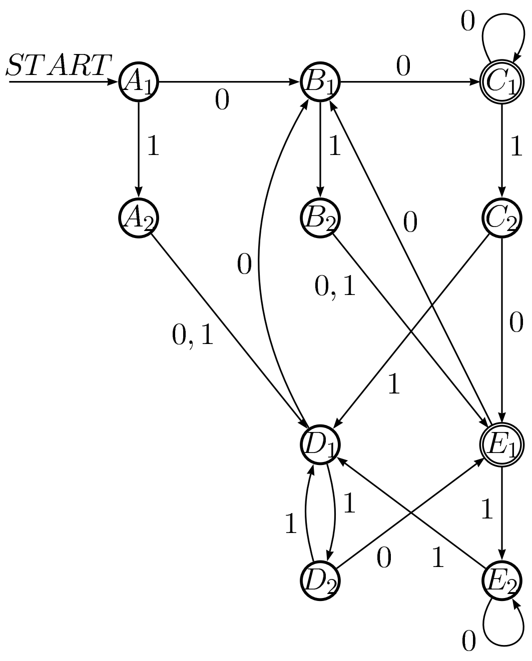


Figure 3.9: Deterministic automaton M_C for approximate pattern matching in the text compressed by the Huffman coding

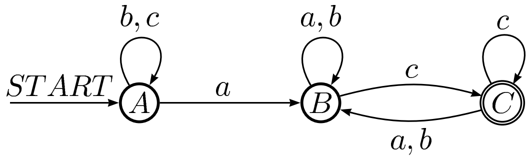


Figure 3.10: Pattern matching automaton M for regular expression pattern matching of regular expression "a(b)*c" over alphabet $\{a, b, c\}$

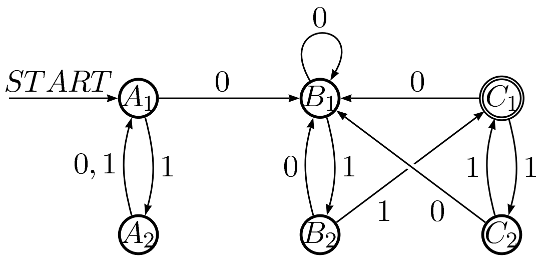


Figure 3.11: Deterministic automaton M_C for regular expression pattern matching in the text compressed by the Huffman coding

According to Theorem 3.9 the approximate pattern matching in the Huffman coded text runs in $\mathcal{O}(n_C + |A_I|^{k+2} \cdot m^{k+1})$ worst case time using $\mathcal{O}(|A_I|^{k+2} \cdot m^{k+1})$ extra space.

According to Theorem 3.13, the "optimized" approximate pattern matching in the Huffman coded text runs in $\mathcal{O}(n'_C + r + |A_I|^{k+2} \cdot m^{k+1} \cdot |A_O| \cdot \lceil \log |A_O| \rceil)$ worst case time using $\mathcal{O}(|A_I|^{k+2} \cdot m^{k+1} \cdot |A_O|)$ extra space.

3.5.3 Regular expression pattern matching

We consider P to contain a regular expression of the length m. In this case the automaton M has at most $\mathcal{O}(2^m)$ states and can be constructed in (approximately) $\mathcal{O}(2^m \cdot |A_I|)$ time.

Let $\mathcal{P}$ be regular expression pattern matching. Than the set of patterns contains only one regular expression, for example: $P =$"a(b)*c". The pattern matching automaton $M = (Q, A_I, \delta, q_0, F)$ constructed for this problem, pattern and distance is depicted in Figure 3.10.

The automaton M_C is depicted in Figure 3.11.

According to Theorem 3.9 the regular expression pattern matching in the Huffman coded text runs in $\mathcal{O}(n_C + 2^m \cdot |A_I|^2)$ worst case time using $\mathcal{O}(2^m \cdot |A_I|)$ extra space.

According to Theorem 3.13, the "optimised" regular expression pattern matching in the Huffman coded text runs in $\mathcal{O}(n'_C + r + 2^m \cdot |A_I|^2 \cdot |A_O| \cdot \lceil \log |A_O| \rceil)$ worst case time using $\mathcal{O}(2^m \cdot |A_I| \cdot |A_O|)$ extra space.

References

[1] A. Amir and G. Benson. Efficient two dimensional compressed matching. In *Proceedings of the 5th ACM-SIAM Annual Symposium on Discrete Algorithms*, pages 279–288, IEEE

Computer Society Press, 1992.

[2] J. Holub. *Simulation of Nondeterministic Finite Automata in Pattern Matching.* PhD thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic, 2000. http://cs.felk.cvut.cz/psc.

[3] S. T. Klein and D. Shapira. Pattern matching in Huffman encoded texts. In *Proceedings of the Data Compression Conference 2001*, pages 449–458, 2001.

[4] J. Lahoda and B. Melichar. Pattern matching in Huffman coded text. In *Proceedings of the Theoretical Computer Science Conference 2003*, pages 274–280, 2003.

[5] 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. http://cs.felk.cvut.cz/psc.

[6] E. S. de Moura, G. Navarro, N. Ziviani, and R. A. Baeza-Yates. Fast searching on compressed text allowing errors. In *Research and Development in Information Retrieval*, pages 298–306, 1998.

[7] E. S. de Moura, G. Navarro, N. Ziviani, and R. A. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.

[8] T. Polcar. On the state complexity of the approximate string matching. To be published.

[9] M. Takeda, Y. Shibata, T. Matsumoto, T. Kida, A. Shinohara, S. Fukamachi, T. Shinohara, and S. Arikawa. Speeding up string pattern matching by text compression: The dawn of a new era. *IPSJ Journal*, 42(3):370–384, 2001.

4 Generalized and Weighted Strings: Repetitions and Pattern Matching Michal Voráček

4.1 Introduction

In this chapter we present a finite state automata approach for searching repetitions and for pattern matching in generalized and weighted strings. Generalized and weighted strings are special strings used mainly in molecular biology to express the variability of DNA at a given position and to express uncertainity of appearance of a given base at a given position in DNA.

A sequencing process as a consequence of technology limits produces sequences which have a set of symbols instead of one symbol in a single position, and each symbol of the set has an assigned probability of occurrence. The sum of the probabilities of the symbols in each set is equal to 1. Such sequences are called *weighted sequences.*

Another type of sequences is used for expressing the variability of DNA in the population. Let us consider two or more corresponding DNA sequences of one type from the same species (e.g. coding sequences of some specific gene taken from the genomes of different people). These DNA sequences are nearly identical, differing only in a small number of positions. Therefore it is not necessary to store the single sequences separately, but we can store only one instance of such a sequence with lists of occurring symbols for the positions where the sequences differ. Such a sequence can thus serve as a representative of the whole population. Possible combinations of symbols of the DNA alphabet can be represented by special symbols. Two examples of such alphabets with special symbols are an alphabet with a "don't care" symbol and the IUB (Degenerate Bases) Code. A "don't care" symbol is a special universal symbol $\circ$ that matches any other symbol including itself, see Tab. 4.1. The IUB Code defines special symbols for all possible combinations of symbols from the basic nucleotide alphabet, see Tab. 4.1.

The abovementioned types of sequences have many common features. Therefore we propose a solution for a given problem only for one type of string and show how to use this solution for obtaining a solution for the second type of string.

This paper is organised as follows. In the next section we present some basic definitions for strings, generalized strings and weighted strings. In Section 3 we present the Generalized Factor Automaton, an algorithm for its construction and an application for searching substrings and repetitions in generalized strings. In Section 4 we show a method for transforming finite automata for pattern matching problems in strings to automata for pattern matching problems in generalized strings.

4.2 Preliminaries

4.2.1 Generalized Strings

Definition 4.1 (Generalized String)
A *generalized string* $v = v_1 v_2 \dots v_n$ over alphabet A is a finite sequence of positions where each position v_i is a subset of A, $v_i \in \mathcal{P}(A)$. Element v_i is called the i-th *symbol set.* The empty generalized string λ is an empty sequence of symbol sets. The set of all generalized strings over alphabet A (including empty string λ) is denoted $\mathcal{P}^*(A)$. We define the operation concatenation on the set of generalized strings in the following way: if v and w are generalized

strings over A, then the concatenation of these generalized strings is vw. The length $|v|$ of string v is the number of symbol sets of v. It holds that $|v| \geq 0, |\lambda| = 0$. The symbol set of string v occurring at position i is denoted v_i or $v[i]$ and its cardinality $|v_i|$ or $|v[i]|$.

Definition 4.2 (Diameter of a Generalized String)
Let $v = v_1 v_2 \dots v_n$ be a generalized string over alphabet A. The *Minimum diameter*, *maximum diameter* of v denoted $diam_{min}(v)$, $diam_{max}(v)$ is defined as $diam_{min}(v) = \min_{1 \leq j \leq n}(|v_j|)$, $diam_{max}(v) = \max_{1 \leq j \leq n}(|v_j|)$, respectively. We will use only *diameter* and denote $diam(v)$ if $diam_{min}(v) = diam_{max}(v) = diam(v)$.

Remark: The string is a special case of the generalized string having a diameter equal to 1.

Definition 4.3 (Set of Elements of a Generalized String)

The set $Elems(v)$, $v \in \mathcal{P}^*(A)$, where A is an alphabet, is a set of all elements of the generalized string v:

$$Elems(v) = \{ \ x \mid x_j \subseteq v_j, \ 1 \leq j \leq n, \ x \in \mathcal{P}^*(A) \ \}.$$

The set $Elems_k(v)$, where $0 \leq k \leq |A|$ is defined as:

$$Elems_k(v) = \{ \ x \mid x \in Elems(v) \ \& \ diam(x) = k, \quad x \in \mathcal{P}^*(A) \ \}.$$

The set $Elems_1(v)$ is also called the set of simple elements of v.

Remark: In fact, the set $Elems_1(v)$ is a language represented by v.

Definition 4.4 (Set of Factors of a Generalized String)
The set $Fact(v)$, $v \in \mathcal{P}^*(A)$, where A is an alphabet, is a set of all factors of the generalized string v:

$$Fact(v) = \{ \ y \mid v = xyz, \ x, y, z \in \mathcal{P}^*(A) \ \}.$$

Definition 4.5 (Set of Partial Factors of a Generalized String)
The set $PartFact(v)$, $v \in \mathcal{P}^*(A)$, where A is an alphabet, is a set of all partial factors of the generalized string v:

$$PartFact(v) = \{ \ x \mid \forall \ x, y : (x \in Elems(y) \ \& \ y \in Fact(v)), \ x, y \in \mathcal{P}^*(A) \ \}.$$

The set $PartFact_k(v)$, where $0 \leq k \leq |A|$ is defined as:

$$PartFact_k(v) = \{ \ x \mid x \in PartFact(v) \ \& \ diam(x) = k, \quad x \in \mathcal{P}^*(A) \ \}.$$

The set $PartFact_1(v)$ is also called the set of simple partial factors of v.

Note: The sets *Pref*, *PartPref*, *PartPref*$_k$ and *Suff*, *PartSuff*, *PartSuff*$_k$ are defined by analogy and their definitions will be omitted here. We denote by $v[i, j] = v_i v_{i+1} \dots v_{j-1} v_j$ for $i \leq i \leq j \leq |v|$ a *factor* of v starting at position i and ending at position j.

Remark: For any $v \in \mathcal{P}^*(A)$ and any $0 \leq k \leq |A|$ it holds:

1. $Fact(v) \subseteq PartFact(v)$,

2. $PartFact_k(v) \subseteq PartFact(v)$.

Example 4.6
Examples of generalized strings are depicted in Fig. 4.1-a, 4.1-e. We can omit the brackets if the cardinality of symbol set is 1. Moreover we can visualize the symbol sets as columns, see Fig. 4.1-b, 4.1-f.

4.2.2 Representing and Processing Generalized Strings

For the following examples, let us consider we have a generalized string $v = v_1 v_2 \dots v_n$ over alphabet $A = \{s_1, s_2, \dots, s_m\}$, where $v_i = \{v_i^1, v_i^2, \dots, v_i^{p_i}\}$. Our aim is to process v from left to right, in stepwise manner, symbol set by symbol set, similarly as strings are processed. There are two basic approaches to representation and processing of v_i:

1. Single symbols $v_i^j \in v_i$ are stored and processed separately.
   In this case, each symbol set can be represented as a list of symbols (Fig.4.1-a, 4.1-e) or as a column in a 2D bit array (Fig.4.1-c, 4.1-g). Examples of mechanisms convenient for separate parallel processing of symbol sets are the multi-tape finite automaton [2] or parallel finite automata [4].

2. Symbol set v_i is taken as atomic.
   We represent symbol sets by single symbols from a new alphabet (Fig.4.1-d, 4.1-h). The new alphabet has cardinality at most $2^{|A|} = 2^m$, which is the number of all subsets of A. The mapping which assigns single symbols to symbol sets is called a *subset code.* In this case, processing a generalized string is in fact classical string processing, and it can be done advantageously by a finite automaton.

Definition 4.7 (Subset Code)
Let A and E be alphabets such that $A \subseteq E$. A *subset code* is a partial mapping $\mathcal{C} : \mathcal{P}^*(A) \longmapsto E$ with the properties:

1. $\mathcal{C}$ is a bijection,

2. $\mathcal{C}(\{a\}) = a$, for all $a \in A$.

Alphabet A is called *basic* and alphabet E is called *extended.*
For generalized strings we define the mapping $\mathcal{C}^* : \mathcal{P}^*(A) \longmapsto E^*$ as follows:

1. $\mathcal{C}^*(\lambda) = \varepsilon$,

2. $\mathcal{C}^*(s) = \mathcal{C}(s)$, for all $s \in \mathcal{D}(\mathcal{C})$,

3. $\mathcal{C}^*(vs) = \mathcal{C}^*(v)\mathcal{C}(s)$, for all $v \in \mathcal{D}(\mathcal{C})^*$, $s \in \mathcal{D}(\mathcal{C})$,

where $\mathcal{D}$ denotes domain.

Note: For simplicity, we will use $\mathcal{C}$ instead of $\mathcal{C}^*$ in all cases where the meaning is obvious from the context.

a)

$v = [t][a, c][c][a, c, t][t]$

b)

$$v = t \begin{bmatrix} a \\ c \end{bmatrix} c \begin{bmatrix} a \\ c \\ t \end{bmatrix} t$$

c)

	0	1	0	1	0		a
	0	1	1	1	0		c
	0	0	0	0	0		g
	1	0	0	1	1		t

d)

$\mathcal{C}_2(v) = tmcht$

e)

$y = [t][a, c, g, t][c][a, c, g, t][t]$

f)

$$y = t \begin{bmatrix} a \\ c \\ g \\ t \end{bmatrix} c \begin{bmatrix} a \\ c \\ g \\ t \end{bmatrix} t$$

g)

	0	1	0	1	0		a
	0	1	1	1	0		c
	0	1	0	1	0		g
	1	1	0	1	1		t

h)

$\mathcal{C}_1(y) = t \circ c \circ t$

Figure 4.1: Generalized strings v, y over $A = \{a, c, g, t\}$ represented by lists (linear and column form), tapes and in subset code $\mathcal{C}_2$, $\mathcal{C}_1$, defined in Example 4.2.2, 4.2.2, respectively.

Simply said, a subset code defines how to denote the subsets of an alphabet by single symbols. Moreover this mapping must ensure that one-symbol sets will be coded by the symbols in the sets.

Example 4.8
$A = \{a, c, g, t\}$, $\mathcal{D}(\mathcal{C}_1) = \{\{a\}, \{c\}, \{g\}, \{t\}, \{a, c, g, t\}\}$,
$E_1 = \{a, c, g, t, \circ\}$.
Mapping $\mathcal{C}_1$ is defined in Tab. 4.1. Strings over alphabet E_1 are strings with "don't cares". As we see, strings with "don't cares" are only a special case of generalized strings.

Example 4.9
$A = \{a, c, g, t\}$,
$\mathcal{D}(\mathcal{C}_2) = \{\{a\}, \{c\}, \{g\}, \{t\}, \{a, c\}, \{a, g\}, \{a, t\}, \{c, g\}, \{c, t\}, \{g, t\}, \{a, c, g\}, \{a, c, t\}, \{a, g, t\}, \{c, g, t\}, \{a, c, g, t\}\}$,
$E_2 = \{a, c, g, t, m, r, w, s, y, k, v, h, d, b, n\}$.
Mapping $\mathcal{C}_2$ is defined in Tab. 4.1. This mapping is known as the *IUB (Degenerate Bases) Code* as defined by the *International Union of Pure and Applied Chemistry* [11].

Table 4.1: Definitions of the mappings $\mathcal{C}_2$ (IUB Degenerate Bases Code) and $\mathcal{C}_1$ (alphabet with a "don't care" symbol)

$\mathcal{D}(\mathcal{C}_2)$	E_2
{a}	a
{c}	c
{g}	g
{t}	t
{a, c}	m
{a, g}	e
{a, t}	w
{c, g}	s
{c, t}	y
{g, t}	k
{a, c, g}	v
{a, c, t}	h
{a, g, t}	d
{c, g, t}	b
{a, c, g, t}	n

$\mathcal{D}(\mathcal{C}_1)$	E_1
{a}	a
{c}	c
{g}	g
{t}	t
{a, c, g, t}	$\circ$

4.2.3 Weighted Strings

Definition 4.10 (Weighted String)
A *weighted string* $w = w_1 w_2 \dots w_n$ over alphabet A is a finite sequence of positions where each position w_i consists of a set of pairs. Each pair is of the form $(a, \pi_i(a))$, where $a \in A$ and $\pi_i(a)$ is the probability of symbol a appearing at position i. For every position w_i, $1 \leq i \leq n$,

$$\sum_{a \in A} \pi_i(a) = 1.$$

Example 4.11
An example of a weighted string is depicted in Fig. 4.2.

Definition 4.12 (Basic String of Weighted String)
The *basic string* of a weighted string $w = w_1 w_2 \dots w_n$ is a generalized string $v = v_1 v_2 \dots v_n$ such that v_i consists of all symbols $a \in A$ for which it holds $\pi_i(a) > 0$, where $(a, \pi_i(a)) \in w_i$, $1 \leq i \leq n$.

Definition 4.13 (Multiplicative Probability)
Let $p = p_1 p_2 \dots p_m$ be a string over A, and $w = w_1 w_2 \dots w_n$ be a weighted string over A. The multiplicative probability of the appearance of p ending at position i inside w, denoted $\mu(p, i)$, is the product of the probabilities of the symbols p_j at positions $i - m + j$, where $1 \leq j \leq m$, $m \leq i \leq n$, of the weighted string w. That is

$$\mu(p, i) = \prod_{j=1}^{m} \pi_{i-m+j}(p_j).$$

Definition 4.14 (Average Additive Probability)
Let $p = p_1 p_2 \dots p_m$ be a string over A, and $w = w_1 w_2 \dots w_n$ be a weighted string over A. The average additive probability of the appearance of p ending at position i inside w, denoted $\alpha(p, i)$, is the average of the probabilities of the symbols p_j at positions $i - m + j$, where $1 \leq j \leq m$, $m \leq i \leq n$, of the weighted string w. That is

$$\alpha(p, i) = \Big(\sum_{j=1}^{m} \pi_{i-m+j}(p_j) \Big) / m.$$

String v from Fig. 4.1. is the basic string of weighted string w from Fig. 4.2.

A weighted string can be seen as an extension of a generalized string by attribute representing probability. Similarly, a generalized string can be seen as a special case of a weighted string when the symbols having zero probability are not considered and the non-zero probabilities of the remaining symbols are equal.

4.2.4 Processing Weighted Strings

The solutions of some problems on weighted strings can be divided into two phases: solution of the problem for a generalized string, followed by probability condition verification.

$$w = \begin{bmatrix} (a,0.00) \\ (c,0.00) \\ (g,0.00) \\ (t,1.00) \end{bmatrix} \begin{bmatrix} (a,0.50) \\ (c,0.50) \\ (g,0.00) \\ (t,0.00) \end{bmatrix} \begin{bmatrix} (a,0.00) \\ (c,1.00) \\ (g,0.00) \\ (t,0.00) \end{bmatrix} \begin{bmatrix} (a,0.50) \\ (c,0.25) \\ (g,0.00) \\ (t,0.25) \end{bmatrix} \begin{bmatrix} (a,0.00) \\ (c,0.00) \\ (g,0.00) \\ (t,1.00) \end{bmatrix}$$

Figure 4.2: Weighted string w over $A = \{a, c, g, t\}$

Algorithm 4.2.4
Decomposition of pattern matching problems on weighted strings.
Input: Weighted string w, pattern matching problem P, probability condition K, other objects depending on P.
Output: List of results R'.
Method:

1. Extract the basic generalized string b from weighted string w .
2. Solve problem P without considering probability condition K for generalized string b and thus obtain the list of results R.
3. Create new list of results R' from R by removing all elements not satisfying probability condition K.

4.3 Generalized Factor Automaton

In this Section we present a novel index structure called a *generalized factor automaton* which is an extension of the *factor automaton* [5] for generalized strings. A generalized factor automaton can be used for fast substring searching and for computing regularities in generalized strings.

Definition 4.15 (Generalized Factor Automaton)
The *generalized factor automaton (GFA)* for generalized string v is a finite automaton accepting the language $PartFact_1(v)$.

4.3.1 Construction of a Generalized Factor Automaton

The construction of a generalized factor automaton is based on the construction of a factor automaton [9], see Fig. 4.3, 4.4. We take the model of a nondeterministic factor automaton (*NFA*) and for each pair of states q_{i-1}, q_i, $1 \leq i \leq n$, between which *NFA* has a transition for the symbol in position i in the string we simply add transitions for all symbols of the symbol set in position i in the generalized string.

Algorithm 4.3.1
Construction of the generalized factor automaton.
Input: Generalized string
$v = [a_{11}, a_{12}, \ldots, a_{1r_1}], [a_{21}, a_{22}, \ldots, a_{2r_2}], \ldots, [a_{n1}, a_{n2}, \ldots, a_{nr_n}]$.
Output: Deterministic generalized factor automaton $M = (Q, A, \delta, q_0, F)$ accepting the set $PartFact_1(v)$.

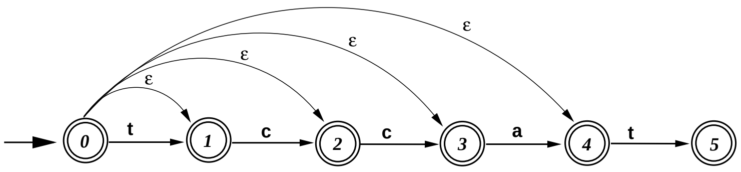


Figure 4.3: Transition diagram of the nondeterministic factor automaton for $x = tccat$

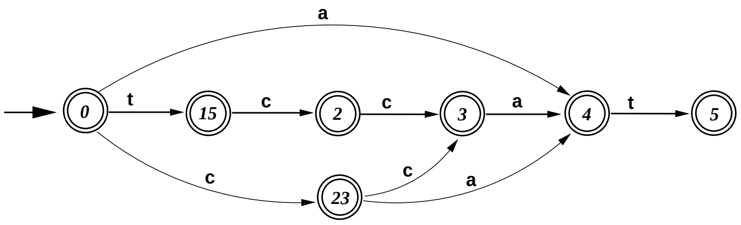


Figure 4.4: Transition diagram of the deterministic factor automaton for $x = tccat$

1. Construct a finite automaton $M_1 = (Q_1, A, \delta_1, q_0, F_1)$ accepting the set $PartPref_1(v)$:
   $Q_1 = \{q_0, q_1, q_2, \dots, q_n\}$,
   A is the basic alphabet of v,
   $\delta_1(q_{i-1}, a_{ij}) = q_i, j = 1, 2, \dots, r_i$, and $i = 1, 2, \dots, n$,
   $F_1 = \{q_0, q_1, q_2, \dots, n\}$.
2. Construct finite automaton M_2 from automaton M_1 by inserting ε–transitions:
   $\delta(q_0, \varepsilon) = \{q_1, q_2, \dots, q_{n-1}\}$.
3. Replace all ε–transitions by non–ε–transitions. The resulting automaton is M_3.
4. Construct the deterministic finite automaton M equivalent to automaton M_3.

Example 4.16
Construct a generalized factor automaton for generalized string $v = t[a, c]c[a, c, t]t$.

1. We create an automaton $M_1 = (Q_1, A, \delta, q_0, F_1)$ accepting $PartPref_1(v)$. Since $|v| = 5$ we create the set of states $Q_1 = \{0, 1, 2, \dots, 5\}$.
   $A = \{a, c, t\}$,
   $q_0 = 0$,
   $F_1 = \{0, 1, \dots, 5\}$,
   $\delta(0, t) = 1$,
   $\delta(1, a) = 2,\ \delta(1, c) = 2$,
   $\delta(2, c) = 3$,
   $\delta(3, a) = 4,\ \delta(3, c) = 4,\ \delta(3, t) = 4$,
   $\delta(4, t) = 5$.
2. We create a nondeterministic generalized factor automaton from M_1 by inserting ε-transitions:

$\delta(0, \varepsilon) = \{0, 1, 2, 3, 4\}$.
The resulting automaton M_2 is depicted in Fig. 4.5.

3. We replace all ε–transitions in M_2 by non–ε–transitions. The resulting automaton is M_3.
   We remove $\delta(0, \varepsilon) = \{0, 1, 2, 3, 4\}$ and we add
   $\delta(0, a) = 2,\ \delta(0, c) = 2,$
   $\delta(0, c) = 3,$
   $\delta(0, a) = 4,\ \delta(0, c) = 4,\ \delta(0, t) = 4,$
   $\delta(0, t) = 5.$

4. Finally, we create the deterministic automaton M equivalent to M_3.
   The resulting automaton M is depicted in Fig. 4.6.

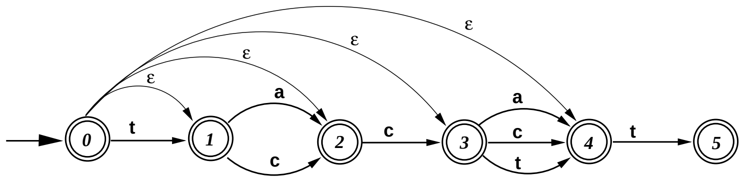


Figure 4.5: Transition diagram of nondeterministic generalized factor automaton M_2 with ε-transitions for $v = t[a, c]c[a, c, t]t$

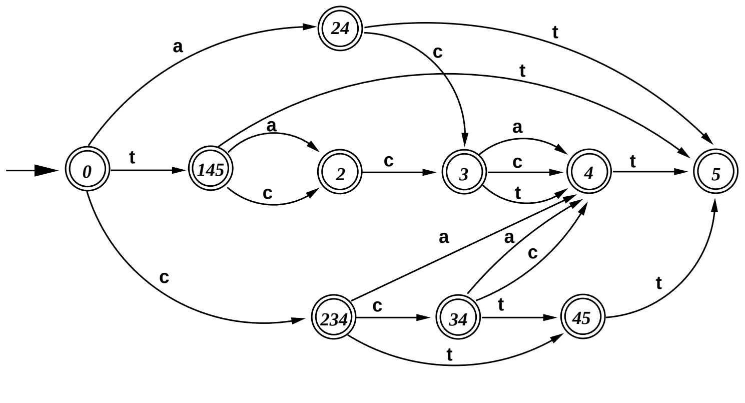


Figure 4.6: Transition diagram of deterministic generalized factor automaton M for $v = t[a, c]c[a, c, t]t$

4.3.2 Properties of the Generalized Factor Automaton

4.3.3 Size of the Generalized Factor Automaton.

Automaton M depicted in Fig. 4.5 accepts the same language as the tree-like automaton M_T

depicted in Fig. 4.7. The branches of automaton M_T are formed by nondeterministic factor automata $M_1^B, M_2^B, \dots, M_L^B$, where $L = 6$. The deterministic factor automaton has $2n-2$ states [5]. The size of the deterministic equivalent M_T' of the tree-like automaton M_T is in direct proportion with the sum of the sizes of the automata on its branches [9]. Therefore the number of states of M_T' is $\mathcal{O}(Ln)$. The number of branches of M_T is given by the number of all strings represented by the generalized string, in other words, by the number of all directed paths from the state q_0 to state q_n (without considering ε–transitions). This number is given by the product of the sizes of the symbol sets in all positions. Hence, in the worst case, it holds $L = |A|^n$ and the total size is $\mathcal{O}(|A|^n\, n)$, where n is the length of the string. But thanks to the fact that all the states q^i, $1 \le i \le L$ in M_T correspond to only one state q in M, all states of the form $\{q_1^{i_1} q_2^{i_2} \dots q_p^{i_p}\}$ in deterministic M_T will merge into one state $\{q_1 q_2 \dots q_p\}$ in deterministic M and thus the number of states of M will rapidly decrease with respect to the size of deterministic M_T. At this moment, no precise estimations of the upper bound of the number of states of *GFA* are available, so it remains $\mathcal{O}(2^n)$, which is given by the subset construction of deterministic automaton.

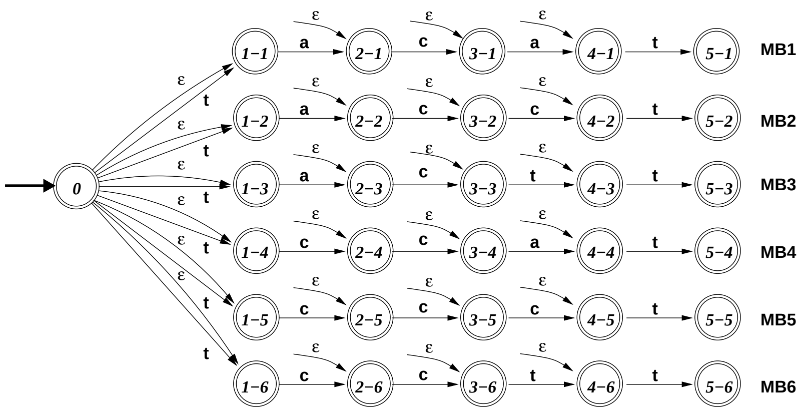


Figure 4.7: Nondeterministic tree-like automaton M_T for $v = t[a,c]c[a,c,t]t$

4.3.4 Experiments.

We performed a series of experiments to investigate the behaviour of the size of the generalized factor automaton with respect to the size of the nondeterministic generalized factor automaton in practice. The results of the experiments for various alphabets and string sizes show that the size of *GFA* tends not to grow exponentially even in the worst case. Both the maximum and the average size of *GFA* seem to be bounded from the top by the quadratic function, and from the bottom by the linear function with respect to the size of the input string, see Fig. 4.11.

4.3.5 Applications

We will show how to use the generalized factor automaton for searching substrings and for computing exact repetitions. In both cases, we adapted algorithms developed for the factor automaton. The basic algorithms process generalized strings, but we can also use them for processing weighted strings by adding preprocessing and postprocessing phases, as described above. *GFA* is an index structure, and therefore the construction is not part of the searching algorithms. Once *GFA* has been built, it is stored and, during the searching phases, it alone is being used. To construct the generalized factor automaton we use Alg. 4.3.1. During this construction, we memorise the subsets of the states of the nondeterministic generalized factor automaton corresponding to each state of the deterministic generalized factor automaton. Let us denote these subsets as *d-subsets* (deterministic subsets). We use these subsets in the algorithms that we present in the following. We will illustrate algorithms for the weighted string w, the generalized string v and *GFA* M, depicted in Fig. 4.1, 4.2 and 4.6, respectively. (Note that v is the basic string of w).

4.3.6 Searching substrings

Problem 4.17
Given a string (pattern) y. Find the end positions i_l of all its occurrences in the generalized string v, using the generalized factor automaton.

Solution 4.18
We start in the initial state. We perform a sequence of moves for symbols of searched pattern y. Elements of the d-subset of the state that we reach represent the end positions of pattern y in the text.

Example 4.19
Given a pattern $y = cc$. We use generalized factor automaton M from Example 4.3.1 (Fig. 4.6). We start in state 0 and we finish in state 34, which means that $y \in v[2..3]$ and $y \in v[3..4]$, in other words, y occurs in v at positions 3 and 4.

Problem 4.20
Given a string (pattern) y, probability function f_p (μ or α) and acceptability threshold d. Find the end positions i_l of all its occurrences, such that $f_p(y, i_l) \geq 1/d$, $1 \leq l \leq r$, for some r, in weighted string w using the generalized factor automaton.

Solution 4.21
The first phase of the solution is identical to Sol. 4.3.6. In the second phase, we check whether all found occurrences in positions i_l satisfy $f_p(y, i_l) \geq 1/d$.

Example 4.22
Given a pattern $y = cc$, probability function $f_p = \mu$ (multiplicative probability) and acceptability threshold $1/d = 0.3$. The first phase of the solution is given in Example 4.3.6. For the first occurrence ending at position $i_1 = 3$, it holds $f_p(y, 3) = 0.5$ hence the occurrence is accepted. For the second occurrence ending at position $i_2 = 4$, it holds $f_p(y, 4) = 0.25$, hence the occurrence is not accepted.

4.3.7 Searching exact repetitions

Problem 4.23
Find all maximum repeating factors of the generalized string v using the generalized factor automaton.

Solution 4.24
The repetitions that we are looking for are obtained by analysing d-subsets of *GFA* having cardinality greater than one. The d-subset of the state at depth h (length of the longest path from the initial state to the state) represents the occurrences (end positions) of maximum repeating factors having length h. The factors for a given state q are obtained by concatenating labels of transitions lying on the longest paths from the initial state to state q. In general, a state of *GFA* unlike *FA* can correspond to more than one longest factor.

Example 4.25
For instance, by analysing the state with label 45 which is at depth 3 we observe that the factor of length 3 (cct) is repeated at positions 4, 5 of v. Since $5 - 4 < 3$ the repetition is with overlapping. The remaining repetitions are summarized in a repetition table, see Tab. 4.2. (The symbols O, G, S used in the table have the following meaning: O - repetition with overlapping, S - square repetition, G - repetition with gap.)

Table 4.2: Repetition table from Example 4.3.7

d-subset	Factor	First occurrence	Repetitions
$1,4,5$	t	1	$(4,G),(5,G)$
$2,3,4$	c	2	$(3,S),(4,S)$
$2,4$	a	2	$(4,G)$
$3,4$	cc	3	$(4,O)$
$4,5$	cct	4	$(5,O)$

Problem 4.26
Given a probability function f_p (μ or α) and an acceptability threshold d. Find all repeating factors y_j ending at positions i_l, such that $f_p(y_j, i_l) \geq 1/d$, $1 \leq j \leq l \leq r$, for some r, in weighted string w using the generalized factor automaton.

Solution 4.27
The first step of the solution is identical to Sol. 4.3.7. Next, we split lines with more than one factor into more than one line, so that the new lines will contain only a single factor. The reason is that, in general, each factor has a different probability of appearance. In the third step, we compute f_p for the found repeating factors, see Tab. 4.3 (the value of f_p is the second element in parentheses).

In the next step, we remove all factors y_j for which $f_p(y_j, i_l) < 1/d$ from the repetition table. This step can cause, firstly, that some rows may contain no occurrence or single occurrence, secondly, the types of repetitions can change and, thirdly, the first occurrence of a factor can change. Therefore, finally, we have to remove all rows with only one occurrence from the repetition table, we have to change the type of repetition where needed, and we have to determine the new first occurrence where needed.

Example 4.28
Given a probability function $f_p = \mu$ (multiplicative probability) and acceptability threshold $1/d = 0.3$. The first step is identical to Example 4.3.7. In the second step, we compute values f_p for all factors in the repetition table Tab. 4.2, and by doing so we obtain Tab. 4.3. Next, we remove all occurrences of factors y_j at positions i_l with $f_p(y_j, i_l) < 0.3$ (marked by $*$), and finally we remove rows with no occurrences or single occurrences from the repetition table (marked by $\times$). The resulting repetition table is Tab. 4.4. We do not have to change either any type of repetition or any first occurrence.

Table 4.3: Auxiliary repetition table from Example 4.3.7

d-subset	Factor	First occurrence	Repetitions	
$1,4,5$	t	$(1,1)$	$(4,0.25,G)\ *, (5,1,G)$	
$2,3,4$	c	$(2,0.5)$	$(3,1,S), (4,0.25,S)\ *$	
$2,4$	a	$(2,0.5)$	$(4,0.5,G)$	
$3,4$	cc	$(3,0.5)$	$(4,0.25,O)\ *$	$\times$
$4,5$	cct	$(4,0.125)\ *$	$(5,0.25,O)\ *$	$\times$

Table 4.4: Repetition table from Example 4.3.7

d-subset	Factor	First occurrence	Repetitions
$1,4,5$	t	$(1,1)$	$(5,1,G)$
$2,3,4$	c	$(2,0.5)$	$(3,1,S)$
$2,4$	a	$(2,0.5)$	$(4,0.5,G)$

4.4 Pattern Matching in Generalized Strings

In this Section, we introduce a general method for constructing finite automata for on-line pattern matching in a generalized string.

Problem 4.29
Solve a given pattern matching problem P for a given string x (pattern) and a given generalized string v (text), both over alphabet A.

Problem 4.30
Solve a given pattern matching problem P for a given string x (pattern) and a given weighted string w (text), both over alphabet A, and a given probability function f_p (μ or α) and acceptability threshold d.

4.4.1 Method

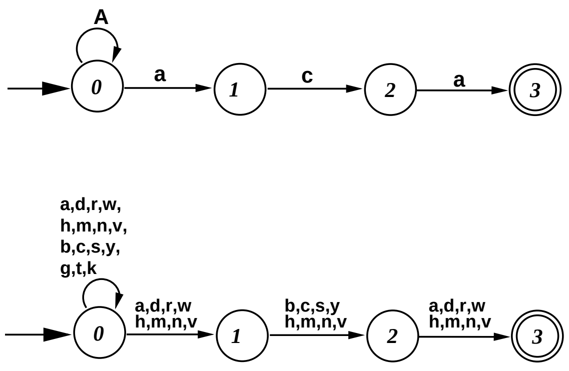


Figure 4.8: Comparison of nondeterministic finite automata M_{11} and M_{12} for exact pattern matching in strings over the basic alphabet A and the extended alphabet E_2 (from Example 4.2.2), respectively, for pattern $x = aca$

Solutions are known for pattern matching problems if the text is also a string. A nondeterministic finite automaton exists for each of the pattern matching problems solving it [9]. Our method transforms algorithm A for construction of automaton M for a given pattern matching problem P on strings, to algorithm A' for the construction of automaton M' for P on generalized strings. Generalized strings need to be in subset code for this application. The main idea is very simple and clear. Let $L(\mathsf{M}) \subseteq A^*$ be a language accepted by M, then M' will accept the language

$$L(\mathsf{M}') = \{x \mid \exists\ u : (u \in L(\mathsf{M}))\ \&\ (u \in PartFact_1(x)),\ u \in A^*,\ x \in \mathcal{P}(A)^*\},$$

Simply said, M' will accept all generalized strings containing strings defined by P as their simple elements.

Before we present the algorithm we must define the mapping which assigns to a symbol of the basic alphabet a set of symbols of the extended alphabet representing that symbol. Such a mapping is derived from a given subset code.

Definition 4.31 (Representation)
Let A and E be alphabets such that $A \subseteq E$ and $\mathcal{C}$ is a *subset code* $\mathcal{C} : \mathcal{P}(A) \longmapsto E$.
A *representation* for $\mathcal{C}$ is a total mapping $\mathcal{R} : A \longmapsto \mathcal{P}(E)$ such that:

$$\mathcal{R}(a) = \{e \mid e \in E\ \&\ a \in \mathcal{C}^{-1}(e)\ \},\ a \in A$$

We say that the members of $\mathcal{R}(a)$ are representatives of a.
We extend the definition of $\mathcal{R}$ for sets in the following way:

$$\mathcal{R}(S) = \bigcup_{a \in S} \mathcal{R}(a),$$

where $S \subseteq A$.

Table 4.5 represents the mapping $\mathcal{R}$ for subset code $\mathcal{C}_2$ from Tab. 4.1.

Algorithm 4.4.1
Transformation of the algorithm for constructing the pattern matching automaton.
Input: Algorithm A for constructing $\mathsf{M} = (Q, A, \delta, q_0, F)$ solving P on A^*, mapping $\mathcal{R}$.

Table 4.5: Mapping $\mathcal{R}$ for subset code $\mathcal{C}_2$ from Tab. 4.1

A	$\mathcal{R}(A)$
a	a, m, r, w, v, h, d, n
c	c, m, s, y, v, h, b, n
g	g, r, s, k, v, d, b, b
t	t, w, y, k, h, d, b, n

Output: Algorithm A' for constructing $\mathsf{M}' = (Q, \mathcal{R}(A), \delta', q_0, F)$ solving P on $\mathcal{P}(A)^*$.
Method:

1. Modify the instruction setting the input alphabet of M to set $\mathcal{R}(A)$ instead of A.
2. Substitute all instructions in the following way:

$$\delta(q_i, x) = q_j,\ \forall x \in S \implies \delta'(q_i, x) = q_j,\ \forall x \in \mathcal{R}(S)$$

$$\delta(q_i, x) = q_j,\ \forall x \in \overline{S} \implies \delta'(q_i, x) = q_j,\ \forall x \in (\ \mathcal{R}(A) \setminus \mathcal{R}(S)\ )$$

After transforming the algorithm, we create a nondeterministic finite automaton. We can create the deterministic equivalent to this automaton and use it for pattern matching, or we can skip determinization and directly simulate the nondeterministic automaton [8]. The latter approach is more suitable when we expect the deterministic automaton to have a large number of states.

If a given pattern matching problem can be decomposed into a pattern matching problem in a generalized string followed by a verification phase, we can use automaton M' also for pattern matching in weighted strings.

The preprocessings (extraction of the basic string) and postprocessings (verification) that we must add are meant to be performed on-line as the first and last steps of the searching procedures, in this case.

4.4.2 Applications

We will demonstrate our method in the solution of the exact pattern matching problem.

Problem 4.32
Transform the algorithm for exact pattern matching in strings [9] into an algorithm for exact pattern matching in generalized strings, for pattern $x = x_1x_2 \dots x_m$, $x \in A^*$. For transformation we use Alg. 4.4.1.

Solution 4.33
An automaton for exact pattern matching in strings can be built by the following algorithm.

A :

$M = (Q, A, \delta, q_0, F)$
$Q = \{q_0, q_1, q_2, \ldots, q_n\}$,
$\delta(q_{j-1}, x_j) = q_j, j = 1, 2, \ldots, n$,
$\delta(q_0, y) = q_0, \; y \in A$,
$F = \{q_n\}$.

We transform algorithm A into an algorithm for exact pattern matching in generalized strings using Alg. 4.4.1. In this way we obtain the following algorithm.

A′ :
$M = (Q, \mathcal{R}(A), \delta', q_0, F)$
$Q = \{q_0, q_1, q_2, \ldots, q_n\}$,
$\delta'(q_{j-1}, \mathcal{R}(x_j)) = q_j, j = 1, 2, \ldots, n$,
$\delta'(q_0, y) = q_0, \; y \in \mathcal{R}(A)$,
$F = \{q_n\}$.

Table 4.6: Transition tables of nondeterministic finite automaton M_{12} (the upper part) and deterministic finite automaton M_{13} (the lower part)

	a	c	g	t	m	r	w	s	y	k	v	h	d	r	n
$\rightarrow 0$	01	0	0	0	01	01	01	0	0	0	01	01	01	0	01
1		2			2			2	2		2	2		2	2
2	3				3	3	3				3	3	3		3
3															
$\rightarrow 0$	01	0	0	0	01	01	01	0	0	0	01	01	01	0	01
01	01	02	0	0	012	01	01	02	02	0	012	012	01	02	012
02	013	0	0	0	013	013	013	0	0	0	013	013	013	0	013
012	013	02	0	0	0123	013	013	02	02	0	0123	0123	013	02	0123
$\leftarrow 013$	01	02	0	0	012	01	01	02	02	0	012	012	01	02	012
$\leftarrow 0123$	013	02	0	0	0123	013	013	02	02	0	0123	0123	013	02	0123

Example 4.34
Let us have the pattern $x = aca$ and text u depicted in Fig. 4.9. The task is to find all exact occurrences of x in u.

First, we build a nondeterministic finite automaton using Alg. A′ for pattern x and the mapping $\mathcal{R}$ from Table 4.5. In this way we observe automaton M_{12} depicted in Fig. 4.8 and its transition table is Tab. 4.6. Next, we create deterministic finite automaton M_{13} by determinizing M_{12}. Automaton M_{13} is depicted in Fig. 4.10 and its transition table is Tab. 4.6. Finally, we perform matching using automaton M_{13}. The process of matching is shown in Tab. 4.7. We see that the pattern $x = aca$ was found in u at positions (end positions) 4, 6, 8 and 10.

Example 4.35
Let us consider the weighted string w from Fig. 4.9 as the text. The first step of the solution

is to extract its basic generalized string. The basic generalized string of w is u from Fig. 4.9. This is the same string as was used as the text in Example 4.4.2. Therefore we find the same occurrences. However, in this case we must check in addition a given probability condition and remove insufficient occurrences. Let us consider the probability function $f_p = \mu$ (multiplicative probability) and the acceptability threshold $1/d = 0.1$. For the occurrences listed above we get the probabilities: 0.125, 0.06, 0.0625 and 0.03125. Therefore only the occurrence in position 4 with probability 0.125 is accepted.

Table 4.7: Exact pattern matching using automaton M_{13}

	1	2	3	4	5	6	7	8	9	10	*position*
	$\begin{bmatrix}g\end{bmatrix}$	$\begin{bmatrix}a\\t\end{bmatrix}$	$\begin{bmatrix}c\\g\end{bmatrix}$	$\begin{bmatrix}a\\c\\t\end{bmatrix}$	$\begin{bmatrix}c\\t\end{bmatrix}$	$\begin{bmatrix}a\\g\end{bmatrix}$	$\begin{bmatrix}c\\t\end{bmatrix}$	$\boxed{\begin{matrix}a\\c\\g\\t\end{matrix}}$	$\begin{bmatrix}a\\c\\g\end{bmatrix}$	$\begin{bmatrix}a\\c\end{bmatrix}$	*text*
	g	w	s	h	y	r	y	n	v	m	*text in* E_2
0	0	01	02	$\underline{013}$	02	$\underline{013}$	02	$\underline{013}$	012	$\underline{0123}$	*states of* M_3
				0.125		0.06		0.0625		0.03125	*probabilities of occurrences*

$$w = \begin{bmatrix} (a,0.00) \\ (c,0.00) \\ (g,1.00) \\ (t,0.00) \end{bmatrix} \begin{bmatrix} (a,0.50) \\ (c,0.00) \\ (g,0.00) \\ (t,0.50) \end{bmatrix} \begin{bmatrix} (a,0.00) \\ (c,0.50) \\ (g,0.50) \\ (t,0.00) \end{bmatrix} \begin{bmatrix} (a,0.20) \\ (c,0.30) \\ (g,0.00) \\ (t,0.50) \end{bmatrix} \begin{bmatrix} (a,0.00) \\ (c,0.60) \\ (g,0.00) \\ (t,0.40) \end{bmatrix} \begin{bmatrix} (a,0.50) \\ (c,0.00) \\ (g,0.50) \\ (t,0.00) \end{bmatrix} \cdot$$

$$\cdot \begin{bmatrix} (a,0.00) \\ (c,0.50) \\ (g,0.00) \\ (t,0.50) \end{bmatrix} \begin{bmatrix} (a,0.25) \\ (c,0.25) \\ (g,0.25) \\ (t,0.25) \end{bmatrix} \begin{bmatrix} (a,0.50) \\ (c,0.25) \\ (g,0.25) \\ (t,0.00) \end{bmatrix} \begin{bmatrix} (a,0.50) \\ (c,0.50) \\ (g,0.00) \\ (t,0.00) \end{bmatrix}$$

$$u = \begin{bmatrix} g \end{bmatrix} \begin{bmatrix} a \\ t \end{bmatrix} \begin{bmatrix} c \\ g \end{bmatrix} \begin{bmatrix} a \\ c \\ t \end{bmatrix} \begin{bmatrix} c \\ t \end{bmatrix} \begin{bmatrix} a \\ g \end{bmatrix} \begin{bmatrix} c \\ t \end{bmatrix} \begin{bmatrix} a \\ c \\ c \\ t \end{bmatrix} \begin{bmatrix} a \\ c \\ g \end{bmatrix} \begin{bmatrix} a \\ c \end{bmatrix}$$

$$\mathcal{C}_2(u) = gwshyrynvm$$

Figure 4.9: Weighted string w, generalized string u in lists and in subset code representations; u is the basic string of w

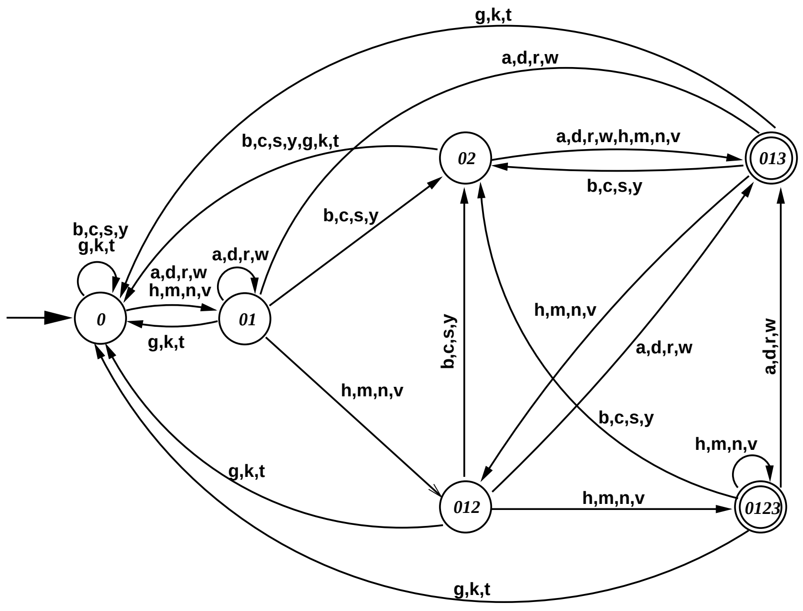
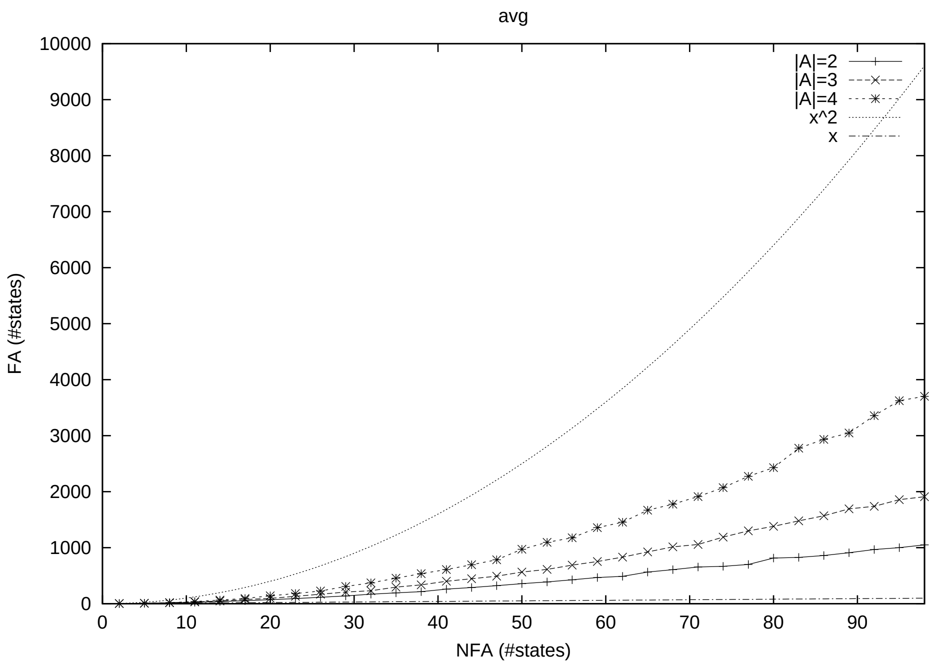


Figure 4.10: Deterministic finite automaton M_{13} for exact pattern matching in strings over extended alphabet E_2, for $x = aca$



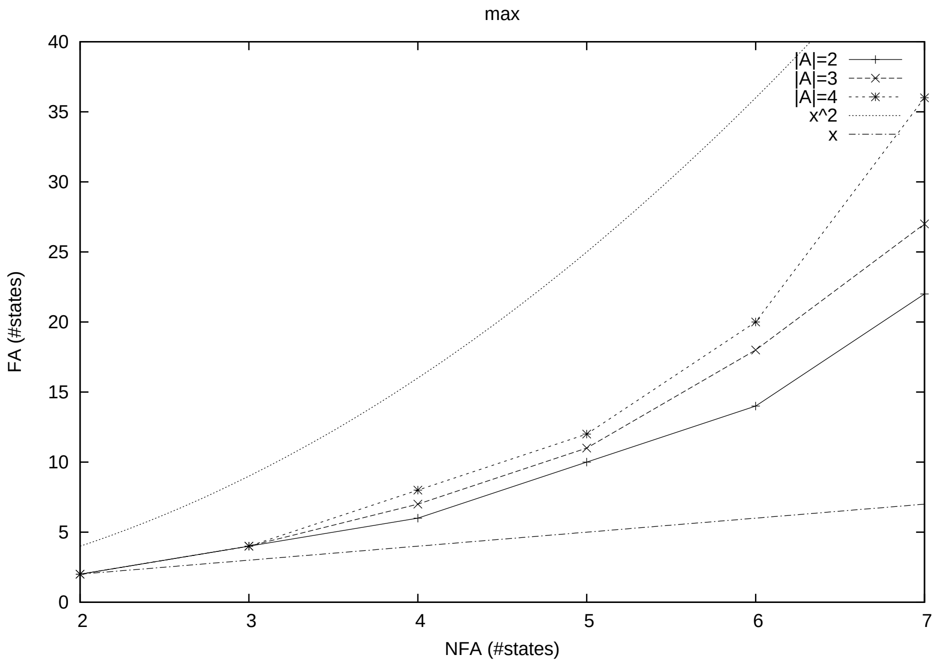


Figure 4.11: Average and maximum sizes of generalized factor automata for a generalized string - comparison with x, x^2, $|A| \in 2..4$

4.5 Conclusion

We have shown how to apply finite state automata in processing generalized strings and, as a consequence also weighted strings.

We have presented the algorithm for constructing the Generalized Factor Automaton, which is an automaton accepting all strings represented by a generalized string, and we have shown how to use it for searching repetitions and substrings in generalized and weighted strings.

Finally we have shown the method how for transforming the algorithm for constructing the finite automaton for pattern matching in strings into the algorithm for constructing a finite automaton solving the same pattern matching problem in generalized strings.

Our future work will be devoted to the development of algorithms solving other pattern matching problems in generalized strings and other problems concerning regularities in generalized strings, such as borders, covers, seeds, etc. based on GFA. The open problem is the space complexity of the Generalized Factor Automaton in relation to the size of the represented generalized string and the cardinality of the used alphabet.

References

[1] K. Abrahamson: **Generalized string matching**, SIAM Journal on Computing, 16(6), 1039-1051, 1987.

[2] A.L. Rosenberg: **On multi-head finite automata**, IBM J. Res. and Develop., 10, 388–394, 1966.

[3] R. Cole, R. Hariharan: **Tree Pattern Matching and Subset Matching in Randomized $O(n \log^3 m)$ Time**, Proceedings of the 29th ACM Symposium on the Theory of Computing,El Paso, TX, 66–75, 1997.

[4] J. Holub, C. S. Iliopoulos, B. Melichar, L. Mouchard: **Distributed string matching using finite automata**, Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms, 114-128, 1999.

[5] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M. T. Chen, J. Seiferas: **The smallest automaton recognizing the subwords of a text**. Theor. Comput. Sci., 40(1), 1985, 31–55.

[6] C. S. Iliopoulos, M. Mohamed, L. Mouchard, K. Perdikuri, W. F. Smyth, A. Tsakalidis: **String Regularities with Don't Cares**. Proceedings of the 2002 Prague Stringology Conference, Prague 2002, 65–74.

[7] C. S. Iliopoulos, L. Mouchard, K. Perdikuri, A. K. Tsakalidis: **Computing the Repetitions in a Weighted Sequence**. Proceedings of the 2003 Prague Stringology Conference, Prague 2003, 91–98.

[8] J. Holub: **Simulation of NFA in Approximate String and Sequence Matching**. Proceedings of the Prague Stringology Club Workshop '97, Prague 1997, 39–46.

[9] B. Melichar, J. Holub, T. Polcar: **Text searching algorithms, Tutorial for Athens course**, Manuscript, 2004.

[10] Prague Stringology Conference: `http://cs.felk.cvut.cz/psc/`

[11] `http://www.chem.qmul.ac.uk/iupac/`

5 Two-dimensional Pattern Matching Using the Finite Automata Jan Žďárek

5.1 Motivation

This chapter presents one of possible approaches to two-dimensional pattern matching, namely the finite automata approach.

The two-dimensional exact matching is a generalization of one-dimensional (string) matching. Suppose we have a *rectangular* digitized picture TA, where each point is given a number indicating, say, its color and brightness. We are also given a smaller rectangular picture PA, which also is digitized, and we want to find all occurrences (possibly overlapping) of the smaller picture in the larger one. We assume that the bottom edges of the two rectangles are parallel to each other. This is a two-dimensional generalization of the exact string matching problem that has been described in detail in the *Athens' Tutorial* [Ath–2004].

Motivation for finding a solution working in linear time (with respect to the size of the greater picture) is the fact, that trivial algorithm of two-dimensional exact pattern matching has $\mathcal{O}(|P||T|)$ asymptotic time complexity. Further discussion of this algorithm can be found in Section 5.3.

Organisation of this text is rather simple: at first, we shortly introduce elementary notions from the area of two-dimensional pattern matching. This work is closely bound to the *Athens' Tutorial*, therefore we do not recall notions from one-dimensional string matching here. It is recommended to a reader not familiar with them to have the *Athens' Tutorial* at hand.

At second, we show basic models of use of one-dimensional finite automata in two--dimensional pattern matching. Then we thoroughly describe one method of two-dimensional exact pattern matching based on one-dimensional pattern matching automata.

5.2 Selected notions of two-dimensional pattern matching

In the *Athens' Tutorial* [Ath–2004] reader has encountered numerous necessary definitions of the one-dimensional pattern matching area. In this text we will try to extend some of them into two-dimensional space. Moreover, whenever some newly defined notion has its one-dimensional equivalent, we are trying to formulate it as analogously as possible. From the one-dimensional case we also keep basic (though not formally defined) terms such are a *text* as a searched data and a *pattern* as a data, which we search for.

Definition 5.1 (Shape)
A *shape* is a geometrical formation in n-dimensional space, $n \in \mathbf{N}$.

Unless otherwise stated, we limit our interest to matching in two-dimensional space and therefore in following text we consider the rectangular shapes. Even in special cases, where we can encounter general geometrical shapes, we are able to determine some rectangular *bounding shape* of pattern and text array, respectively.

Remark: Despite of the fact that detailed discussion of such refinement of the presented problem is beyond scope of this text, reader should notice that 2D pattern matching methods presented below are after slight modification able to work even over irregular shapes of text and pattern array, respectively.

A *pattern array* (PA) and *text array* (TA) are in fact two-dimensional strings. In two--dimensional space we call them *pictures.* [GR–1997]

Definition 5.2 (Element of a picture)
An *element of a picture* or of a *two-dimensional string* P, denoted by $P[i,j]$, is a symbol over alphabet A, $P[i,j] \in A$.

Definition 5.3 (Picture (two-dimensional string))
A *picture* (*two-dimensional string*) over A is a rectangular array P of size $(n \times n')$ of symbols taken from a finite alphabet A. Formally, $\forall i,j,\ 1 \le i \le n,\ 1 \le j \le n';\ P[i,j] \in A$.

Definition 5.4 (Picture size)
The *size* of a picture is a size of its rectangular shape.

Definition 5.5 (Picture origin)
The *origin* of a picture is an element at position $(1,1)$.

The size of the figured picture is $(n \times n')$. Geometrical axes have their source at the upper left corner of the picture, in the same way as in certain applications of Computer Graphics (see Figure 5.1).
$\vec{x}-$axis therefore points from left to right,
$\vec{y}-$axis points from top to bottom of the array.

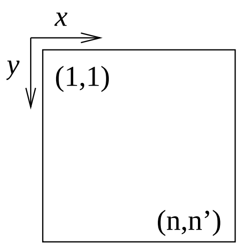


Figure 5.1: Orientation of coordinate system of two-dimensional pattern matching

Definition 5.6 (Empty picture)
The *empty picture* is a picture denoted by λ and its size is $|\lambda| = (0 \times 0)$. Pictures of size $(0 \times n)$ or $(n \times 0)$, where $n > 0$, are not defined.

Definition 5.7 (Set of all pictures)
The *set of all pictures* over alphabet A is denoted by A^{**}. (A two-dimensional language over A is thus a subset of A^{**}.)

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

Definition 5.8 (Subpicture)
Let $P \in A^{n \times n'}$ be a picture of size $(n \times n')$ and $R \in A^{m \times m'}$ be a picture of size $(m \times m')$. A picture R is a *subpicture* (a *block*, a *sub-array*) of picture P if and only if

$$m \le n,\ m' \le n' \text{ and } \exists k,l;\ k \le n-m,\ l \le n'-m'$$

such that

$$R[i,j] = P[i+k, j+l];\ \forall i,j,\ 1 \le i \le m,\ 1 \le j \le m'.$$

It means that the picture R is included in P (every element of R belongs also to P).

Remark: Although term *picture* was extensively used in previous definitions, it is equivalent with term *rectangular array.* (Except in context where it is substantial, the adjective "rectangular" will be omitted.)

In places, where it is important to distinguish between text and pattern, we usually prefer to use the term "*array*" instead of "*picture*". The reason is that it sounds better to say pattern or text array than pattern picture or text picture.

Definition 5.9 (Two-dimensional pattern matching problem)
A *two-dimensional pattern matching* problem is to locate an $(m \times m')$ pattern array PA (not necessarily a picture) inside an $(n \times n')$ (text) array TA (Figure 5.2).

Definition 5.10 (Two-dimensional occurrence)
A *two-dimensional occurrence* of pattern array PA in text array TA is

- exact, when PA is included in TA as a sub-array,
- approximate (with k errors), when PA or its bounding picture is included in TA as a sub-array with k not-matching positions (errors).

The position of a *two-dimensional occurrence* of PA in TA is a pair (i, j), such that $PA = TA[i, \dots, i+m-1;\ j, \dots, j+m'-1]$, as in Figure 5.2. Note that $PA[1,1] = TA[i,j]$ and therefore the lower right corner of PA in Figure 5.2 is at position $(i+m-1, j+m'-1)$ in TA. When there is not risk of ambiguity, we will refer to two-dimensional occurrence simply as an occurrence.

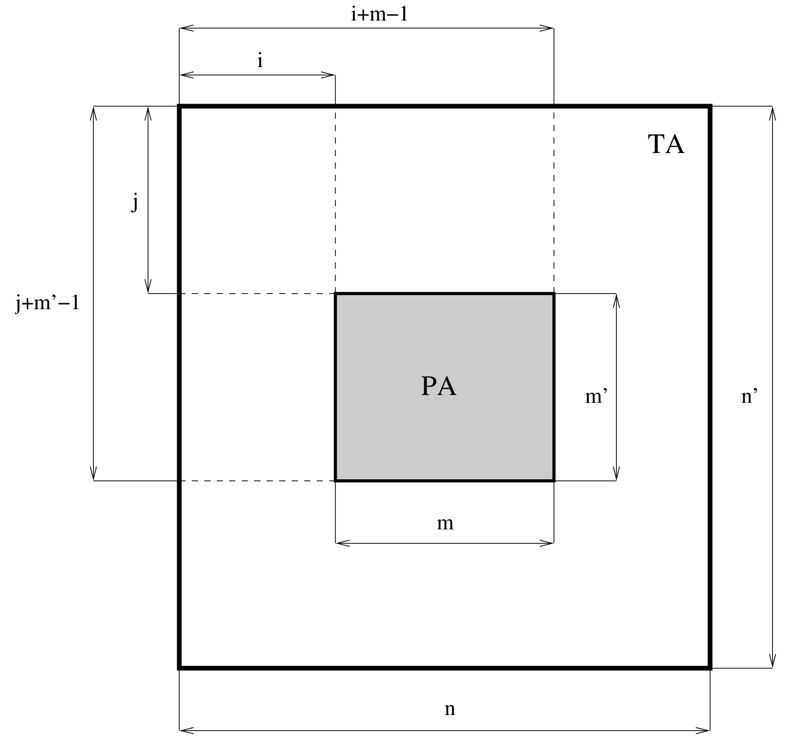


Figure 5.2: Pattern array PA occurs at position (i, j) in text array TA

5.3 Trivial algorithm for two-dimensional exact matching

At the beginning of this chapter we mentioned so called *trivial algorithm* for the exact pattern matching in two dimensions. Motivation for finding two-dimensional pattern matching algorithms working in linear time (linear with respect to the size of the text array) is the fact that the trivial algorithm has $\mathcal{O}(|TA||PA|) = \mathcal{O}(nn' \cdot mm')$ asymptotic time complexity.

The trivial algorithm for the two-dimensional exact pattern matching is analogical to the one in one-dimensional case: we are given above-mentioned 2D arrays PA and TA. Assume their sizes are $(m \times m')$ and $(n \times n')$, respectively. Without loss of generality let hold $m \geq m'$ and $n \geq n'$.

The trivial algorithm repeatedly verifies for each element of the text array whether there begins an occurrence of the pattern array or not. Asymptotical time complexity of this single check is $\mathcal{O}(|PA|) = \mathcal{O}(m^2)$ and it is done $\mathcal{O}(|TA|) = \mathcal{O}(n^2)$ times. Therefore overall asymptotical time complexity of such algorithm is

$$\mathcal{O}(|TA||PA|) = \mathcal{O}(n^2 \cdot m^2)\,.$$

We clearly see that there cannot be an occurrence of PA in elements of TA such that $TA[i, j]$, $i > n - m + 1$, and $j > n' - m' + 1$, but asymptotically it does not matter if we make such an enhancement and do not check an occurrence there or not. Because in this algorithm we are interested in the exact occurrences, we also see that checking for an occurrence will end in the moment the first non-matching element is found, thus *usually* it will end prematurely.

However, consider following situation: let PA consists of all zeros (0) and in the bottom right element is single one (1). Let TA consists of all zeros (0). At this moment, we can end our deliberation stating that this is the worst case, where every element of the pattern array PA inevitably must be compared with appropriate elements of the text array TA, and even in case we are interested in the first occurrence of PA in TA only, we have to check elements of whole text array.

Attentive reader has noticed that 2D trivial algorithm is analogous to the 1D case (cf. [Ath–2004], Section 2.1). In practice the first mismatch between element of PA and TA occurs usually very soon. But this analogy is limited though, because we do not have any *natural language* in terms of pictures, therefore we cannot simply find any constant C_L and say that for this kind of pictures the 2D pattern matching using the trivial algorithm is linear.

5.4 General models of two-dimensional pattern matching using finite automata

Reusing of the finite automata from one-dimensional pattern matching into two-dimensional pattern matching has some advantages: the same formal method of modelling of pattern matching algorithms in both cases and description of all problems (one- and more- dimensional) using a unified view.

Melichar and Holub [MH–1997] showed, that all one-dimensional pattern matching problems are sequential problems and therefore it is possible to solve them using finite automata.

Of course, we cannot use one-dimensional automata directly, because two or more dimensional problems are inherently not sequential. Therefore our aim is to find a way of appropriate transformation of nonsequential problems to sequential problems.

In general we can

- use multiple automata passing their results among them, or
- modify run of pattern matching automaton to process text and pattern array in somehow "linearized" form.

Before we turn our attention to the first of these two approaches, let us briefly explain the latter one. This method is based on the following observation: we can treat whole image as one one-dimensional string using a path in a picture defined along some space filling curve. The simplest curve is the *C-curve* (the compound curve). It starts at the picture origin and goes to the end of the first row, then continues from the beginning of the next row until it reaches the last element of the picture. Having such curve, we can apply for example *SFOECS* automaton on it and proceed as usual in one-dimensional pattern matching. *SFOECS* automaton ([Ath–2004], Figure 4.18) is *NFA* for exact string matching of sequence of strings – rows of the pattern array.

Using the C-curve we can imagine that our automaton "reads" the text in the same manner we usually do: row-wise from left to right. Such linearization is very simple, on the other hand we are only able to find fragments (rows) of the original pattern array that may be far away from each other and even in the same row of the text array (i.e. for $n > 2m$).

Considering its disadvantages we will not discuss this approach further in this text.

Our general model of two-dimensional pattern matching based on the first one of those approaches (i.e. passing of results) consists of two processing steps and involves two types of pattern matching automata:

1. (Nondeterministic) finite automaton preprocessing text array TA as a set of columns (or rows), which produce text array TA' of the same shape as TA.
2. (Nondeterministic) finite automaton searching for a *representing string* in rows (or columns, it depends on direction selected in the previous step) of TA', occurrence of this string determine position of the 2D occurrence of PA in TA.

5.5 Two-dimensional exact pattern matching

Two-dimensional exact pattern matching is at present time relatively well explored area. The method presented hereinafter follows the basic approach found by Bird and, independently, by Baker [Bir–1977, Bak–1978]. Since then, many additional methods have been presented, however they are often quite complicated.

Bird and Baker in their algorithm employed algorithms simulating nondeterministic pattern matching automata, namely the Aho-Corasick and the Knuth-Morris-Pratt pattern matching machines ([Ath–2004], Chapter 5 – *Simulation of nondeterministic pattern matching automata*). Instead of these algorithms we will use determinized versions of selected automata described in the second chapter of the *Athens' Tutorial*.

Let the pattern array PA be viewed as a sequence of strings. Without loss of generality let these strings be its columns. To locate columns of the pattern array within columns of the text array requires searching for several strings, see the idea in Figure 5.3 or Figure 5.2.

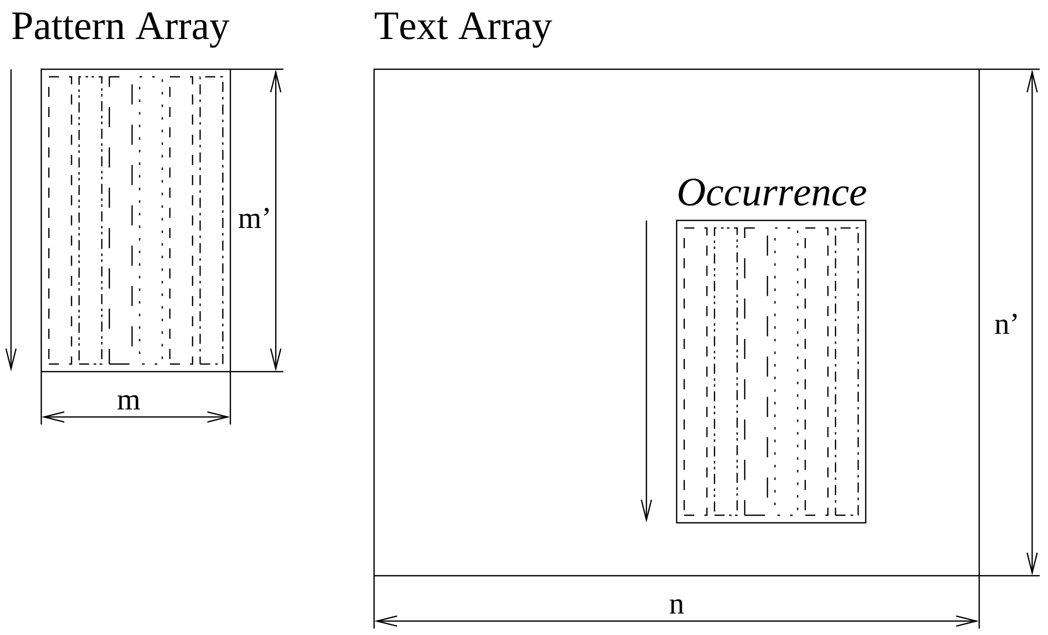


Figure 5.3: Pattern matching searching for columns of the pattern array PA in the text array TA

Moreover, the *two-dimensional occurrence* of the pattern PA must be found in a particular configuration within rows. All columns of the pattern are to be found in the order specified by the order of strings in the pattern, and all ending on the same row of the text array.

The strategy of searching for PA in the text array TA is as follows. Let PS be the set of all (distinct) columns of PA, treated as individual strings. First we build the string-matching automaton $M(PS)$ with final states. Each final state of this automaton corresponds to some pattern in the set PS. Therefore, each column of the pattern array PA is unambiguously identified with corresponding final state of the automaton $M(PS)$. However, there can be less than m final states owing to possible identities between some columns (Figure 5.3). Once the automaton $M(PS)$ is constructed, it is applied to each column of TA. We generate an array, say TA', of the same size as TA, and whose elements are determined by the run of $M(PS)$.

The pattern array PA itself is replaced by a *string* P over the set of final states: the i^{th} symbol of P is the final state identified with the i^{th} column of PA. The rest of the entire process consists of locating P inside the rows of TA', reporting eventual occurrences of P and therewith also PA.

5.5.1 Construction of *NFA* for the exact matching of a set of strings

Let m be the number of columns and m' be the number of rows of pattern array PA, m_d be the number of distinct columns of PA, $m_d \leq m$. Construction of automaton $M(PS)$, $M(PS) = (Q, A, \delta, q_0, F)$, for searching for the set PS, $PS = P_1, P_2, \ldots, P_{m_d}$, consists of m_d nondeterministic finite automata for the exact pattern matching, $M_1, M_2, \ldots, M_{m_d}$. Careful reader of the *Athens' Tutorial* already knows this kind of automata as the *SFOECO* automata

([Ath–2004], Figure 2.2) and their construction algorithm ([Ath–2004], Algorithm 2.1).

Our nondeterministic finite automaton $M(PS)$, also known as the *SFFECO* automaton ([Ath–2004], Figure 2.13), and its construction algorithm ([Ath–2004], Algorithm 2.9) is based on the union of *SFOECO sub-NFA*s for each pattern from the set PS. The Algorithm 5.1 is a variant of above mentioned *SFFECO* construction algorithm customized for our purpose.

The resulting automaton $M(PS)$ with the maximum number of its states is in Figure 5.4.

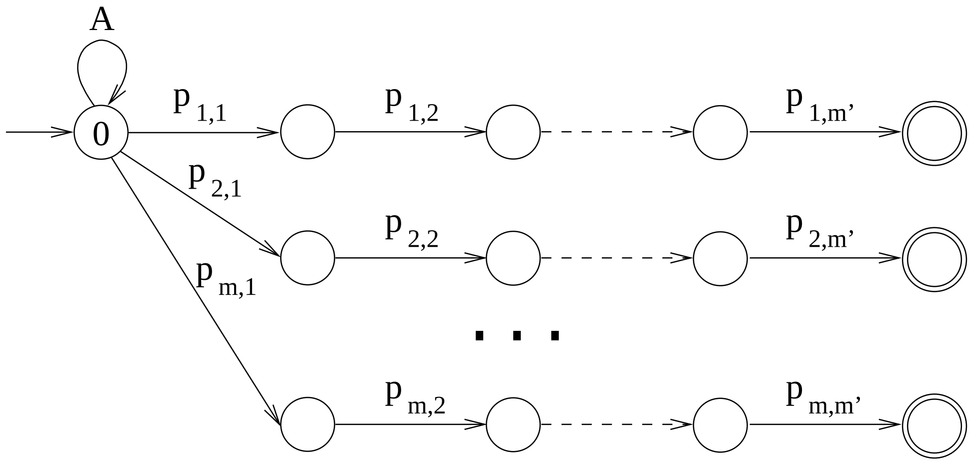


Figure 5.4: *NFA* for searching for a set of strings PS, $m_d = m$

Lemma 5.11
A non-minimized nondeterministic finite automaton for the exact string matching of set of m_d patterns each of length m', constructed by Algorithm 5.1, has at most $m_d m' + 1$ states. The maximum of states of such *NFA* (the size of a maximal *trie* over the set PS) is $mm' + 1$ and $mm' + 1 \geq m_d m' + 1$.

Proof
We have built m_d *NFA*s for the exact pattern matching, constructed by *SFOECO* automaton construction algorithm ([Ath–2004], Algorithm 2.1) and we made a union of them. From the same algorithm we already know, that each of these *NFA*s has $m' + 1$ states. Due to the operation union all initial states are merged into q_0, to the initial state of the first of these *sub-NFA*s. Moreover, all states representing common prefixes of particular strings are merged. The worst case takes place if each string starts with different symbol than the others, i.e. there is no common prefix of two or more strings. Therefore the number of states of constructed *NFA* $M(PS)$ is at most $m_d m' + 1$.

Moreover the number of distinct strings in the set PS cannot be greater than the number of columns of the pattern array PA, $m \geq m_d$, thus inequality $mm' + 1 \geq m_d m' + 1$ holds. □

To be able to preprocess the text array TA the determinization of the automaton $M(PS)$ is required. We can obtain deterministic finite automaton from the *NFA* in several ways. One of them is an algorithm from [Ath–2004], Section 2.3 – *Deterministic pattern matching automata.* Using this algorithm we are able to determinize the *NFA* $M(PS)$ in time $\Theta(|A||Q|)$. Such plausible time complexity we can reach owing to the special characteristics of the automaton

Algorithm 5.1: Construction of *NFA* for the exact matching of the set of strings PS
Input: Set PS of (distinct) strings of pattern array PA (its columns or rows). Cardinality of the set PS is $|PS| = m$.
Output: *NFA* M accepting language $L(M)$, $L(M) = \{wP_i \mid w \in A^*, 1 \leq i \leq m,\ P_i \in PS\}$.
Description: Note that the first part of this algorithm does not construct exactly the *SFOECO* automata, because they lack their, for our purpose useless, self loops of the initial states.
Method:
NFA $M = (Q, A, \delta, q_0, F)$, where Q, F and the mapping δ are constructed in the following way:

```
$m = |PS|$
$m' = |P_1|$                                              { $|P_1| = |P_2| = \cdots = |P_m|$ }
$Q = \{q_0, q_1, \ldots, q_{m(m'+1)}\}$
for $i = 1$ to $m$ do
    for $j = 0$ to $m' - 1$ do
        $\delta(q_{(i-1)(m'+1)+j}, p_{i,j+1}) = \{q_{(i-1)(m'+1)+j+1}\}$     { forward transitions for $P_i[j+1]$ }
    end
    $\delta(q_{(i-1)(m'+1)+j}, a) = \emptyset, \forall a \in A$          { no transition leading from the final state }
end
for $i = 1$ to $m - 1$ do
    $\delta(q_{(i-1)(m'+1)}, \varepsilon) = \{q_{i(m'+1)}\}$                  { constructs $\varepsilon$-transitions }
end
Create the union of these $m$ sub-NFAs.
$\delta(q_0, a) = \delta(q_0, a) \cup \{q_0\}, \forall a \in A$                { self loop of the initial state }
```

$M(PS)$, for detailed discussion see [Ath–2004], Section 2.4 – *The state complexity of the deterministic pattern matching automata*, especially Theorems 2.28 and 2.29.

Since *DFA* $M_D(PS)$ accepts the same language as *NFA* $M(PS)$, $L(M_D(PS)) = L(M(PS))$, these automata are equivalent.

5.5.2 Processing of image with *DFA* for the exact matching of a set of strings

Suppose we have an array TA' of the same size as TA. Let *DFA* $M(PS)$, $M(PS) = (Q, A, \delta, q_0, F)$, denotes determinized version of *NFA* constructed by Algorithm 5.1. Let F be the set of all final states of $M(PS)$, the number of final states be denoted by m_d and m be the number of columns of the pattern array PA.

Preprocessing of the text array TA consists of finding occurrences of strings from the set PS for all elements of PA.

Firstly we have to assign an output action to every state of the *DFA* $M_D(PS)$. When state q, $q \in Q$, becomes active, its associated output action puts its identification into array TA' on the active position (position that has been read in the actual step).

Definition of *DFA* guarantees, that at most one such output action can be active in each step and that after processing of TA by *DFA* $M_D(PS)$ all elements of TA' are properly set.

After application of $M(PS)$ on every column of TA we obtain the text array TA' prepared for application of the one-dimensional deterministic finite automaton for the exact pattern matching. *NFA* for the exact patern matching of one pattern has been mentioned above as the *SFOECO* automaton.

Usage of the new text array TA' is not optimal, and we have to avoid it. Our preprocessing automaton $M(PS)$ can work over the original array TA, replacing every symbol read with the appropriate state identifier q, $q \in Q$, thus eliminating need for any extra space, except for the automata themselves. This is very important, because TA is expected to be large.

Formally, for $M_D(PS)$, $M_D(PS) = (Q, A, \delta, q_0, F)$, $q \in Q$, q is the active state after reading symbol from the element $TA[i, j]$, holds

$$TA'[i,j] = \delta(q, TA[i,j]) \text{ for } \forall i, j \;\; 1 \le i \le n, \; 1 \le j \le n'$$

5.5.3 Searching for two-dimensional exact occurrences

We have to construct *DFA*, say M'_D, as the determinized version of *SFOECO* automaton over the alphabet Q of all state identifiers of the automaton $M_D(PS)$ searching for a string consisting of properly ordered final state identifiers of $M_D(PS)$ (or even $M(PS)$, because selected determinization method leaves final states' identifiers of $M(PS)$ intact).

More precisely, let determinized $M(PS)$ be $M_D(PS)$, $M_D(PS) = (Q_D, A, \delta_D, q_{0D}, F)$. Let F, $F = \{f_1, f_2, \dots, f_{m_d}\}$, $|F| = m_d$, be the set of all final state identifiers of the automaton $M(PS)$. Then it is possible to construct deterministic finite automaton $M'_D = \{Q'_D, Q_D, \delta'_D, q'_{0D}, F'_D\}$ as a determinized version of the *SFOECO* automaton that searches for the pattern P of length m in TA', see Figure 5.5.

Pattern P (the *representing string*) is created by Algorithm 5.2.

Algorithm 5.2: Construction of pattern P representing the pattern array PA for the exact pattern matching
Input: Correspondence between strings from the set PS and final states of automaton $M(PS)$, $M(PS) = (Q, A, \delta, q_0, F)$. Let pattern P be $P = \{p_1 p_2 \cdots p_m\}$ and $|PS| = m_d$.
Output: Pattern P of length m.
Method:

```
for $i = 1$ to $m$ do
    $p_i = f_j$, $f_j \in F$, such that column $PA[i]$ corresponds to the final state $f_j$ of $M(PS)$,
    $1 \le j \le m_d$
end
```

When deterministic finite automaton M'_D, $M'_D = \{Q'_D, Q_D, \delta'_D, q'_{0D}, F'_D\}$, is constructed, it is applied on each row (or column, depending on the direction that has been selected for the preprocessing step) of TA' and eventually will find occurrences of the representing string P within rows of TA'. Consequently, it will find 2D occurrences of PA in TA, because position of the element of TA' where 1D occurrence of P is eventually reported, is also position of the lower right element of 2D occurrence of PA in TA. Knowing position of this element it is easy to compute the origin of the 2D occurrence. Let (x, y) be the actual position, where an occurrence has been reported, then position of the origin is $(x - m + 1, y - m' + 1)$.

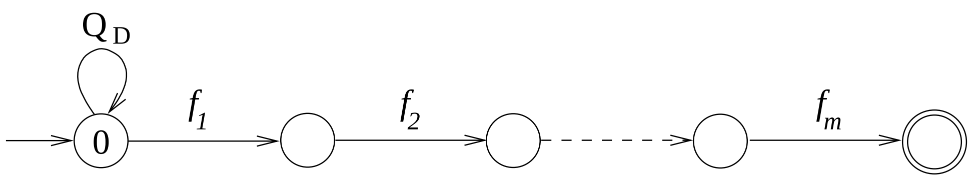


Figure 5.5: *NFA* for the exact pattern matching of string P, $|P| = m$, consisting of the final states of *DFA* $M_D(PS)$ over finite alphabet Q_D

5.6 Results

Presented method of two-dimensional exact pattern matching yields the subsequent results.

Theorem 5.12
Let A be finite alphabet, TA be the text array of size $(n \times n') = N_1$, $TA \in A^{n \times n'}$, PA be the pattern array of size $(m \times m') = N_2$, $PA \in A^{m \times m'}$, m_d be the number of distinct columns of PA, $m_d \leq m$, and let PS be set of these columns. Moreover, suppose that condition $N_2 \leq N_1$ holds. Then two-dimensional exact pattern matching using the finite automata can be done with $\mathcal{O}(N_1)$ asymptotic time complexity.

Proof
The construction of the (nondeterministic) *SFFECO* pattern matching automaton $M(PS)$ takes time $\mathcal{O}(N_2)$, according to Algorithm 5.1. Its determinization takes time $\mathcal{O}(|A||Q|m) = \mathcal{O}(|A||N_2|m)$, because the number of states of *NFA* $M(PS)$ is $Q = m_d m' + 1 = \mathcal{O}(N_2)$ (Lemma 5.11). Informally speaking, the determinization works with such time complexity in our case, because $M(PS)$ is acyclic (tree-like) automaton with only one self loop at its initial state. Interested reader may consult ([Ath–2004] Section 2.4) for details. However, there exist an algorithm constructing automaton simulating our model that builds the AC-automaton in time $\mathcal{O}(N_2)$.

The construction of the array of state identifiers TA', i.e. processing TA by determinized automaton $M(PS)$, clearly takes $\Theta(N_1)$ time.

String P representing pattern array PA can be created in $\Theta(m)$ time, where length of P is m. This operation and construction of corresponding (nondeterministic) *SFOECO* automaton require $\Theta(m)$ time. Determinization then takes analogously with the previous step $\mathcal{O}(|A|m)$ time, using the following refinement: let X be $X = Q \setminus F$ (all nonfinal states of $M_D(PS)$), then we can treat all nonfinal state identifiers as one special "identifier" X, so there is only $\mathcal{O}(m+1) = \mathcal{O}(m)$ symbols of the alphabet. Hence we are able to determinize automaton M' in the desired $\mathcal{O}(|A|m)$ time.

The final search phase consists of pattern matching of string P inside rows of TA'. This is done using determinized *SFOECO* pattern matching automaton for string P and it takes time $\Theta(N_1)$.

Total asymptotic time complexity of two-dimensional exact pattern matching for fixed alphabet is then

$$\mathcal{O}(N_2) + \Theta(N_1) + \mathcal{O}(m) + \Theta(N_1) \ = \ \mathcal{O}(2N_1 + N_2 + m) \ = \ \mathcal{O}(N_1 + N_2).$$

Let us suppose, that condition $N_2 \leq N_1$ holds. Then our proof can be completed stating that

$$\mathcal{O}(N_1 + N_2) \ = \ \mathcal{O}(N_1)$$

□

Theorem 5.13
Let presumptions from the previous Theorem 5.12 hold.
Then asymptotic space complexity of two-dimensional exact pattern matching using the finite automata is $\mathcal{O}(\max\{N_2, m^2\})$.

Proof
Two-dimensional exact pattern matching needs two different automata.

1. In the first step it is determinized *SFFECO* pattern matching automaton with $\mathcal{O}(|A|N_2)$ space complexity.

2. In the second string P is created, representing pattern array PA with $\Theta(m)$ space complexity, and

3. determinized *SFOECO* pattern matching automaton with $\mathcal{O}((m+1)m) = \mathcal{O}(m^2)$ space complexity, using the observation that it is possible to replace all nonfinal state identifiers by one special identifier (see the explanation in proof of Theorem 5.12).

Therefore asymptotic space complexity of two-dimensional exact pattern matching is the maximum from these space requirements: $\mathcal{O}(\max\{|A|N_2, m^2\})$.
Assuming fixed alphabet it is $\mathcal{O}(\max\{N_2, m^2\})$. □

5.7 Example

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

$$PA = \begin{array}{|c|c|c|}\hline a & c & a \\ \hline b & b & a \\ \hline c & a & b \\ \hline \end{array}\ , \ TA = \begin{array}{|c|c|c|c|c|c|c|}\hline b & a & c & a & b & a & c \\ \hline c & b & b & a & a & a & c \\ \hline b & b & a & b & b & a & b \\ \hline a & a & c & a & c & b & a \\ \hline b & b & b & a & c & a & c \\ \hline a & c & a & b & b & a & b \\ \hline c & a & a & c & a & b & a \\ \hline b & b & b & b & a & c & c \\ \hline a & c & c & a & b & a & b \\ \hline \end{array}\ , \ |PA| = (3 \times 3),\ |TA| = (7 \times 9).$$

Then set PS is the set of three strings over finite alphabet $A = \{a, b, c\}$, $PS = \{abc$, cba, $aab\}$. *NFA* $M(PS) = (Q, A, \delta, q_0, F)$, constructed by Algorithm 5.1, is in Figure 5.6. An equivalent deterministic finite automaton to the automaton $M(PS)$ can be constructed using determinisation algorithm from [Ath–2004].

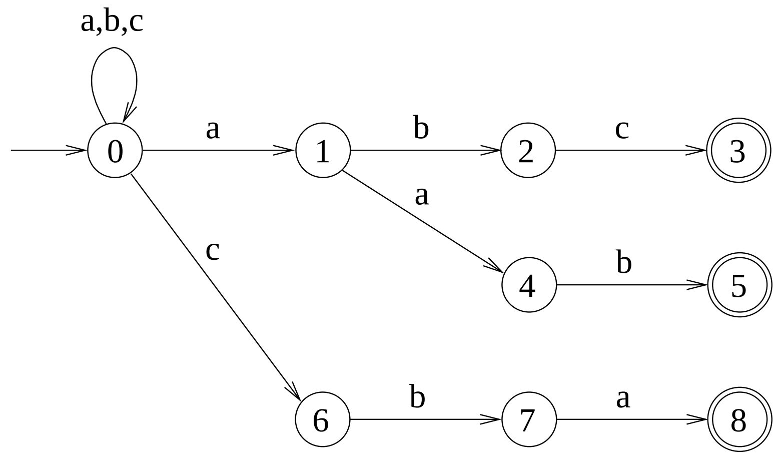


Figure 5.6: Minimized nondeterministic finite automaton for the exact matching of set of patterns PS, $PS = \{abc, cba, aab\}$, is a trie of PS

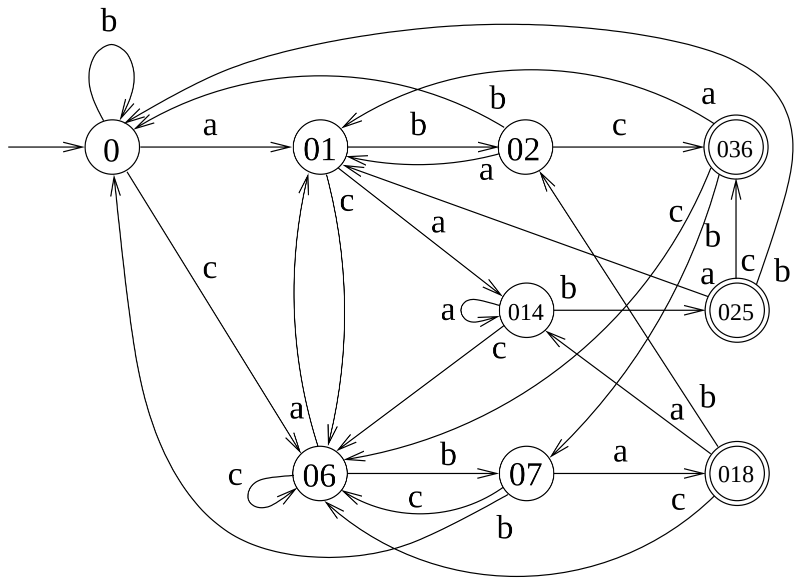


Figure 5.7: Deterministic version of the finite automaton from Figure 5.6 for the exact matching of set of patterns PS

Figure 5.7 shows the result of determinization. Notice that the number of states of *DFA* did not change compared to its source *NFA*. It suggests that *NFA* has already been minimal.

Table 5.1: Determinisation of *NFA* $M(PS)$

	Q	a	b	c		Q	a	b	c
$\rightarrow$	0	0,1	0	0,6	$\rightarrow$	0	01	0	06
	1	4	2	–		01	014	02	06
	2	–	–	3		02	01	0	036
$\leftarrow$	3	–	–	–		06	01	07	06
	4	–	5	–		07	018	0	06
$\leftarrow$	5	–	–	–		014	014	025	06
	6	–	7	–	$\leftarrow$	018	014	02	06
	7	8	–	–	$\leftarrow$	025	01	0	036
$\leftarrow$	8	–	–	–	$\leftarrow$	036	01	07	06

Table 5.2: TA and TA' as a result of processing columns of TA by *DFA* from Figure 5.7

b	a	c	a	b	a	c	0	01	06	01	0	01	06
c	b	b	a	a	a	c	06	02	07	014	01	014	06
b	b	a	b	b	a	b	07	0	018	025	02	014	07
a	a	c	a	c	b	a	018	01	06	01	036	025	018
b	b	b	a	c	a	c	02	02	07	014	06	01	06
a	c	a	b	b	a	b	01	036	018	025	07	014	07
c	a	a	c	a	b	a	06	01	014	036	018	025	018
b	b	b	b	a	c	c	07	02	025	07	014	036	06
a	c	c	a	b	a	b	018	036	036	018	025	01	07

Table 5.3: Result of matching for P in rows of TA' by *DFA* from Figure 5.9 and found 2D occurrences of PA in TA

							1	1	1	1	1	1	1
							1	1	1	1	1	1	1
							1	1	1	1	1	1	1
	a	*c*	*a*				1	1	1	1	2	1	1
	b	*b*	**a**	**c**	**a**		1	1	1	1	1	1	1
	c	*a*	**b**	**b**	**a**		1	*2*	*3*	*4*	1	1	1
		a	**c**	**a**	**b**		1	1	1	**2**	**3**	**4**	1
		b	b	a			1	1	1	1	1	2	1
		c	a	b			1	2	*2*	*3*	*4*	1	1

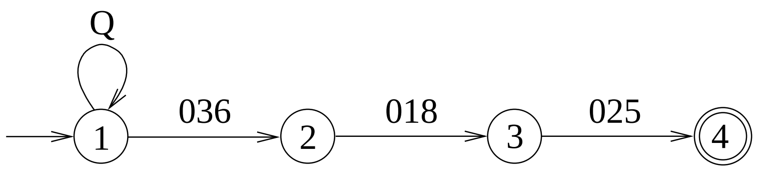


Figure 5.8: Nondeterministic *SFOECO* automaton for exact matching of the representing string P

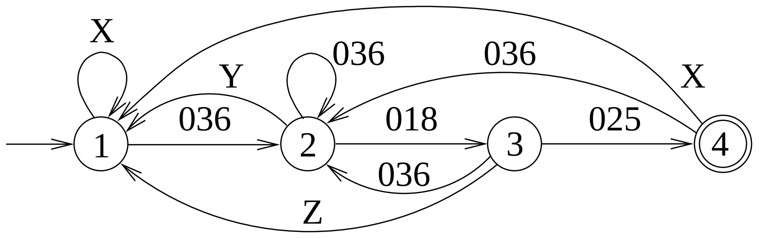


Figure 5.9: Determinized finite automaton from Figure 5.8, $X = Q\backslash\{036\}$, $Y = Q\backslash\{018, 036\}$, $Z = Q \setminus \{025, 036\}$

References

[Bak–1978] Baker, T.P.: A technique for extending rapid exact-match string matching to arrays of more than one dimension, in: SIAM Journal on Computing, Vol. 7, No. 4 (November 1978), pp. 533–541

[Bir–1977] Bird, R.S.: Two-dimensional pattern matching, in: Information Processing Letters, Vol. 6, No. 5 (October 1977), pp. 168–170

[GR–1997] Giammarresi, D. and Restivo, A.: Two-dimensional languages, in: Handbook of Formal Languages, Vol. III Beyond Words, Springer-Verlag, Heidelberg (1997), pp. 216–267

[MH–1997] Melichar, B. and Holub, J.: 6D classification of pattern matching problems, in: Proceedings of the Prague Stringologic Club Workshop '97, Collaborative Report DC-97-03, editor: Holub, J., Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, Prague (1997), pp. 24–32

[Ath–2004] Melichar, B., Holub, J. and Polcar, T.: Text searching algorithms, Tutorial for Athens course, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, Prague (2004)