Kybernetika

Jan Holub

The finite automata approaches in stringology

Kybernetika, Vol. 48 (2012), No. 3, 386--401

Persistent URL: http://dml.cz/dmlcz/142945

KYBERNETIKA — VOLUME *48* (2012), NUMBER 3, PAGES 386–401

THE FINITE AUTOMATA APPROACHES IN STRINGOLOGY

Jan Holub

We present an overview of four approaches of the finite automata use in stringology: deterministic finite automaton, deterministic simulation of nondeterministic finite automaton, finite automaton as a model of computation, and compositions of finite automata solutions. We also show how the finite automata can process strings build over more complex alphabet than just single symbols (degenerate symbols, strings, variables).

Keywords: exact pattern matching, approximate pattern matching, finite automata, dynamic programming, bitwise parallelism, suffix automaton, border array, degenerate symbol

Classification: 93E12, 62A10

1. INTRODUCTION

Many tasks in Computer Science can be solved using finite automata. The finite automata theory is well developed formal system that is used in various areas. The original formal study of finite state systems (neural nets) is from 1943 by McCulloch and Pitts [38]. In 1956 Kleene [32] modeled the neural nets of McCulloch and Pitts by finite automata. In that time similar models were presented by Huffman [29], Moore [43], and Mealy [39]. In 1959, Rabin and Scott introduced nondeterministic finite automata (NFAs) in [46].

Stringology[1] is a part of Computer Science dealing with processing strings and sequences. The first tasks of stringology were exact string matching. Later on more complicated tasks appeared like approximate string matching and spell-checking. In the last years new and even more complicated tasks appeared arising from such application fields like data compression, natural and formal language processing, DNA processing, bioinformatics, image processing, musicology, etc.

The finite automata play very important role in stringology. Some algorithms solving stringology tasks use deterministic finite automata (DFAs) and some were developed without the knowledge they actually simulate NFAs, which can be constructed to solve these tasks. If we construct an NFA solving a new stringology task, we can use the same principle of NFA simulation. Thus we get an algorithm solving the new stringology

[1]The term stringology was for the first time used by Zvi Galil in [17].

task. Not only the NFA simulation but also other automata approaches discussed in this paper may be involved in designing a new algorithm for a given stringology task.

A general introduction of automata in stringology is given in [20] which is rather focused on application in bioinformatics. This paper presents more theoretically oriented view and adds some approaches for composition of automata and input alphabet issues.

2. FINITE AUTOMATA IN STRINGOLOGY

Let $w \in \Sigma^*$ be a string over alphabet Σ. The length of string w (denoted by $|w|$) is the number of symbols of the string. String w can be constructed as a concatenation of strings x, y, and z (i. e., $w = xyz, x, y, z \in \Sigma^*$). We say that x, y, and z are a prefix, a factor, and a suffix of w, respectively.

A *nondeterministic finite automaton* (NFA) is a five-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite input alphabet, δ is a mapping from $Q \times (\Sigma \cup \{\varepsilon\}) \mapsto 2^Q$, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states. Finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ is *deterministic* (DFA) if $\delta(q, \varepsilon) = \emptyset$ for all $q \in Q$, and $|\delta(q, a)| \leq 1$ for all $q \in Q$ and $a \in \Sigma$. A *language accepted by finite automaton* M denoted by $\mathcal{L}(M)$, is the set $\{w \mid w \in \Sigma^*, \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$ where $\hat{\delta}$ is the extended transition function defined $\hat{\delta}(q, \varepsilon) = \{q\}$, $\hat{\delta}(q, ua) = \{p \mid q' \in \hat{\delta}(q, u), \delta(q', a) = p, a \in \Sigma, u \in \Sigma^*\}$.

We have identified four kinds of finite automata use involved in efficient solutions of various stringology tasks:

1. a direct use of deterministic finite automata (DFA),
2. a simulation of nondeterministic finite automata (NFA),
3. a use of finite automata as a model for computation,
4. a composition of various automata approaches for particular subtasks.

2.1. The direct use of DFA

The traditional finite automata as defined above are accepting automata. The accepting automaton $\mathcal{A}_A$ reads whole input text $t \in \Sigma^*$ and then text t is accepted (verified) if the finite automaton has reached a final state (i. e., $t \in \mathcal{L}(\mathcal{A}_A)$). In addition, pattern matching automata have been designed in stringology. They traverse the text t and report any location of a given pattern p (i. e., $up \in \mathcal{L}(\mathcal{A}_{\mathrm{PM}})$, where up is a prefix of t and $u \in \Sigma^*$). So in each step the pattern matching automaton $\mathcal{A}_{\mathrm{PM}}$ checks whether any final state is reached. The verification (accepting) automaton $\mathcal{A}_A$ performs the check only once at the end of the text traversal. Automaton $\mathcal{A}_A$ can be used for answering question if pattern p is a factor of the text t (i. e., if $t = upv \in \mathcal{L}(\mathcal{A}_A), u, v \in \Sigma^*$) while automaton $\mathcal{A}_{\mathrm{PM}}$ can find all locations of p in t. See Figure 1.

Another approach used in stringology is to exchange text and pattern—we build an automaton for the input text and give the pattern as an input to the finite automaton. This is the case of indexing automata: suffix trie, suffix tree [34], suffix automata [7, 12] (also called DAWG – Directed Acyclic Word Graph), and compact suffix automata [8]. The relation among the indexing automata is show in Figure 2. We build an indexing

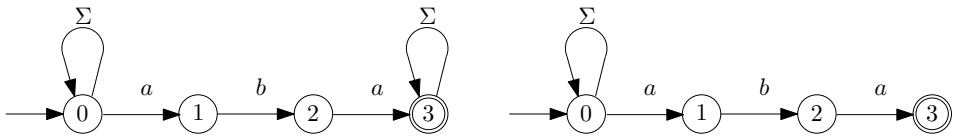


Fig. 1. Nondeterministic finite automata used for searching for pattern $p = aba$.

automaton for a given text t and then in m steps we figure out if a given pattern p of length m is a factor (substring) of text t. That is very efficient full-text searching in a collection of documents where the documents are given in advance while the pattern is provided at the moment when we want to know in which documents it is located.

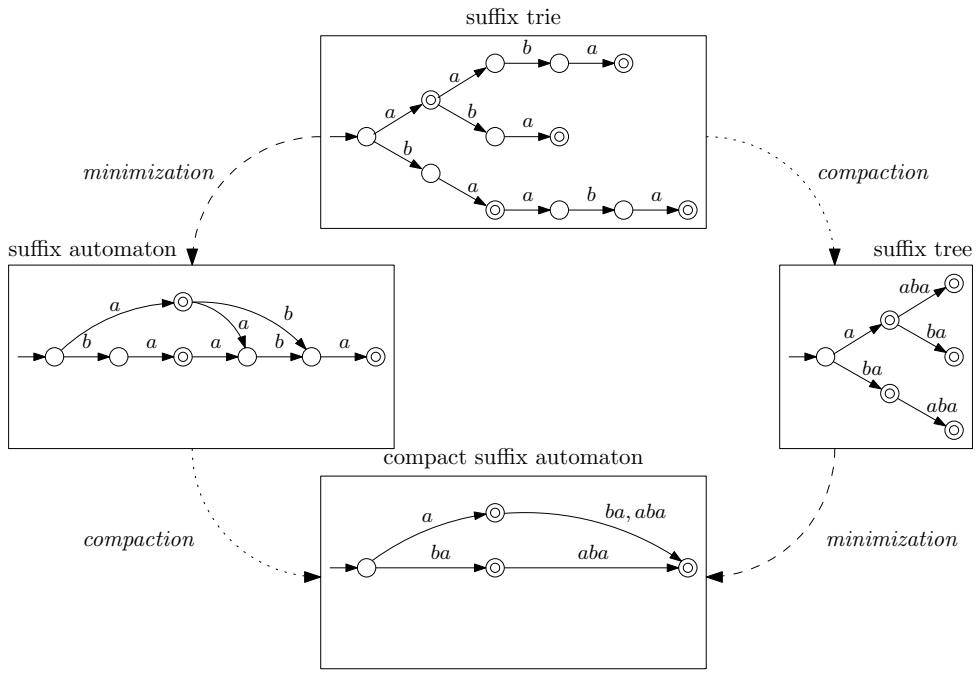


Fig. 2. Automata used for indexing text $t = baaba$.

For all above mentioned finite automata usages the DFAs are used. If we construct NFA, we have to transform it to the equivalent DFA. Knuth-Morris-Pratt (KMP) type automaton [33][2] is a special DFA extended by fail function. KMP automaton does not have all outgoing transitions defined (it is called not completely defined automaton). If

[2]KMP automaton [33] is an optimized version of the original Morris-Pratt (MP) automaton [44].

the automaton should use an undefined transition, the fail function is used instead to reach another state. The fail function is used until we reach a state with the required transition defined. In the worst case the initial state is reached where transitions are defined for all symbols of input alphabet. KMP automaton is very memory efficient, it needs memory linear to the length of the pattern. KMP automaton is used for exact string matching for one pattern. Aho-Corasick (AC) automaton [2] is an extension of KMP automaton for a finite set of patterns.

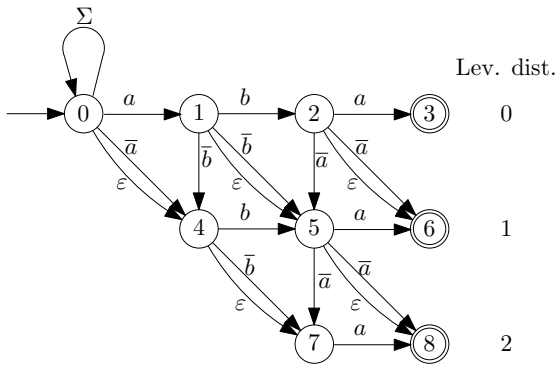
The pattern matching automata are used also in backward pattern matching like in Boyer–Moore [9] family of algorithms. While in Boyer–Moore algorithm the pattern matching automaton does only simple verification and can be replaced by simple `for` loop, its extension [11] for a finite set of patterns the pattern matching automaton performs more complex work.

Another approach is to use an indexing DFA as a filter. The DFA then accepts more strings than just all factors of the text t but on the other hand the size of the DFA is much smaller. When the DFA reports a match, the result has to be verified by an exact method. An example of such technique is the factor oracle [3]—an automaton build over an input text t. The factor oracle is smaller than the suffix automaton. When the factor oracle reports an occurrence of a pattern p, it still has to be verified if p is a factor of t. When the factor oracle reports no occurrence of p, p is for sure no factor of t.

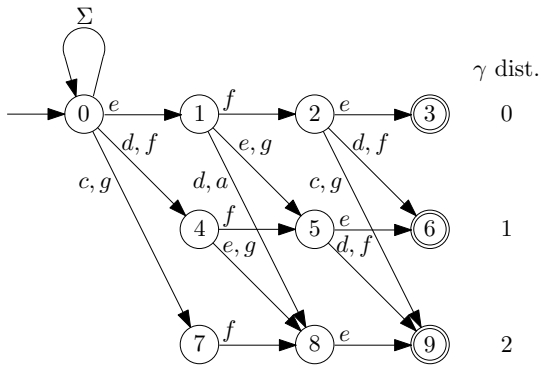
In the approximate string matching the text t is searched not only for exact occurrences of some pattern p but also for occurrences of any string u within a given edit distance bound k (i. e., $d(p, u) \leq k$). The edit distance is the minimum number of edit operations needed to convert p to a found string u. There are many edit operations like: *replace*, *delete*, *insert*, *transpose* (two consecutive symbols be swapped; each symbol can be involved only in one transpose), *increase/decrease symbol rank* in alphabet, ... There are many edit distances defined: Hamming (*replace*) [18], Levenshtein (*replace, insert, delete*) [37], Damerau (*replace, insert, delete, transpose*) [13], indel (*insert, delete*), δ (maximum of rank changes for each position) [10], γ (maximum of rank changes for whole string) [10], (δ, γ) [10], etc.

A nondeterministic pattern matching automaton for approximate string matching can be constructed as $k + 1$ copies of pattern matching automaton for exact string matching like in Figure 1(b) (one for each allowed edit distance $0, 1, \ldots, k$). Then we connect the copies using transitions representing edit operation and delete useless states. The resulting automaton for Levenshtein distance is shown in Figure 3(a). The *replace* transition reads one symbol from the input, advances position in the pattern and leads to the next copy of automaton as the edit operation is increased by one. The *insert* transition reads one symbol from the input, keeps the position in the pattern and leads to the next copy of automaton as the edit operation is increased by one. The *delete* transition reads no symbol from the input, advances the position in the pattern and leads to the next copy of automaton as the edit operation is increased by one.

The automaton [42] used in music for γ-distance is shown in Figure 3(b). In music the alphabet of pitches in one octave is $\Sigma = \{c, d, e, f, g, a, b\}$. We see that the edit operation advances the position in the pattern while it leads to one of the copies according to the difference of ranks of symbols (e. g., $\gamma(e, f) = 1$, $\gamma(e, d) = 1$, $\gamma(e, g) = 2$, $\gamma(e, c) = 2$).



(a) Levenshtein distance, $p = aba, k = 2$



(b) γ distance, $p = efe, k = 2, \Sigma = \{c, d, e, f, g, a, b\}$

Fig. 3. Nondeterministic finite automata used for approximate pattern matching.

2.2. The NFA simulation

DFA cannot be always used. Theoretically we can face up to exponential increase of number of states when determinizing NFA. If the resulting DFA cannot fit into memory, we have to search for another approach. Thus the NFA simulation is used. Instead of determinizing the NFA we traverse the NFA in breadth-first order with a goal to reach a final state. The Basic Simulation Method (BSM) [19] was designed for that purpose. It was implemented using bit vectors.

For NFA with a regular structure we can use other simulation methods called bit parallelism [5, 14, 48, 52] and dynamic programming [47, 50]. They improve time and space complexities, however they cannot be used for general NFA. For example NFA for pattern matching using Levenshtein distance has for each state transitions coming from the same directions as show in Figure 4. The transition *match* **M** keeps the level (representing edit distance), advances the depth (representing position in the pattern) while reading one input symbol. The transitions *replace* **R** and *delete* **D** both advance

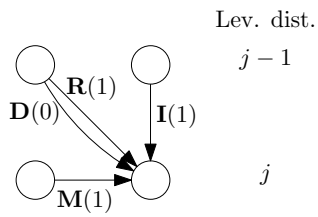


Fig. 4. Transitions in the NFA used for approximate pattern matching.

the level and the depth while only *replace* reads an input symbol. The transition *insert* **I** advances the level, keeps the depth while reading one input symbol.

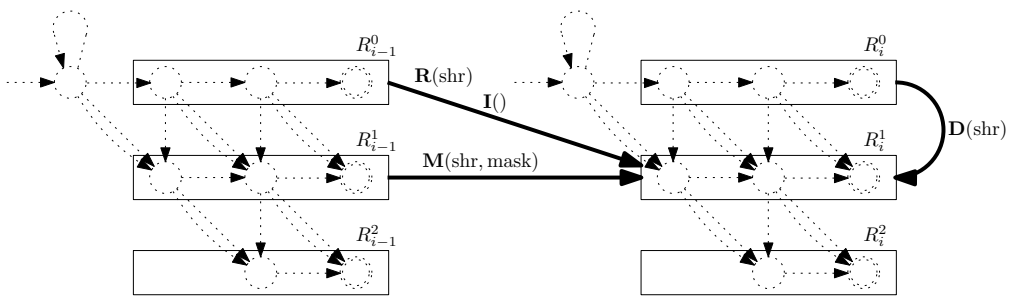


Fig. 5. Bit-Parallelism: NFA simulation.

It was shown [22] that bit parallelism simulates NFA for approximate string matching in such a way that it has for each level $j, 0 \leq j \leq k$, of states (one row) one bit vector R^j as shown in Figure 5. Value 0 means the corresponding state is active. The incoming transitions of all states accommodated in bit vector R^j_i (in step i) are computed in parallel. The directions are the same as shown in Figure 4. Following the directions the *match* transition shifts the vector (R^j_{i-1}) of the same level (j) in the previous step $(i-1)$ to the right while one have to select only the matching positions. It is done using operation OR with a mask vector for the current input symbol t_i. The *replace* transition shifts the vector (R^{j-1}_{i-1}) of the previous level in the previous step to the right. The *insert* transition takes the unshifted vector (R^{j-1}_{i-1}) of the previous level in the previous step. And *delete* transition shifts the vector (R^{j-1}_i) of the previous level in the current step (as no input symbol is read) to the right. All incoming transitions are put together using operation AND. The definition of matrix of masking vectors is shown in Formula 1 and the resulting formula for the simulation is shown in Formula 2. The initial state is always active so it is represented by the property of right shift which inserts 0 to the first position of the bit vector. For more details see [22].

$$D[x] = \begin{bmatrix} d_{1,x} \\ d_{2,x} \\ \vdots \\ d_{m,x} \end{bmatrix}, 0 \leq i \leq n, 0 \leq l \leq k, x \in \Sigma. \tag{1}$$

Each element $d_{j,x}$, $0 < j \leq m$, $x \in \Sigma$, contains 0, if $p_j = x$, or 1, otherwise.

$$\begin{array}{lcll} r^l_{j,0} & \leftarrow & 0, & 0 < j \leq l, 0 < l \leq k, \\ r^l_{j,0} & \leftarrow & 1, & l < j \leq m, 0 \leq l \leq k, \\ R^0_i & \leftarrow & \mathbf{shr}(R^0_{i-1}) \text{ OR } D[t_i], & 0 < i \leq n, \\ R^l_i & \leftarrow & (\mathbf{shr}(R^l_{i-1}) \text{ OR } D[t_i]) & \\ & & \text{AND } \mathbf{shr}(R^{l-1}_{i-1} \text{ AND } R^{l-1}_i) \text{ AND } R^{l-1}_{i-1}, & 0 < i \leq n, 0 < l \leq k. \end{array} \tag{2}$$

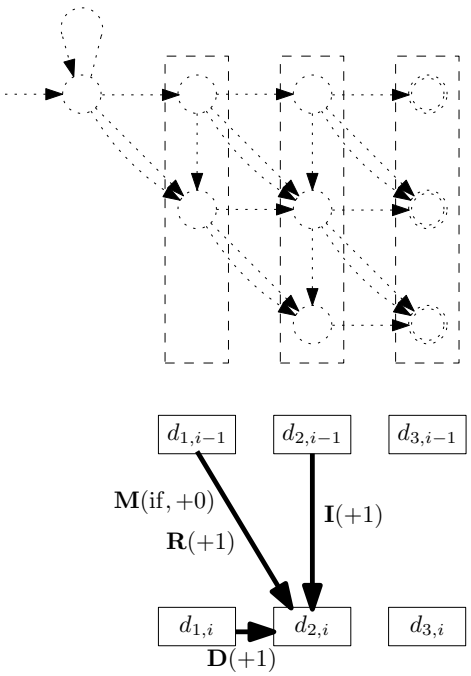


Fig. 6. Dynamic Programming: NFA simulation.

It was shown [22] that dynamic programming simulates NFA for approximate string matching in such a way that there is for each depth $j, 0 \leq j \leq m$, of automaton (one column) one integer variable $d_{j,i}$. It holds the minimum level (edit distance) of all active states in the depth j in step i. As all incoming transition come from the same directions

for each depth of automaton, we can use simple Formula 3 shown below. For more details see [22].

$$\begin{array}{lcll} d_{j,0} & \leftarrow & j, & 0 \leq j \leq m, \\ d_{0,i} & \leftarrow & 0, & 0 \leq i \leq n, \\ d_{j,i} & \leftarrow & \textbf{min}(\textbf{if } t_i = p_j \textbf{ then } d_{j-1,i-1} \textbf{ else } d_{j-1,i-1} + 1, & \\ & & \quad \textbf{if } j < m \textbf{ then } d_{j,i-1} + 1, & \\ & & \quad d_{j-1,i} + 1), & 0 < i \leq n, \\ & & & 0 < j \leq m. \end{array} \tag{3}$$

NFA simulation runs slower than DFA but it requires less memory. NFA simulation is used also in the case when determinization would take too much time with respect to the length of input text. A resolution system [27] was developed that for given stringology task configuration (task, $|p|$, $|t|$, $|\Sigma|$) recommends the most suitable method (DFA or one of simulation methods).

In order to speed up the BSM we have implemented Deterministic State Cache (DSC) [21]. The BSM actually computes the DFA states but only those needed in each step of simulation. It does not store them. On the other hand DFA approach computes all DFA states at the beginning (even those never used). BSM with DSC combines advantages of both DFA run and NFA simulation. It computes the DFA states on the fly and store them in the DSC. If DSC contains both source and destination DFA states of required transition it is used from DSC instead being computed. If DSC is full, the standard cache technique to free some space in it is used.

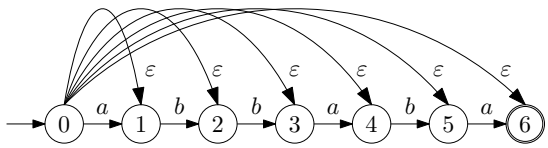
Once we know how to simulate NFA efficiently we can use this approach for other stringology tasks. The bit parallel simulation applies in one of the fastest exact string matching algorithms called BNDM (Backward Nondeterministic DAWG Matching) [45] working on the principle of backward pattern matching. While DFA directly used cannot bring sublinear solution, BNDM runs in sublinear time. The bit parallelism is used also for factor oracle in BOM (Backward Oracle Matching) algorithm [3] working in sublinear time.

2.3. The use of finite automata as a model for computation

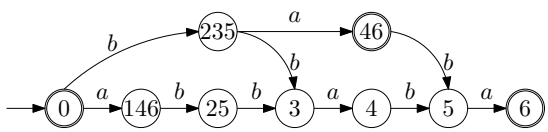
The finite automaton does not need to be run to solve some task. Even the construction of DFA can give us an answer. In [40] it is shown how to find all repetitions in the given string using finite automata. First a nondeterministic suffix automaton (see Figure 7(a)), where each state represents one position in the input text (the position would be a number of the state), is constructed. Then a deterministic suffix automaton (see Figure 7(b)) is constructed by eliminating ε-transitions and determinizing it by the standard subset construction [28] with preserving these positions in d-subsets[3]. Traversing the resulting deterministic suffix automaton in breadth-first search following only the states with d-subsets of cardinality greater than 1, we identify the following repetitions. Factor a is repeated 3-times (at positions 1, 4, and 6), factor b is repeated 3-times (at

[3] d-subset is a set of nondeterministic states composing deterministic states. It is a product of the standard subset construction.

positions 2, 3, and 5), factor ab is repeated twice (at positions[4] 2 and 5), and factor ba is repeated twice (at positions 4 and 6).



(a) Nondeterministic suffix automaton



(b) Deterministic suffix automaton

Fig. 7. Suffix automaton used for searching for repetitions in text $t = abbaba$.

Another example is searching for all borders of the text by a construction of intersection of prefix and suffix automata as shown in [49]. The border of the text is a prefix which is simultaneously a suffix of the text. As we know the automaton model behind the computation we can easily extend into searching for the approximate borders as also shown in [49].

2.4. The composition of various automata approaches for particular subtasks

A solution of a complex stringology task can be decomposed into several subtasks. If we find particular solutions of the individual subtasks using some of above-mentioned automata approaches, we can compose these particular solutions to get the solution of the original task.

Parallel composition In the approximate string matching task, we search for all exact and approximate occurrences of a given pattern p in a given text t. Both text t and pattern p are preprocessed. We build a suffix automaton $\mathcal{A}_{\mathrm{S}}$ for the text t and an approximate string matching automaton $\mathcal{A}_{\mathrm{ASM}}$ for the pattern p with given maximum edit distance k. The solution is then the composition of these two automata—we build an automaton $\mathcal{A}_{\mathrm{ASM}\cap\mathrm{S}}$ for intersection of languages accepted by the original automata. $\mathcal{L}(\mathcal{A}_{\mathrm{S}})$ contains all factors of t and $\mathcal{L}(\mathcal{A}_{\mathrm{ASM}})$ contains all words within a given edit distance from string p. If the resulting automaton $\mathcal{A}_{\mathrm{ASM}\cap\mathrm{S}}$ has a final state (i. e., a state composed of final state q of $\mathcal{A}_{\mathrm{ASM}}$ and a final state q' of $\mathcal{A}_{\mathrm{S}}$), then text t contains pattern p with the edit distance given by the final state q. It is not necessary to build whole automaton $\mathcal{A}_{\mathrm{ASM}\cap\mathrm{S}}$. We have to follow the directions leading to final states (starting from those representing edit distance 0 towards k). For more details see [23].

[4]Positions where the string ends.

Serial composition A task to find a sequence of patterns each of different kind like (p_1, exact string matching), (p_2, approximate string matching, Levenshtein distance), (p_3, regular expression matching) is an example of serial composition. One has to construct NFA for each part (i. e., for each pattern p_1, p_2, and p_3) and connect final states of the previous NFA to the initial state of the following NFA as shown in Figure 8. Only the final state ($3''$) of the last NFA became final. The final states $(3, 3', 6')$ of the other NFAs became nonfinal in the resulting automaton.

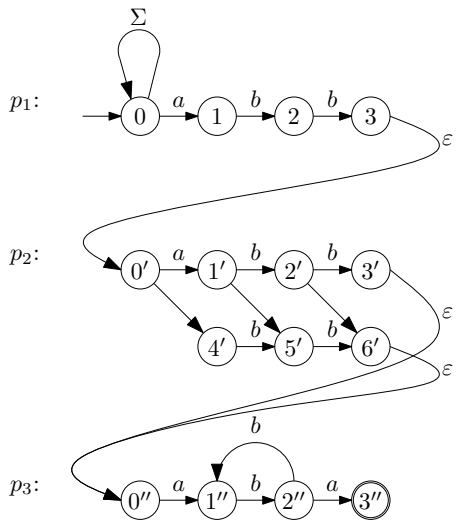
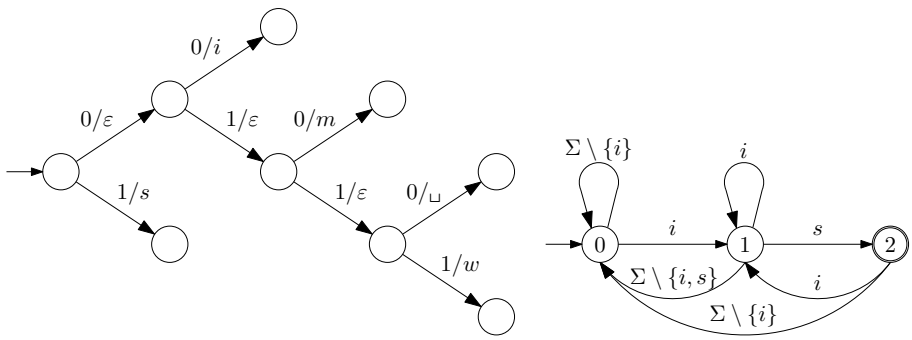


Fig. 8. Serial composition: Connecting NFA for exact string matching ($p_1 = abb$), NFA for approximate string matching ($p_2 = abb$, Hamming distance up to 1), and NFA for regular expression $(ab(bb)^*a)$.

There is also more complex serial composition of automata solutions, where the solution of one subtask is built on top of the solution of the previous subtask. In [4] an algorithm that uses finite automata to find the common motifs with gaps occurring in all strings belonging to a finite set $S = \{S_1, S_2, \ldots, S_r\}$ is presented. First the factors that exist in each string should be identified. Therefore the algorithm begins with constructing a suffix automaton $\mathcal{A}_{S_i}$ for each string S_i. An automaton that accepts the union of the above-mentioned automata is computed. Thus we get an automaton that can identify all factors of all strings and for each factor a list of strings it is located in. A factor alphabet Σ_{fac} is created that contains only those factors located in all strings of S together. Using Σ_{fac} a finite automaton is created for each string S_i that accepts all non overlapping sequences of strings of Σ_{fac} in S_i. The intersection of the latter automata produces the finite automaton which accepts all the common subsequences with gaps over the factor alphabet that are present in all the strings of the set $S = \{S_1, S_2, \ldots, S_r\}$. These common subsequences are the common motifs of the strings.

There has been an attempt [41] to describe pattern matching tasks according to 6

dimensions (nature of pattern, integrity of pattern, number of patterns, edit distance, importance of symbols, number of pattern instances). The idea was to have a method for each dimension modifying a pattern matching algorithm as we move along the dimension. Given a pattern matching task one starts with the simplest task algorithm (exact string matching) and modifies the algorithm by a method for each dimension. The system is based on NFA. The resulting NFA was then determinized in order to obtain the resulting deterministic solution. However, there appeared too many new different tasks so the system cannot be kept up to date.



(a) A finite transducer translating codewords over $\{0, 1\}$ to original alphabet $\{␣, i, m, s, w\}$

(b) Pattern matching DFA for pattern $p = is$

Fig. 9. Cascade composition: Pattern matching in compressed text using Huffman coding.

Cascade composition. In the pattern matching in compressed text the cascade composition may be applied. Given string $t' \in \{0, 1\}^*$ which is the result of compressing a string $t \in \Sigma^*$ using Huffman Coding [30]. The task is to find all occurrences of a given pattern $p \in \Sigma^*$ in t. We can have a finite transducer (see Figure 9(a)) reading compressed text t' and outputting t. Then the exact string matching DFA (see Figure 9(b)) can read output of the transducer and performs pattern matching. In [35] each transition in the DFA labeled by an alphabet symbol ($\{␣, i, m, s, w\}$) is replaced by a sequence of transitions labeled by bits according to the Huffman Code used. The resulting pattern matching DFA then runs over the compressed text and reports all occurrences of the pattern without the need of decompression. Note that we cannot simply search for the Huffman code of p in t' as there may be false reports due to wrong alignment. Huffman codes are in general variable length codes.

2.5. Complex Input Alphabet

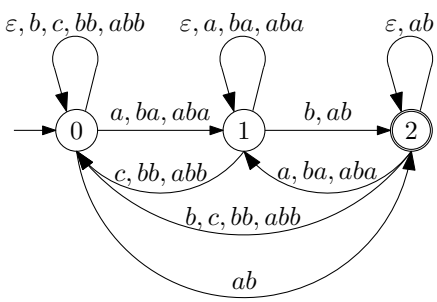
The finite automata can easily handle more complex input alphabet than just single symbols.

Symbol matching a set of symbols. The input alphabet for the pattern and/or the text is not always a finite set of solid symbols Σ. Sometimes a don't care symbol (denoted '$*$') [15] is used that matches any symbol of Σ. A generalisation is a degenerate symbol [31, 51] (also called generalized [1] or indeterminate [24, 25, 26]) which matches some subset of Σ. The finite automata can process these symbols very easily. The transition representing a degenerate symbol μ is implemented by multiple transition labeled by all symbols matching μ. The NFA simulation methods discussed in Section 2.2 need only small modifications. BSM should be built over the NFA with multiple transitions. Bit Parallelism needs just an extra preprocessing of the mask table. Only Dynamic Programming needs a slightly higher time and space complexity as we need to use a table for degenerate symbol matching relation. The finite automata can also efficiently describe amino acids that may be encoded by various triplets of nucleobases where the strings over degenerate symbols are not sufficient [20].

Symbol as variable. In parametrized string matching [6] the alphabet Σ of fixed symbols is extended by an alphabet Λ of parameter symbols. The symbols of Σ must match exactly while the parameter symbols may be renamed. If we have $\Sigma = \{a, b\}$, $\Lambda = \{x, y, z\}$ then $axbaxbybbya$ matches $aybaybzbbza$ with parameter mapping $x \mapsto y$ and $y \mapsto z$. The automata solution in [16] uses the suffix automaton [7] and bit parallelism.

Class	members
ε	ε
a	$a, aa, ca, \ldots$
b	b
c	$c, ac, cb, cc, \ldots$
ab	$ab, abab, aab, bab, \ldots$
ba	$ba, baa, bba, \ldots$
bb	$bb, bc, bac, bbb, bbc, \ldots$
aba	$aba, abaa, abba, \ldots$
abb	$abb, abc, abac, aabb, abbb, abbc, \ldots$

(a) String equivalence classes with respect to $\mathcal{A}_{\mathrm{PM}}$.



(b) DFA for matching in compressed text ($p = ab$.)

Fig. 10. Pattern matching in LZ78 compressed text.

Symbol matching whole strings. More complex input alphabet preprocessing for exact string matching in LZ78 [53] compressed text is presented in [36]. The exact string matching DFA $\mathcal{A}_{\mathrm{PM}}$ is constructed for given pattern p over alphabet Σ. A system of equivalence classes (see Figure 10(a)) of all strings over Σ with respect to $\mathcal{A}_{\mathrm{PM}}$ is built. $\mathcal{A}_{\mathrm{PM}}$ is extended so that each state has outgoing transitions for all equivalence classes (see Figure 10(b)). An LZ78 compressed text is read. Each LZ78 token represents a string which is classified into one of equivalence classes. $\mathcal{A}_{\mathrm{PM}}$ then performs a transition labeled by the equivalence class. Given an LZ78 compressed text $(0, a), (1, a), (2, b), (3, b)$ representing a sequence of strings $a, aa, aab, aabb$. $\mathcal{A}_{\mathrm{PM}}$ performs the following sequence

of transitions $0 \xrightarrow{a} 1 \xrightarrow{a} 1 \xrightarrow{ab} 2 \xrightarrow{abb} 0$. Thus the text is searched in time linear with the length of the compressed text and not with the length of the original text.

3. CONCLUSION

We have identified several finite automata approaches and shown their applications in various stringology algorithms. Very important technique is to decompose a given task into several subtasks (in parallel, serial, or cascade way), to find solutions for the subtasks, and to construct the resulting solution form the particular solutions. Complex alphabet symbols (degenerate, variables, strings) can be processed just by modifications of transitions in finite automata.

ACKNOWLEDGEMENT

This research has been partially supported by the Ministry of Education, Youth and Sports of Czech Republic under research program MSM 6840770014, and by the Czech Science Foundation as project No. 201/09/0807.

(Received July 16, 2011)

REFERENCES

[1] K. Abrahamson: Generalized string matching. SIAM J. Comput. *16* (1987), 6, 1039–1051.

[2] A. V. Aho and M. J. Corasick: Efficient string matching: an aid to bibliographic search. Commun. ACM *18* (1975), 6, 333–340.

[3] C. Allauzen, M. Crochemore, and M. Raffinot: Factor oracle: A new structure for pattern matching. In: SOFSEM'99, Theory and Practice of Informatics (J. Pavelka, G. Tel, and M. Bartošek, eds.), Milovy 1999, Lect. Notes Comput. Sci. *1725* (1999), Springer-Verlag, Berlin, pp. 295–310.

[4] P. Antoniou, J. Holub, C. S. Iliopoulos, B. Melichar, and P. Peterlongo: Finding common motifs with gaps using finite automata. In: Implementation and Application of Automata (O. H. Ibarra and H.-C. Yen, eds.), Lect. Notes Comput. Sci. *4094* (2006), Springer-Verlag, Heidelberg, pp. 69–77.

[5] R. A. Baeza-Yates and G. H. Gonnet: A new approach to text searching. Commun. ACM *35* (1992), 10, 74–82.

[6] B. S. Baker: Parameterized duplication in strings: algorithms and an application to software maintenance. SIAM J. Comput. *26* (1997), 5, 1343–1362.

[7] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M. T. Chen, and J. Seiferas: The smallest automaton recognizing the subwords of a text. Theor. Comput. Sci. *40* (1985), 1, 31–44.

[8] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnel: Complete inverted files for efficient text retrieval and analysis. J. Assoc. Comput. Mach. *34* (1987), 3, 578–595.

[9] R. S. Boyer and J. S. Moore: A fast string searching algorithm. Commun. ACM *20* (1977), 10, 762–772.

[10] E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon: Algorithms for computing approximate repetitions in musical sequences. In: Proc. Tenth Australian Workshop on Combinatorial Algorithms (R. Raman and J. Simpson, eds.), AWOCA'99, School of Computing, Curtin University of Technology, Perth 1999, pp. 129–144.

[11] B. Commentz-Walter: A string matching algorithm fast on the average. In: Proc. 6th International Colloquium on Automata, Languages and Programming (H. A. Maurer, ed.), Graz 1979, Lect. Notes Comput. Sci. *71* (1979), Springer-Verlag, Berlin, pp. 118–132.

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

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

[14] B. Dömölki: An algorithm for syntactical analysis. Comput. Linguistics *3* (1964), 29–46, 1964.

[15] M. J. Fischer and M. Paterson: String matching and other products. In: Proc. SIAM-AMS Complexity of Computation (R. M. Karp, ed.), Providence 1974, pp. 113–125.

[16] K. Fredriksson and M. Mozgovoy: Efficient parameterized string matching. Inf. Process. Lett. *100* (2006), 3, 91–96.

[17] Z. Galil: Open problems in stringology. In: Combinatorial Algorithms on Words (A. Apostolico and Z. Galil, eds.), NATO ASI Series F *12* (1985), Springer-Verlag, Berlin, pp. 1–8.

[18] R. W. Hamming: Error detecting and error correcting codes. The Bell System Techn. J. *29* (1950), 2, 147–160.

[19] J. Holub: Bit parallelism-NFA simulation. In: Implementation and Application of Automata (B. W. Watson and D. Wood, eds.), Lect. Notes in Comput. Sci. *2494* (2002), Springer-Verlag, Heidelberg, pp. 149–160.

[20] J. Holub: Finite automata in pattern matching. In: Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications (M. Elloumi and A. Y. Zomaya, eds.), John Wiley and Sons, 2011, pp. 51–71.

[21] J. Holub and T. Kadlec: NFA simulation using deterministic state cache. In: London Algorithmics 2008: Theory and Practice (J. Chan, J. W. Daykin, and M. S. Rahman, eds.), College Publications 2009, pp. 152–166.

[22] J. Holub and B. Melichar: Implementation of nondeterministic finite automata for approximate pattern matching. In: Proc. 3rd International Workshop on Implementing Automata, Rouen 1998, pp. 74–81.

[23] J. Holub and B. Melichar: Approximate string matching using factor automata. Theor. Comput. Sci. *249* (2000), 2, 30–311.

[24] J. Holub and W. F. Smyth: Algorithms on indeterminate strings. In: Proc. 14th Australasian Workshop On Combinatorial Algorithms (M. Miller and K. Park, eds.), Seoul National University, Seoul 2003, pp. 36–45.

[25] J. Holub, W. F. Smyth, and S. Wang: Hybrid pattern-matching algorithms on indeterminate strings. In: London Stringology Day and London Algorithmic Workshop 2006 (J. Daykin, M. Mohamed, and K. Steinhoefel, eds.) King's College London Series Texts in Algorithmics 2007, pp. 115–133.

[26] J. Holub, W. F. Smyth, and S. Wang: Fast pattern-matching on indeterminate strings. J. Discret. Algorithms *6* (2008), 1, 37–50.

[27] J. Holub and P. Špiller: Practical experiments with NFA simulation. In: Proc. Eindhoven FASTAR Days 2004 (L. Cleophas and B. W. Watson, eds.), TU Eindhoven 2004, The Finite Automata Approaches in Stringology *15*, pp. 73–95.

[28] J. E. Hopcroft and J. D. Ullman: Introduction to automata, languages and computations. Addison-Wesley, Reading 1979.

[29] D. A. Huffman: The synthesis of sequential switching circuits. J. Franklin Inst. *257* (1954), 161–190, 275–303.

[30] D. A. Huffman: A method for the construction of minimum-redundancy codes. Proc. Inst. Radio Engrs *40* (1952), 9, 1098–1101.

[31] C. S. Iliopoulos, I. Jayasekera, B. Melichar, and J. Šupol: Weighted degenerated approximate pattern matching. In: Proc. 1st International Conference on Language and Automata Theory and Applications, Tarragona 2007.

[32] S. C. Kleene: Representation of events in nerve nets and finite automata. Automata Studies (1956), 3–42.

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

[34] S. Kurtz: Reducing the space requirements of suffix trees. Softw. Pract. Exp. *29* (1999), 13, 1149–1171.

[35] J. Lahoda and B. Melichar: Pattern matching in text coded by finite translation automaton. In: Proc. 7th International Multiconference Information Society (M. Heričko et al., eds.), Institut Jožef Stefan, Ljubljana 2004, pp. 212–214.

[36] J. Lahoda and B. Melichar: General pattern matching on regular collage system. In: Proc. Prague Stringology Conference'05 (J. Holub and M. Šimánek, eds.), Czech Technical University in Prague 2005, pp. 153–162.

[37] V. I. Levenshtein: Binary codes capable of correcting deletions, insertions and reversals. Sov. Phys. Dokl. *6* (1966), 707–710.

[38] W. S. McCulloch and W. Pitts: A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophys. *5* (1943), 115–133.

[39] G. H. Mealy: A method for synthetizing sequential circuits. Bell System Techn. J. *34* (1955), 5, 1045–1079.

[40] B. Melichar: Repetitions in text and finite automata. In: Proc. Eindhoven FASTAR Days 2004 (L. Cleophas and B. W. Watson, eds.), TU Eindhoven 2004, pp. 1–46.

[41] B. Melichar and J. Holub: 6D classification of pattern matching problems. In: Proceedings of the Prague Stringology Club Workshop'97 (J. Holub, ed.), Czech Technical University in Prague, 1997, pp. 24–32. Collaborative Report DC-97-03.

[42] B. Melichar, J. Holub, and T. Polcar: Text searching algorithms. Vol. I: Forward string matching. Textbook for course Text Searching Algorithms, 2005.

[43] E. F. Moore: Gedanken experiments on sequential machines. Automata Studies (1965), 129–153.

[44] J. H. Morris, Jr and V. R. Pratt: A Linear Pattern-Matching Algorithm. Report 40, University of California, Berkeley 1970.

[45] G. Navarro and M. Raffinot: A bit-parallel approach to suffix automata: Fast extended string matching. In: Proc. 9th Annual Symposium on Combinatorial Pattern Matching (M. Farach-Colton, ed.), Lect. Notes in Comput. Sci. *1448*, Piscataway, NJ, 1998. Springer-Verlag, Berlin, pp. 14–33.

[46] M. O. Rabin and D. Scott: Finite automata and their decision problems. IBM J. Res. Dev. *3* (1959) 114–125.

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

[48] R. K. Shyamasundar: A Simple String Matching Algorithm. Technical Report, Tata Institute of Fundamental Research 1976.

[49] M. Šimůnek and B. Melichar: Borders and finite automata. In: Implementation and Application of Automata (O. H. Ibarra and H.-C. Yen, eds.), Lecture Notes in Comput. Sci. *4094*, Springer-Verlag, Heidelberg 2006, pp. 58–68.

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

[51] M. Voráček, L. Vagner, S. Rahman, and C. S. Iliopoulos: The constrained longest common subsequence problem for degenerate strings. In: Implementation and Application of Automata (J. Holub and J. Žďárek, eds.), Lecture Notes in Comput. Sci. *4783*, Springer-Verlag, Heidelberg 2007, pp. 309–311.

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

[53] J. Ziv and A. Lempel: Compression of individual sequences via variable length coding. IEEE Trans. Inf. Theory *24* (1978), 530–536.

Jan Holub, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, Thákurova 2700/9, 160 00 Praha 6. Czech Republic.
e-mail: Jan.Holub@fit.cvut.cz