České vysoké učení technické v Praze
Fakulta elektrotechnická

Czech Technical University in Prague
Faculty of Electrical Engineering

Faktorové automaty a jejich implementace
Factor Automata and Their Implementations

Ing. Jan Holub, Ph.D.

Summary

Factor and suffix automata accept all factors and suffixes, respectively, of a given text T for which they are built. These automata allow us to find whether a given pattern P is located in text T in time linear with the length of pattern P. The automata represent a complete index of text T therefore they are called indexing automata. There are many other tasks over the indexing automata. There are also other data structures for indexing the text like suffix trie, suffix tree, and suffix array. However, suffix and factor automata are more memory efficient than suffix trie and suffix tree and they allow faster searching than suffix array. We present survey of indexing structures as well as efficient implementations of factor automata.

Souhrn

Faktorové a sufixové automaty přijímají všechny faktory (podřetězce) a sufixy (přípony) textu T, pro který jsou vytvořeny. Tyto automaty nám umožní zjistit, zda daný vzorek P je obsažen v textu T v čase lineárním k délce vzorku. Tyto automaty vytváří tzv. kompletní index textu T, a tak jsou nazývány indexujícími automaty. Existuje mnoho úloh, které indexující automaty řeší. Dále existuje mnoho dalších datových struktur pro úplné indexování textu jako suffix trie, suffix tree a suffix array. Nicméně faktorové a sufixové automaty jsou paměťově méně náročné než suffix trie a suffix tree. V této přednášce představujeme přehled indexovacích struktur a efektivních implementací faktorových automatů.

Klíčová slova: faktorový automat, sufixový automat, vyhledávání v textu, úplný index textu, indexující struktury, implementace

Keywords: factor automaton, suffix automaton, pattern matching, complete index of text, indexing structures, implementation

Contents

1 Introduction

The amount of information to be processed by computers is increasing in extremely rapid speed. Together with the grow of amount of information the need for efficiency of information processing, storing (compressing), sorting and searching is more and more important. The best developed area is a processing of structured data. The structured data can be easily stored in relational databases. The query language, storage and concurrent access is already very efficient. Besides this strictly structured data including XML databases (with query languages like XPath) the importance of Internet grows where data are presented in HTML form. Here, the basic elements are individual words that compose sentences, paragraphs and pages. One of the companies processing such data is Google who announced in 2006 that it already processed more than 25×10^9 pages.

Unstructured data is another class of information that is needed to be searched in. A typical representative is DNA sequence which is composed of four elements called nucleotides (Adenine, Cytosine, Guanine, and Thymine). DNA sequence is completely unstructured text which is very long. For example Escherichia Coli, which is a bacterium commonly found in the lower intestine of warm-blooded animals, has about 4×10^6 nucleotides. On the other hand the human genome has about 3×10^9 nucleotides. Searching for some pattern in a statistically relevant collection of human genomes is a complicated task. It is not possible to read all DNA sequences and search for the pattern. We have to preprocess the DNAs first.

This lecture focuses on structures called factor automata which store the information in a space efficient form. Although the number of all factors (substrings) in a text $T = t_1 t_2 \dots t_n$ (of length n) is n^2 the resulting factor automaton is linear with n. Moreover it allows to search for the pattern $P = p_1 p_2 \dots p_m$ (of length m) in the time linear to the length m of pattern regardless the size n of the text. This is very important since in biology the pattern size is usually dozens of nucleotides.

Automata for indexing text are discussed below and their relation is presented in Fig. 1. *Suffix trie* [Fre60, UW93] is a basic deterministic finite automaton with a tree structure where each substring is represented by a state (a leaf or an internal node in the transition diagram) in the tree. Some states are called final states. The path from the initial state (root node) to a final state then spells the suffix represented by the final state. Each transition is labelled by exactly one symbol. We distinguish two operations on the suffix trie: compaction and minimization.

The compaction is a process replacing each non-branching path by a transition labelled by a string spelling the path while all final states must be preserved. When we apply operation compaction to the suffix trie we get the *suffix tree* [Wei73, McC76, Ukk95] as shown in Fig. 1. Since no final state can disappear due to compaction, the suffix tree in Fig. 1 contains two internal states and not only one.

The minimization is the standard minimization of number of states of finite automaton. When we apply the minimization to the suffix trie we get the *suffix automaton* (also called DAWG, Directed Acyclic Word Graph) [BBE$^+$85, Cro86]. Further compaction leads then to the *compact suffix automaton* (also called CDAWG, Compact DAWG) [BBE$^+$85, BBE$^+$87, CV97b], which has less states and transitions labeled by strings. The *compact suffix automaton* can also be constructed by minimization of the suffix tree. Direct constructions, avoiding a preliminary suffix automaton construction, has been de-

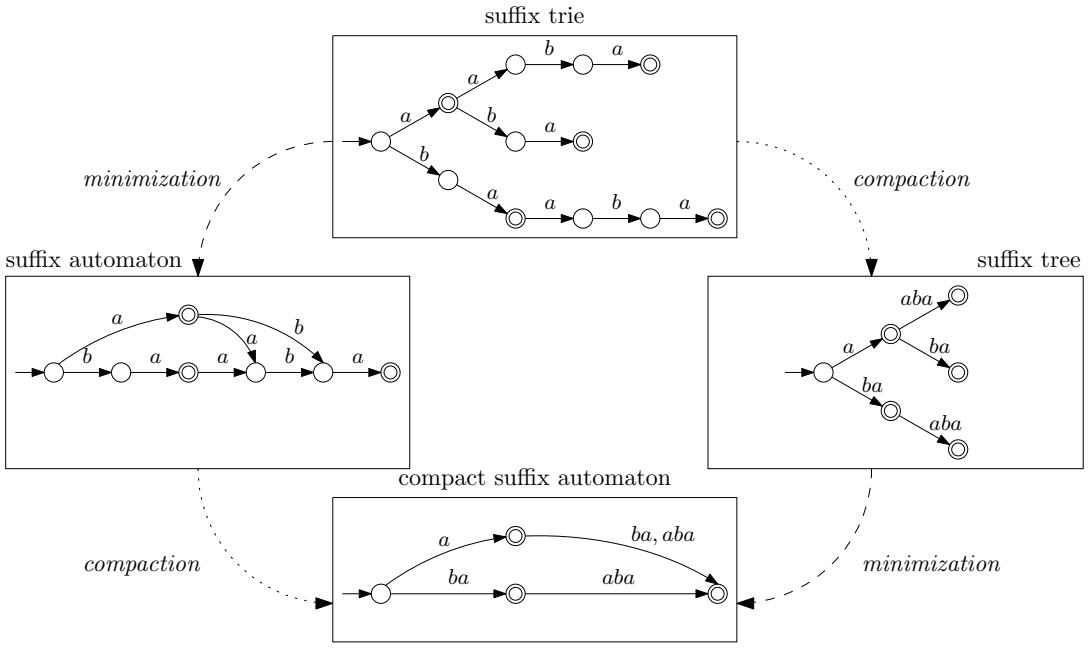

Figure 1: Relation among suffix trie, suffix tree, suffix automaton, and compact suffix automaton ($T = baaba$)

signed in [CV97b] and [IHS+01]. See the difference between suffix and factor automaotn in Fig. 2.

The suffix automaton of text T recognizes all suffixes and factors of T. On the other hand the factor automaton of text T is able to recognize only all factors of T. Since the suffix automaton is easier to build and it is not much larger than the corresponding factor automaton, the suffix automaton is usually used instead of the factor automaton. Well known notion DAWG (resp. CDAWG) does not distinguish between factor automaton and suffix automaton (resp. compact factor automaton and compact suffix automaton).

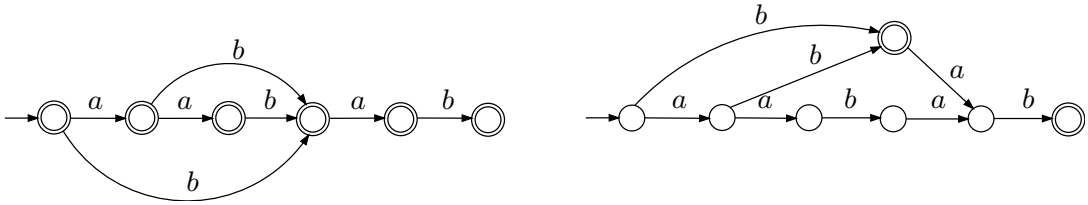

Figure 2: Factor automaton versus suffix automaton ($T = aabab$)

In this lecture we describe our implementation of compact suffix automaton and compare it with other similar structures allowing fast pattern searching. While previous implementations of compact suffix automata required from $7n$ to $23n$ bytes of memory space [Kur99], we show that ours achieves $1.7n$ to $5n$ bytes for a text T of length n. This proves that the implementation is suitable for large data files, since it minimizes the number of disk accesses.

We also would like to mention a basic non-automata-based data structure for indexing text. It is very popular structure called *suffix array*. It is an array of pointers to all suffixes of T sorted lexicographically. Each pointer references the first symbol of the

corresponding suffix in T. The structure is popular among programmers, because it is simple to construct and it performs fast string matching algorithm. The basic binary search takes $\mathcal{O}(m \log n)$ comparisons. Using some additional information the complexity can be decreased to $\mathcal{O}(m + \log n)$.

2 Factor Automata

Let Σ be a nonempty input alphabet, Σ^* be the set of all strings (words) over Σ, ε be the *empty string*, and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. If $w \in \Sigma^*$, then $|w|$ denotes the *length* of w ($|\varepsilon| = 0$). If $a \in \Sigma$, then $\overline{a} = \Sigma \setminus \{a\}$ denotes a *complement* of a over Σ. If $w = xyz$, $x, y, z \in \Sigma^*$, then x, y, z are *factors* (substrings) of w, moreover, x is a *prefix* of w and z is a *suffix* of w.

Deterministic finite automaton (DFA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, Σ is a set of input symbols, δ is a partial mapping (transition function) $Q \times \Sigma \mapsto Q$, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states. We extend δ to a function $\hat{\delta}$ mapping $Q \times \Sigma^+ \mapsto Q$. DFA with complete mapping δ is called complete, otherwise it is called incomplete. *Terminal state* denotes a state $q \in Q$ that has no outgoing transition (i.e., $\forall a \in \Sigma$, $\delta(q, a) = \emptyset$ or using $\hat{\delta}$: $\forall u \in \Sigma^+, \hat{\delta}(q, u) = \emptyset$).

The suffix automaton of a text T is defined [CH97] as the minimal DFA (not necessarily complete) that recognizes the (finite) set of suffixes of T and the factor automaton of a text T is defined as the minimal DFA that recognizes the (finite) set of factors of T.

The suffix automaton of a text T recognizes word w as a suffix of T if both of the following conditions hold:

1. it reads whole word w (i.e., $\hat{\delta}(q_0, w) = q, q \in Q$),
2. it finishes in a final state (i.e., $\hat{\delta}(q_0, w) = q, q \in F$).

If only the first condition holds, then w is recognized as a factor of T.

On the other hand the factor automaton has all states final so it recognizes word w as a factor of T if it reads whole word w.

The number of states of the suffix automaton of text T ranges from $|T| + 1$ to $2|T| - 1$ [CH97] while the number of states of the factor automaton of text T ranges from $|T| + 1$ to $2|T| - 2$ [CH97].

As one can see the suffix automaton covers the functionality of the factor automaton while it is not much larger. Therefore in practice the suffix automata are used instead of the factor automata since they are easier to build.

Both the factor automaton and the suffix automaton with all states set to final will be considered as factor automaton (the minimality condition is relaxed) further in the text.

3 Application of Factor Automata

The basic task for the factor automaton of text T is to decide if pattern P is located in text T (i.e., if P is a factor of T). In addition it can report the number of occurrences and their positions.

The factor automaton is also used in backward exact string matching—Backward DAWG Matching algorithm (BDM) [CR94]—where it is build for reversed pattern. Its nondeterministic version is also used for fast backward exact string matching—Backward Nondeterministic DAWG Matching (BNDM) [NR00].

The factor automaton can be used also for approximate pattern matching where we allow some errors in each occurrence as shown in [HM00]. The number and nature of errors is given by Hamming distance [Ham50], Levenshtein distance [Lev65], or Damerau distance [Dam64]. In this case we build an approximate pattern matching automaton $\mathcal{M}(P)$ and perform intersection between $\mathcal{M}(P)$ and the factor automaton of text T. If we reach the final state, the approximate pattern was found.

The factor automaton can also be used for findings exact and approximate repeats as shown in [Mel04]. However, for approximate repeats it has to be build with respect to the edit distance used.

4 Implementation

The algorithm at the origin of our implementation of compact suffix automaton processes in two steps: first, we construct the compact suffix automaton as described in [CV97a] and we sort states according to their topological order; then, we classify the states into three classes according to their maximum length of incoming transition label (*maxLen*) and we add fourth class for the terminal state q_T, from which no transition leads:

- Class I (Q_{I}): the initial state q_0 and the states with $maxLen = 1$,
- Class II (Q_{II}): the states with $1 < maxLen \leq Limit$,
- Class III (Q_{III}): the states with $maxLen > Limit$,
- Class IV (Q_{IV}): the terminal state q_T (no transition leads from q_T).

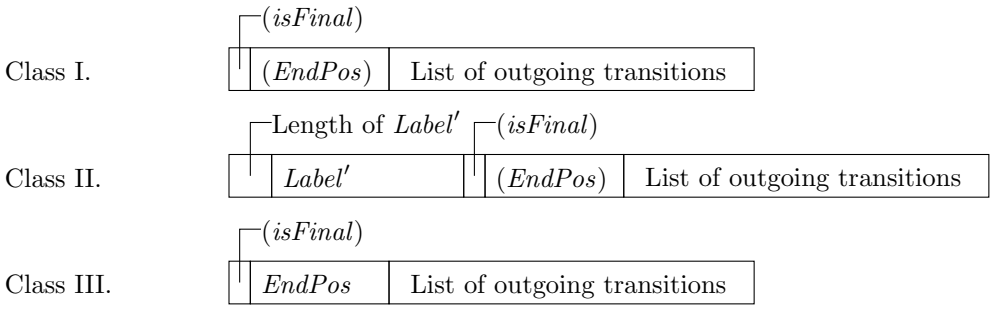

Figure 3: Representation of nodes

The typical distribution of states according to *maxLen* is shown in Table 1. As we can see the most of the labels are very short. That is the reason, why we have decided to create Classes. The most frequent Class I occupies the smallest part of memory. For each state with $maxLen > 1$, a parameter *Limit* distinguishes what implementation (either Class II or III) is more space-efficient.

maxLen	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	···
no. of states	2546	464	231	142	86	57	40	27	25	16	5	3	11	6	3	1	2	0	2	1	0	···

Table 1: Distribution of states according to their *maxLen* in Calgary Corpus file paper4. The maximum *maxLen* for file paper4 is 28.

The structure of information stored for each Class is shown in Fig. 3. For states Q_{I} we store just outgoing transitions. For states Q_{II} and Q_{III} we have to store also incoming transition label. While in Class II we store the whole label (*label*[1], the longest of all incoming transition labels), in Class III we store only a pointer to the source text (*EndPos*). The terminal state (the state in Q_{IV}) is not stored.

We can also store some optional additional data. In Classes I, II, and III we can store one bit (*isFinal*) indicating, whether the corresponding state is final or not. Doing so, the compact suffix automaton recognizes all suffixes and factors of T, otherwise it cannot distinguish between factors and suffixes of T considering that all states are final. To be precise, when we store *isFinal* we implement compact suffix automaton while without *isFinal* we should talk about implementation of compact factor automaton even though it may not be minimal. In Classes I and II we can also store the corresponding position in text (*EndPos*). The terminal state is always final and has $EndPos = n$.

Then we store outgoing transitions as shown in Fig. 4. We store the number (*NoTrans*) of outgoing transitions and the string *FirstSymbols* containing the first symbols of all outgoing transition labels (string of first symbols—*SFS*). Then we store the outgoing transition records as shown in Fig. 5.

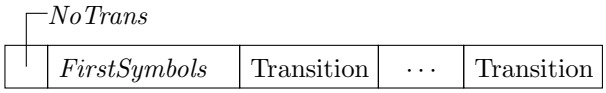

Figure 4: Representation of list of outgoing transitions

At the beginning of each outgoing transition we store the number of Class of the destination state. For Classes I, II, and III we store a pointer to the beginning of the destination state (number of bits to be skipped to reach the destination state). For Classes II, III, and IV we store also the length of outgoing transition label ($Len - 1$).

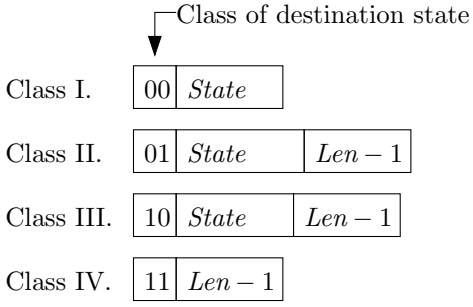

Figure 5: Representation of outgoing transitions

[1]Actually we store *label*′ which has the first symbol deleted, since it is already stored.

We do not store the first symbol of the label since it is already stored in the *SFS*. Removing the *SFS* and storing entire label strings in destination states, could decrease space requirements, but using the compact suffix automaton and searching for the desired transition, would force to read from as many parts of the compact suffix automaton data-file as the number of outgoing transitions. Thus this is likely to require more disk accesses and would significantly increase the resulting searching time.

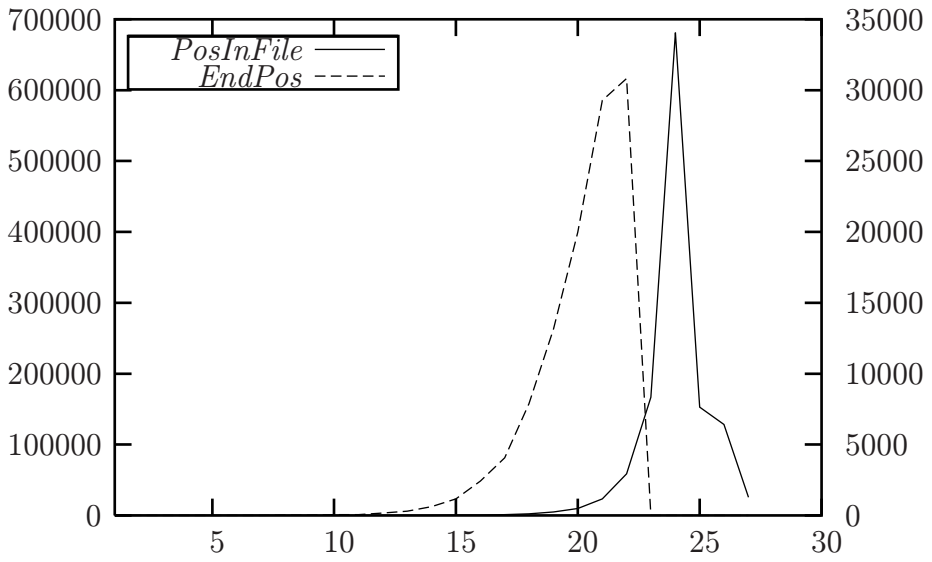

Figure 6: Distribution of numbers *EndPos* (the right scale) and *PosInFile* (the left scale) according to their lengths in bits for file `bible`.

We did an analysis of distribution of integers used in the implementation (pointer to the source text file *EndPos* and pointer in the index file *PosInFile*). Unfortunately the distribution is unsuitable for any variable length encoding (like Elias, Fibonacci, or Golomb codes) so we had to use block binary code whose length is specified in the header of the output bit stream (index file).

5 Results

Kurtz [Kur99] presents implementation experiments on several of indexing structures (suffix automaton, compact suffix automaton, suffix tree).

There is an implementation of suffix automaton by Balík [Bal98]. His implementation is focused on minimization of the size of the index file. On the other hand our implementation focuses on the speed of traversing. Some data are usually stored on several places in the output bit stream while they are stored only once in his implementation. On the other hand it increases the number of disc accesses. Moreover, his implementation uses Huffman encoding for some data.

We made some experiments on Calgary and Canterbury Corpora test files[2] [Bel]. The comparison with other methods is shown in Table 2. A single character in the second

[2]File `ptt5` from Canterbury Corpus and `pic` from Calgary Corpus are the same.

file	type	$\|\Sigma\|$	length	SA	CSA	Suff.Tree	SA.B	CSA.HC1	CSA.HC2
book1	*e*	81	768771	30.35	15.75	9.83	**3.66**	4.42	3.78
book2	*e*	96	610856	29.78	12.71	9.67	3.17	3.67	**3.14**
paper1	*e*	95	53161	30.02	12.72	9.82	2.98	3.26	**2.72**
paper2	*e*	91	82199	29.85	13.68	9.82	3.06	3.58	**3.01**
paper3	*e*	84	46526	30.00	14.40	9.80	3.12	3.62	**3.02**
paper4	*e*	80	13286	30.34	14.76	9.91	3.04	3.46	**2.82**
paper5	*e*	91	11954	30.00	14.04	9.80	2.97	3.34	**2.72**
paper6	*e*	93	38105	30.29	12.80	9.89	2.96	3.27	**2.73**
alice29	*e*	74	152089	30.27	14.14	9.84	**3.20**	3.82	3.23
lcet10	*e*	84	426754	29.75	12.70	9.66	3.12	3.56	**3.03**
plrabn12	*e*	81	481861	29.98	15.13	9.74	**3.52**	4.15	3.53
bible	*e*	64	4047392	29.28	10.87	7.27	2.94	3.26	**2.88**
world192	*e*	94	2473400	27.98	7.87	9.22	2.53	2.43	**2.09**
bib	*f*	81	111261	28.53	9.94	9.46	2.68	2.68	**2.24**
news	*f*	98	377109	29.48	12.10	9.54	3.15	3.44	**2.91**
progc	*f*	92	39611	29.73	11.87	9.59	2.87	3.06	**2.54**
progl	*f*	87	71646	29.96	8.71	10.22	2.40	2.39	**2.03**
progp	*f*	89	49379	30.21	8.28	10.31	2.35	2.28	**1.92**
trans	*f*	99	93695	30.47	6.69	10.49	2.35	1.95	**1.66**
fields.c	*f*	90	11150	29.86	9.40	9.78	2.43	2.39	**1.96**
cp.html	*f*	86	24603	29.04	10.44	9.34	2.64	2.58	**2.12**
grammar	*f*	76	3721	29.96	10.60	10.14	2.36	2.44	**1.97**
xargs	*f*	74	4227	30.02	13.10	9.63	2.75	2.99	**2.40**
asyoulik	*f*	68	125179	29.97	14.93	9.77	3.34	3.84	**3.23**
geo	*b*	256	102400	26.97	13.10	7.49	3.18	2.66	**1.92**
obj1	*b*	256	21504	27.51	13.20	7.69	2.98	2.39	**1.67**
obj2	*b*	256	246814	27.22	8.66	9.30	2.67	1.96	**1.51**
pic	*b*	159	513216	27.86	8.08	8.94	**1.63**	2.17	1.79
kennedy	*b*	256	1029744	21.18	7.29	4.64	1.57	1.65	**1.10**
sum	*b*	255	38240	27.79	10.26	8.92	2.53	2.86	**2.29**
E.coli	*d*	4	4638690	34.01	23.55	12.56	**4.46**	5.46	5.24

Table 2: Space requirements for suffix data structures applied to files of the Calgary and Canterbury Corpora (values are in bytes per symbol of text)

column of the table denotes the type of file: *e* for English text, *f* for formal text (like programs), *b* for binary files (i.e. containing 8-bit symbols), and *d* for DNA sequences. The values for SA (suffix automaton, DAWG), CSA (compact suffix automaton, CDAWG), and Suffix Tree are taken from [Kur99]; SA.B is the implementation of suffix automaton designed by Balík [Bal98]; finally, CSA.HC1 is our implementation. If we do not care about the number of disk accesses, we can move symbols from the *SFS* to the destination state. The space requirements further decrease as we can see in column CSA.HC2. The best results are highlighted.

We made also experiments on several other DNA sequences (of size $n = 40$ kB to 1.5 MB) and the space requirements were about $4.5n$ (CSA.HC1: 4.21–5.15, CSA.HC2:

3.99–4.92).

The suffix array takes space $5n$ according to [MM90].

A further decrease of space requirements can be achieved, if the states are stored at even addresses or addresses divisible by 4 or 8.

6 Conclusions

In the lecture we presented a survey of text indexing structures based on finite automata. We have discussed their application and presented an efficient implementation of compact suffix automaton. The space requirements are much lower than in the previous works and vary from 1.65 to 5.46 bytes per symbol of input text. The implementation requires that the source text is stored, so the total space increases by one byte per character, but other implementations also need the source text. Only [Bal98] can reconstruct the source text from its implementation of suffix automaton but it takes some time.

The space requirements can be further decreased. For instance we can compress the labels or we can remove the *SFS* and store all incoming transition labels in the destination states (CSA.HC2). However as mentioned above, it would increase the number of disk accesses. In our implementation we require at most $m + 1$ disk accesses[3] when searching for a pattern of length m—we traverse at most $m + 1$ states (including the initial state) and all outgoing transitions are located with the state. But in case of CSA.HC2, we would get $(m\tau + 1)$ disk accesses, where τ is the average number of outgoing transitions (related to the size of the alphabet). In such a case we need, for each outgoing transition, to look at the destination state to find out the transition label.

It was shown how to implement the compact suffix automaton so that it can be used for large source texts. In such a case we require minimum disk accesses that are in the worst case (the required data are not in disk cache) 100,000 times slower than memory accesses. When using compact suffix automaton, we traverse the resulting data-file forward (we do not go backward). We can also set any size of compact suffix automaton buffer (the part of compact suffix automaton stored in main memory) and thus control space requirements when traversing compact suffix automaton.

7 References

[Bal98] M. Balík. Implementation of DAWG. In J. Holub and M. Šimánek, editors, *Proceedings of the Prague Stringology Club Workshop '98*, pages 26–35, Czech Technical University in Prague, Czech Republic, 1998. Collaborative Report DC–98–06.

[BBE+85] 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(1):31–55, 1985.

[BBE+87] 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(3):578–595, 1987.

[3]The worst case is, when no traversed state is in disk cache, which is unlikely to happen.

[Bel] T. Bell. The Canterbury Corpus. `http://corpus.canterbury.ac.nz/`.

[CH97] M. Crochemore and C. Hancart. Automata for matching patterns. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 2 Linear Modeling: Background and Application, chapter 9, pages 399–462. Springer-Verlag, Berlin, 1997.

[CR94] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.

[Cro86] M. Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.

[CV97a] M. Crochemore and R. Vérin. Direct construction of compact directed acyclic word graphs. In A. Apostolico and J. Hein, editors, *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, number 1264 in Lecture Notes in Computer Science, pages 116–129, Aarhus, Denmark, 1997. Springer-Verlag, Berlin.

[CV97b] M. Crochemore and R. Vérin. On compact directed acyclic word graphs. In J. Mycielski, G. Rozenberg, and A. Salomaa, editors, *Structures in Logic and Computer Science*, number 1261 in Lecture Notes in Computer Science, pages 192–211. Springer-Verlag, Berlin, 1997.

[Dam64] F. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, 1964.

[Fre60] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[Ham50] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950.

[HM00] J. Holub and B. Melichar. Approximate string matching using factor automata. *Theor. Comput. Sci.*, 249(2):305–311, 2000.

[IHS+01] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi. On-line construction of compact directed acyclic word graphs. *Lecture Notes in Computer Science*, 2089:169–180, 2001.

[Kur99] S. Kurtz. Reducing the space requirements of suffix trees. *Softw. Pract. Exp.*, 29(13):1149–1171, 1999.

[Lev65] V. I. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. Algorithms*, 23(2):262–272, 1976.

[Mel04] B. Melichar. Repetitions in text and finite automata. In L. Cleophas and B. W. Watson, editors, *Proceedings of the Eindhoven FASTAR Days 2004*, pages 1–46. TU Eindhoven, The Netherlands, 2004.

[MM90] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327, 1990.

[NR00] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA)*, 5(4), 2000. `http://www.jea.acm.org/2000/NavarroString`.

[Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[UW93] E. Ukkonen and D. Wood. Approximate string matching with suffix automata. *Algorithmica*, 10(5):353–364, 1993.

[Wei73] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.

8 Ing. Jan Holub, Ph.D.

Address:
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo náměstí 13
121 35 Praha 2

E-mail:
holub@felk.cvut.cz

Education:

- Ph.D. in Informatics and Computer Science, DCSE CTU, September 2000, thesis supervisor: prof. B. Melichar
- M.Sc. (Ing.) in Computer Engineering, DCSE CTU, February 1996, thesis supervisor: prof. B. Melichar

Employment:

- 1999–now: Assistant Professor at the Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague

Publications:

1. Holub, J., Smyth, W. F., Wang, S.: Fast Pattern-Matching on Indeterminate Strings. Journal of Discrete Algorithms, Vol. 6, No. 1, Elsevier, pp. 37-50, 2008.
2. Holub, J. (editor): Special Issue of International Journal of Foundations of Computer Science, Vol. 19, No. 1, 2008.
3. Fiala, M., Holub, J.: DCA using Suffix Arrays. In J. A. Storer, M. W. Marcellin (eds.): Proceedings of Data Compression Conference 2008, IEEE Computer Society Press, p. 516, 2008.
4. Holub, J., Smyth, W. F., Wang, S.: Hybrid Pattern-Matching Algorithms on Indeterminate Strings. In Daykin, J., Mohamed, M., Steinhoefel, K. (eds.): London Stringology Day + London Algorithmic Workshop 2006, King's College London Series Texts in Algorithmics, pp. 115-133, 2007.
5. Holub, J. (editor): Special Issue of Journal of Automata, Languages and Combinatorics, Vol. 10, No. 5/6, 2005, (printed in 2007).
6. Holub, J.: Finite Automata Implementations Considering CPU Cache. In Acta Polytechnica, Vol. 47, No. 6, pp. 51-55, 2007.

7. Holub, J., Žďárek, J. (editors): Implementation and Application of Automata, The proceedings of CIAA2007. Czech Technical University in Prague, Prague, July 16-18, 2007, 324 pages, ISBN 978-3-540-76335-2, LNCS 4783, SpringerLink, Springer-Verlag, 2007.

8. Antoniou, P., Holub, J., Iliopoulos, C.S., Melichar, B., Peterlongo, P.: Finding Common Motifs with Gaps using Finite Automata. In O. H. Ibarra, Hsu-Chun Yen (eds.): Proceedings of the 11th International Conference on Implementation and Application of Automata (CIAA2006), National Taiwan University, Taipei, Taiwan, LNCS 4094, Springer-Verlag, pp. 69-77, 2006.

9. Holub, J. (editor): Special Issue of International Journal of Foundations of Computer Science, Vol. 17, No. 6, 2006.

10. Holub, J., Žďárek, J. (editors): Proceedings of the Prague Stringology Conference '06. Czech Technical University in Prague, Prague, September 2006.

11. Holub, J. (editor): Special Issue of International Journal of Foundations of Computer Science, Vol. 16, No. 6, 2005.

12. Holub, J., Šimánek, M. (editors): Proceedings of the Prague Stringology Conference '05. Czech Technical University in Prague, Prague, September 2005.

13. B. Melichar, J. Holub, T. Polcar: Text Searching Algorithms. Textbook for courses ATHENS (Advanced Technology Higher Education Network, Socrates), November 2005.

14. Franěk, F., Holub, J., Rosa, A.: Two factorizations of small complete graphs II: The case of 13 vertices. Journal of Combinatorial Mathematics and Combinatorial Computing, Vol. 51, 2004, pp. 89-94.

15. Holub, J., Šimánek, M. (editors): Proceedings of the Prague Stringology Conference '04. Czech Technical University in Prague, Prague, September 2004.

16. Holub, J., Špiller, P.: Practical Experiments with NFA Simulation. In L. Cleophas, B. W. Watson (eds): Proceedings of the Eindhoven FASTAR Days 2004, invited talk, Technical University of Eindhoven, 2004, pp. 73-95.

17. Franěk, F., Holub, J., Smyth, W. F., Xiao, X.: Computing Quasi Suffix Arrays. In Journal of Automata, Languages and Combinatorics, Vol. 8, No. 4, Otto-von-Guericke University, Magdeburg, 2003, pp. 593-606.

18. Holub, J. (editor): Special Issue of Nordic Journal of Computing, Vol. 10, No. 1, 2003.

19. Holub, J., Smyth, W. F.: Algorithms on Indeterminate Strings. In Miller, M., Park, K. (eds.): Proceedings of the 14th Australasian Workshop on Combinatorial Algorithms AWOCA'03, Seoul National University, Seoul, Korea, 2003, pp. 36-45.

20. Holub, J.: Dynamic Programming—NFA Simulation. Proceedings of the 7th Conference on Implementation and Application of Automata, University of Tours, Tours, France, July 2002, LNCS 2608, Springer-Verlag, pp. 295-300.

21. Holub, J.: Dynamic Programming for Reduced NFAs for Approximate String and Sequence Matching. Kybernetika, Vol. 38(1), 2002, pp. 81-90.

22. Holub, J., Crochemore, M.: On the Implementation of Compact DAWG's. Proceedings of the 7th Conference on Implementation and Application of Automata, University of Tours, Tours, France, July 2002, LNCS 2608, Springer-Verlag, 2003, pp. 289-294.

23. Holub, J.: Bit Parallelism—NFA Simulation. Proceedings of the 6th Conference on Implementation and Application of Automata, University of Pretoria, Pretoria, South Africa, July 2001, LNCS 2494, Springer-Verlag, 2002, pp. 149-160.

24. Holub, J., Iliopoulos, C.S., Melichar, B., Mouchard, L.: Distributed Pattern Matching Using Finite Automata. Journal of Automata, Languages and Combinatorics, Vol. 6(2), Otto-von-Guericke University, Magdeburg, 2001, pp. 191-204.

25. Holub, J., Melichar, B.: Approximate String Matching using Factor Automata. Theoretical Computer Science, Vol. 249 (2), Elsevier Science, 2000, pp. 305-311.

26. Holub, J., Šimánek, M. (editors): Proceedings of the Prague Stringology Club Workshop '99. Czech Technical University in Prague, Prague, July 1999.

27. Melichar, B., Holub, J.: Algorithms for Pattern Matching. In Proceedings of Summer School of Information Systems and Their Applications 1999, Ruprechtov, Czech Republic, September 1999, pp. 69-78.

28. Holub, J.: Simulation of Nondeterministic Finite Automata in Approximate String and Sequence Matching. Research Report DC-98-04, Czech Technical University in Prague, Prague, April 1998, 28 pages.

29. Holub, J., Melichar, B.: Implementation of Nondeterministic Finite Automata for Approximate Pattern Matching. In Proceedings of Third International Workshop on Implementing Automata WIA'98, University of Rouen, France, September 1998, LNCS 1660, Springer-Verlag, Berlin, pp. 92-99.

30. Holub, J., Šimánek, M. (editors): Proceedings of the Prague Stringology Club Workshop '98. Czech Technical University in Prague, Prague, September 1998.

31. Melichar, B., Holub, J.: Pattern Matching and Finite Automata. In Proceedings of Summer School of Information Systems and Their Applications 1998, Ruprechtov, Czech Republic, September 1998, pp. 154-183.

32. Holub, J. (editor): Proceedings of the Prague Stringology Club Workshop '97. Czech Technical University in Prague, Prague, July 1997, 68 pages.

33. Holub, J.: Simulation of NFA in Approximate String and Sequence Matching. Proceedings of the Prague Stringology Club Workshop '97, Czech Technical University in Prague, Prague, July 1997, pp. 39-46.

34. Melichar, B., Holub, J.: 6D Classification of Pattern Matching Problems. Proceedings of the Prague Stringology Club Workshop '97, Czech Technical University in Prague, Prague, July 1997, pp. 24-32.

35. Melichar, B., Holub, J., Mužátko, P.: Languages and Translations. Czech Technical University in Prague, Prague, November 1997, 143 pages, textbook.

36. Holub, J. (editor): Proceedings of the Prague Stringology Club Workshop '96. Czech Technical University in Prague, Prague, August 1996, 83 pages.

37. Holub, J.: Reduced Nondeterministic Finite Automata for Approximate String Matching. Proceedings of the Prague Stringology Club Workshop '96, Czech Technical University in Prague, Prague, August 1996, pp. 19-27.

Professional membership:

- The Prague Stringology Conference (annual conference since 1996)—member of Program Committee and Organizing Committee
- Workshop on Algorithms in Molecular Biology (ALBIO'08)—member of Program Committee
- 13th International Conference on Implementation and Application of Automata (CIAA 2008)—member of Program Committee
- 15th String Processing and Information Retrieval Symposium (SPIRE 2008)—member of Program Committee
- 12th International Conference on Implementation and Application of Automata (CIAA 2007)—Co-Chair of both Program and Organizing Committees
- 11th International Conference on Implementation and Application of Automata (CIAA 2006)—member of Program Committee
- guest editor in International Journal of Foundations of Computer Science, Nordic Journal of Computing, and Journal of Automata, Languages and Combinatorics

Stays abroad:

- February–March, 2007: Research stay at Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada
- July 2002–July 2003: Postdoctoral Fellowship at Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada
- August 1999: Research stay at School of Computing, Curtin University of Technology, Perth, Western Australia
- May–June, 1999: Research stay at Gaspard Monge Institute, University of Marne-la-Vallée, Paris, France
- November–December, 1998: Research stay at Gaspard Monge Institute, University of Marne-la-Vallée, Paris, France