Construction of Aho Corasick Automaton in Linear Time for Integer Alphabets

Shiri Dori*
University of Haifa

Gad M. Landau†
University of Haifa &
Polytechnic University

Abstract

We present a new simple algorithm that constructs an Aho Corasick automaton for a set of patterns, P, of total length n, in $O(n)$ time and space for integer alphabets. Processing a text of size m over an alphabet Σ with the automaton costs $O(m \log |\Sigma| + k)$, where there are k occurrences of patterns in the text.

A new, efficient implementation of nodes in the Aho Corasick automaton is introduced, which works for suffix trees as well.

Key words: Design of Algorithms, String Matching, Suffix Tree, Suffix Array.

1 Introduction

The *exact set matching* problem [7] is defined as finding all occurrences in a text T of size m, of any pattern in a set of patterns, $P = \{P_1, P_2, ..., P_q\}$, of cumulative size n over an alphabet Σ. The classic data structure solving this problem is the automaton proposed by Aho and Corasick [2]. It is constructed in $O(n \log |\Sigma|)$ preprocessing time and has $O(m \log |\Sigma| + k)$ search time, where k represents the number of occurrences of patterns in the text. This solution is suitable especially for applications in which a large number of patterns is known and fixed in advance, while the text varies. We will explain the data structure in detail in Section 2.

The suffix tree of a string is a compact trie of all the suffixes of the string. Several algorithms construct it in linear time for a constant alphabet size [14, 16, 17]. Farach [5] presented a linear time algorithm for integer alphabets. Generalized suffix trees for a set of strings, as defined in [7], are also constructed in time linear to the cumulative length of the strings. Lately, much attention has been paid to the suffix array [6, 13], a sorted enumeration of all the suffixes of a string. The algorithm in [13] constructed this space-efficient alternative to suffix trees in $O(n \log n)$ time, but recently a few $O(n)$ algorithms

*Department of Computer Science, University of Haifa, Mount Carmel, Haifa 31905, Israel, phone: (972-4) 828-8375, FAX: (972-4) 824-9331; email: shiri@cri.haifa.ac.il; Partially supported by the Israel Science Foundation grants 282/01 and 35/05.

†Department of Computer Science, University of Haifa, Mount Carmel, Haifa 31905, Israel, phone: (972-4) 824-0103, FAX: (972-4) 824-9331, email: landau@cs.haifa.ac.il; Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840, USA; Partially supported by the Israel Science Foundation grants 282/01 and 35/05.

were suggested [8, 10, 12]. New developments suggest further refinements; the enhanced suffix array can entirely replace the suffix tree [1] with the same incurred complexities [9]. In [9], a history of the evolution of suffix trees and arrays was relayed.

The linear time construction of the suffix tree and suffix array for integer alphabets was achieved after years of research. It is only natural to extend this development to the third "member" of this data structure family – the Aho Corasick automaton. We answer this challenge and present a new algorithm which constructs the Aho Corasick automaton in $O(n)$ preprocessing time for integer alphabets. The search time remains as it was, $O(m \log |\Sigma| + k)$.

At the heart of our algorithm lies the observation of a close connection between the Aho Corasick Failure function and the index structure of the reversed patterns, be it a suffix tree or an enhanced suffix array. This connection is explained in detail in Section 3.2.

Our algorithm uses the new linear time algorithms to construct suffix arrays and suffix trees for integer alphabets. In this paper, we handle three issues: constructing the Goto trie for all patterns, using the linear time construction of suffix arrays; constructing the Failure function for all pattern prefixes by building a generalized suffix tree, using the linear time construction of suffix trees; and finally, keeping logarithmic search time in each node, using a new, simple approach.

Our approach to keeping logarithmic search time in each node is simple yet innovative. The traditional data structures for a node are either an array of size $|\Sigma|$, which is time efficient but not space efficient; a linked list, which is space efficient but not time efficient; or a balanced search tree, which is considered a heavy-duty compromise and often not practical. We suggest a new, space efficient implementation of the nodes. Each node holds a sorted array whose size is equal to the number of the node's children. The space usage of this data structure is minimal, and search time is kept logarithmic. Clarity and ease of use are also important features that arise from using this approach.

The paper is organized as follows: in Section 2, we define the Aho Corasick automaton. In Section 3, we describe the construction of the automaton in several steps: the Goto function (Section 3.1), the Failure function (Section 3.2), and the combination of the two (Section 3.3). The original Output function, which we have not changed, is mentioned briefly in Section 3.4. We explain the new data structure and show how to maintain the $O(m \log |\Sigma| + k)$ bound on query processing in Section 4, and summarize our results in Section 5.

2 Preliminaries

The *exact set matching* problem [7] is defined as finding all occurrences in a text, T, of any pattern in a set of patterns, $P = \{P_1, P_2, ..., P_q\}$. This problem was first solved by Aho and Corasick [2] by creating a finite state machine to match the patterns with the text in one pass. In other words, their method defined the following functions (slightly redefined in [7]):

- Goto function: an uncompressed trie representation of the set of patterns. Let $L(v)$ denote the label along the path from the root to node v. Then, for each node v, $L(v)$ represents the prefix of one or more patterns; for each leaf u, $L(u)$ represents a pattern. The Goto function is also known as a keyword tree [2, 7].
- Failure function: based on a generalization of the Knuth-Morris-Pratt algorithm [11], the func-

tion is defined in two parts:

  - The Failure function of a string s (which is a prefix of a pattern) is the longest proper suffix of s that is also a prefix of some pattern [2, 7].
  - The Failure link, $v_1 \mapsto v_2$, between two nodes in the Goto trie, links v_1 to a (unique) node v_2 such that the Failure function of $L(v_1)$ is $L(v_2)$ [7].

- Output function: for a node v, the Output function is defined as all patterns which are suffixes of $L(v)$, i.e., end at this node. As shown in the original article [2] and embellished in [7], the Output function for each node v_1 consists of two parts:

  - $L(v_1)$, if it equals a pattern.
  - $Output(v_2)$, where $v_1 \mapsto v_2$ is a Failure link. Note that this is a recursive construction.

These three functions, Goto, Failure and Output, completely define an Aho Corasick automaton, which is no longer an automaton or a state machine in the classic sense. Preprocessing is applied to the set of patterns, and queries can be made on different texts. When a text is processed, we attempt to traverse the trie using the Goto function. For a given character, if no appropriate child exists in the current node, the Failure links are traversed instead, until we find such a node or reach the root. Whenever a node with a non-empty output is reached, all output words must be found. Figure 1 shows an example of an Aho Corasick automaton for a set P.

3 Building the Aho Corasick automaton

Our algorithm to construct the Aho Corasick automaton consists of three main steps: constructing the Goto function, constructing the Failure function, and merging the two. We will now explain each step in detail. We will often use a notation S_P for the concatenation of the patterns in P, with unique endmarkers separating them: $S_P = \$_0\ P_1\ \$_1\ P_2\ \$_2 ...\ \$_{q-1}\ P_q\ \$_q$.

3.1 Building the Goto function

The Goto function is constructed in two steps: sorting the patterns in P and building the trie that represents them.

The patterns can be sorted using suffix arrays, by constructing the suffix array for S_P, thus sorting all suffixes of S_P; out of these, we can filter only the complete words and receive their sorted order. Alternatively, a two-pass radix sort can be employed to sort the strings [3, 15]. The sorting then costs $O(D + |\Sigma|)$, where D represents the minimal amount of characters needed to distinguish among the patterns, which must be smaller than their cumulative length. For the purpose of radix sort, the alphabet can also be considered to be bounded by the size of the patterns, so the sorting phase takes $O(D + |\Sigma|) = O(n)$.

Once the patterns are sorted, the trie is built simply by traversing and extending it for each pattern in the sorted order. We keep a list of children for each node, with the invariant that they are ordered alphabetically. For each pattern in turn, we will attempt to traverse the trie, knowing that if the next

character exists, it will be at the tail of the list of children in the current node. If not, we may create it and insert it to the tail, thus keeping the invariant. Once a character was not found, we can extend the trie for the rest of the pattern from this point on. This is nearly identical to the original method employed by Aho and Corasick [2], with the sole difference that they kept arrays of size $|\Sigma|$ in each node, whereas we keep sorted lists.

Time Complexity. Sorting takes $O(n)$ time for integer alphabets using a suffix array for S_P [8]. As for the trie, the work for each character is either traversing a link at a known position or creating a new node; both take up $O(1)$. The work for each pattern is proportional to its length, so for the entire set it will be $O(n)$.

3.2 Building the Failure Function

Having constructed the trie representing the Goto function for P, we turn to construct the Failure links on top of it. We chose to describe the algorithm using a suffix tree for simplicity, but an enhanced suffix array can easily replace it [1, 9]. Implementation considerations can be taken into account to choose among the two.

We have defined S_P as the string representing the patterns in P. We define T^R to be the suffix tree of the reverse of this string, that is, the suffix tree of $(S_P)^R$. The properties of trees for reverse strings were discussed in [4]. Also, Gusfield [7] has discussed generalized suffix trees for a set of strings. In Section 6.4 of [7], he showed how to construct this tree without creating synthetic suffixes that combine two or more patterns. Therefore, \$ signs never appear in the middle of edges of the suffix tree, and only mark the end of a label.

Observe the tree in Figure 2. This is T^R for the same patterns seen in Figure 1. The following properties of T^R are relevant to our discussion:

- For each node v in the trie representing the Goto function, there exists a unique node u in T^R which represents its reverse label, that is, $L(v) = L(u)^R$. For example, in Figure 1, node 16, labeled "their", has its match in Figure 2, with node m, labeled "rieht".
- When considering a node u_1 and its ancestor u_2 in T^R, the label of u_2 is a prefix of the label of u_1. Since these are reverse labels of the original string, the original label of u_2 is a suffix of the original label of u_1.
- When a node u in T^R is marked by a \$, this means that its original label begins with a \$; i.e., the reverse label of u is a prefix of some pattern in P.
- In Lemma 1 we will show that all nodes in T^R which are marked by a \$, and only those, have a corresponding node in the trie.

Observe the Failure links in Figure 1. For example, the Failure function of node 10, "iris", is node 11, "is", as seen by the dotted arrow between the two nodes. Now, let us turn our attention to their corresponding nodes in T^R. Here, node n, "si" (the reverse of "is"), is the nearest ancestor of node o, "siri" (the reverse of "iris"), and it is also marked by a \$. In other words, "is" is the longest proper suffix of "iris" which is also a prefix of some pattern: exactly the definition of the Failure function. Also, notice node 8, labeled "ir", which corresponds to node l, "ri", in T^R. The Failure function of

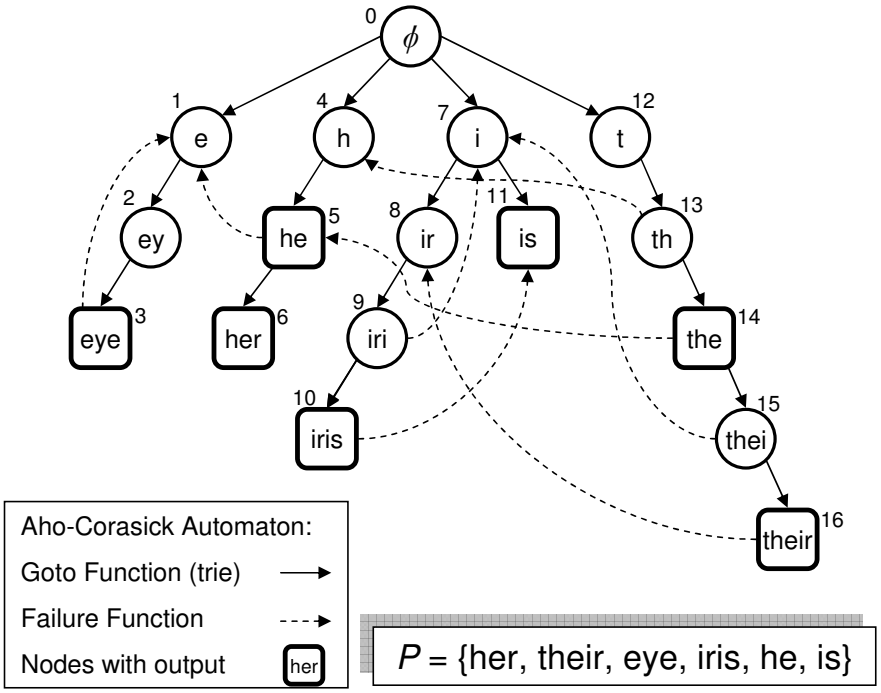

Figure 1: An Aho Corasick automaton for the set P. The dotted arrows represent Failure links. For brevity's sake, we did not include Failure links that point to the root. Nodes that have a non-empty output are emphasized.

"ir" is not "r", since "r" is not a prefix of any pattern. Indeed, node j representing "r" in T^R is not marked by a \$ sign, and additionally, there is no node in the original trie representing "r". Therefore the Failure function of "ir" is ϕ (node z), its nearest ancestor in T^R with a \$ sign.

Definition 1 *Consider node u_1 in T^R. We define the proper ancestor of u_1, $u_2 = PA(u_1)$, as the closest ancestor of u_1 in T_R which is marked by a \$.*

PA can be computed for every node in T^R by a simple preorder traversal of T^R.

Now, consider nodes v_1 and v_2 in the trie, so that there exists a Failure link $v_1 \mapsto v_2$. Let u_1, u_2 be the corresponding nodes in T^R. Then $u_2 = PA(u_1)$. Our algorithm is based on this property. Now that we've found this information in T^R, we would like to relay it to the nodes in the trie. Each node in the trie must be connected to its corresponding node in T^R, and we will handle this in Section 3.3.

Time Complexity. The suffix tree T^R built in this step can be constructed in linear time, either using Farach's algorithm [5], or indirectly using suffix arrays and Longest Common Prefix values [5, 8]. The tree is then traversed once in order to find PA, but in a preorder fashion and not as a search, therefore the traversal is also linear.

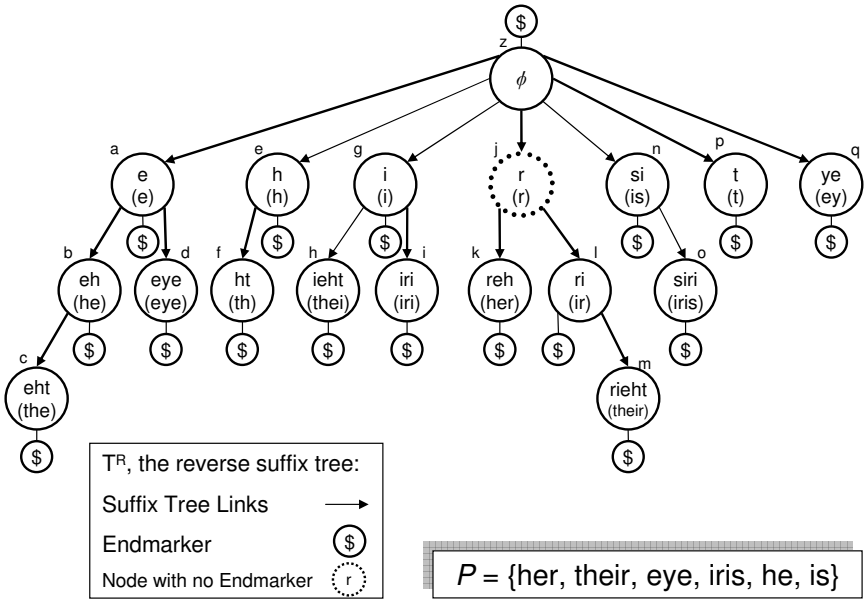

Figure 2: T^R for the patterns of Figure 1. Since the labels are reversed, we included the original label in parenthesis. For example, node c representing "eht" in T^R corresponds to node 14 representing "the" in the trie of Figure 1. The endmarker, \$, marks one of two options: for a leaf, its label ends with a \$; for an internal node, it has a son which represents this label ending with a \$. If we treat an internal node as marked by a \$, this is actually shorthand for noting the second case.

3.3 Integrating Failure with Goto

We have shown how to obtain $PA(x)$ for each node in T^R, but we must integrate this information with the trie representing the Goto function. We do this once, during construction, recording the Failure links in the trie nodes, so that the trie is a self-contained Aho Corasick automaton. The integration is based upon the ability to infer, for each node in the trie, its corresponding \$-marked node in T^R, and vice versa. This is achieved through the actual string S_P, in the following steps:

- During or after construction of the Goto function, we compute the following: for each character in S_P, we will keep a pointer to its representative node in the trie – the one visited or constructed when this character was dealt with. An example of this is shown in Figure 3(a).
- As part of the construction of any suffix tree, and so of T^R, each node records the first and last indices of one of the appearances of its label in the string, $(S_P)^R$ in our case. From these indices we can compute the corresponding appearance of the original label in S_P: for \$-marked nodes, the label's last index in S_P represents a prefix of some pattern. An example of this appears in Figure 3(b) and (c).
- Combining the information we have about S_P, we map a trie node for each \$-marked node in T^R and vice versa. Lemma 1 below shows that this is a one-to-one mapping, as shown also in Figure 3(d).

- We use this mapping to record the information of PA, garnered from T^R, among nodes in the trie as Failure links. For each node v_1 in the trie, we:
  - Find node u_1 in T^R which corresponds to v_1.
  - Find $u_2 = PA(u_1)$.
  - Find node v_2 in the trie which corresponds to u_2.
  - Create a Failure link, $v_1 \mapsto v_2$, among the trie nodes.
- Once these steps are completed, one can discard T^R entirely.

Lemma 1 *There exists a one-to-one mapping between \$-marked nodes in T^R and nodes in the trie.*

Proof: Any \$-marked node u in T^R represents a prefix of some pattern in S_P, and thus has a unique corresponding node v in the trie. Now we prove the other direction: a label of any node v in the trie represents a prefix of a pattern. Hence, in S_P the label is preceded by a \$, meaning that its reversed label in $(S_P)^R$ is followed by a \$. Therefore, there is a single \$-marked node u in T^R corresponding to v. Note that there can be \$ signs only at the end of a node's label in a generalized suffix tree [7].

Time Complexity. Constructing the mapping takes constant time for each node, and the creation of Failure links consists of a traversal of the trie. The entire phase takes $O(n)$ time.

3.4 Building the Output Function

The Output function can be computed, just as in [2], during the computation of the Goto and Failure functions, with a traversal on the trie. Gusfield refines these computations and explains them in detail in Section 3.4.6 of [7]. The recursive manner of the Output function is exemplified in Figure 1, where node 14 labeled "the" has an output of "he", due to its Failure link to node 5.

Time Complexity. Computing the Output function consists of some constant-time processing during the construction of the Goto function, and a traversal of the trie with the Failure links, so it costs $O(n)$ time.

4 Processing Queries

It is a generally accepted fact that, for non-negligible alphabets, there exists a tradeoff between the space consumed and the time required to search in each node, depending on the branching method in the nodes [7, 9]. If an array of size $|\Sigma|$ is used in each node, the time to search will be constant. If only a list of actual sons is held, the cumulative space for the tree will be proportional to the number of nodes, but search time in each node will rise to $O(|\Sigma|)$. Finally, the generally accepted, heavy-duty solution, is to hold a balanced search tree in each node, thus limiting the space needed in the tree overall, but increasing both insertions and search time to the tree by a factor of $\log |\Sigma|$. Here, we suggest a new data structure that is at once simple and efficient - a sorted array, built to size.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
S_P	$\$_0$	h	e	r	$\$_1$	t	h	e	i	r	$\$_2$	e	y	e	$\$_3$	i	r	i	s	$\$_4$	h	e	$\$_5$	i	s	$\$_6$
Trie		4	5	6		12	13	14	15	16		1	2	3		7	8	9	10		4	5		7	11	

(a)

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
S_P^R	$\$_6$	s	i	$\$_5$	e	h	$\$_4$	s	i	r	i	$\$_3$	e	y	e	$\$_2$	r	i	e	h	t	$\$_1$	r	e	h	$\$_0$
n-i	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(b)

T^R node	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q
Index in S_P	10-**11**	0-**2**	4-**7**	10-**13**	0-**1**	4-**6**	14-**15**	4-**8**	14-**17**	3, 9, 16	0-**3**	14-**16**	4-**9**	22-**24**	14-**18**	4-**5**	10-**12**
2nd option		19-**21**			19-**20**		22-**23**			(no $)							

(c)

T^R node	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q
Trie node	1	5	14	3	4	13	7	15	9	none (no $)	6	8	16	11	10	12	2

(d)

P = {her, their, eye, iris, he, is}

Figure 3: Integrating the Goto and Failure functions shown in Figures 1 and 2. **(a)** S_P and the trie nodes corresponding to each character. **(b)** $(S_P)^R$ and corresponding indices in S_P. **(c)** T^R nodes and the indices in S_P corresponding to their labels. The first index is always a \$, and the last index (in bold) is the relevant one. A label may appear more than once in the string, e.g. node b representing "he" has two possible mappings to the string. **(d)** T^R nodes and their corresponding trie nodes, computed from parts (a) and (c) combined. For example, the label "thei", which is a prefix of the pattern "their", spans indices 4-8 in S_P (including \$); its last index, 8, maps to trie node 15 as shown in (a). Its reverse label, "ieht", is represented by node h, as shown in (c); combining these, we map node 15 to node h in (d). Note that even if several options exist in (c), they all map to the same trie node. Also note that node j, which is not marked by a \$, has no corresponding node in the trie.

Constructing the Goto trie as described in our algorithm will yield an alphabetically ordered linked list of children in each node. Hence, we can visit every node and allocate an array in size equal to the number of its sons, then transfer the list of sorted sons to this array, in $O(n)$ total time. The array of children can then be searched in $O(\log(\#of sons))$, which is in the worst case $O(\log|\Sigma|)$.

It is important to note that this simple, yet novel approach can be used also when constructing a suffix tree indirectly from a suffix array and Longest Common Prefix values. This construction [5, 8] also proceeds in lexicographic fashion, and so can likewise use this approach, and keep linear space. At the same time, the new data structure will be simpler to use than the traditional balanced search tree in each node, while also reducing the constants involved.

Time Complexity. Traversing the automaton costs $O(m\log|\Sigma|)$ time for a text of length m. The Output function allows a constant time spent for each occurrence of any pattern in the text, so finding the output patterns will cost $O(k)$ where k denotes the number of occurrences of patterns from P in the text. Hence, the entire complexity of query processing is $O(m\log|\Sigma|+k)$.

5 Summary

We have presented a new algorithm, which is simple to understand and implement, that constructs the Aho Corasick automaton in $O(n)$ time. This algorithm is an addition to a growing group of linear algorithms for string problems over integer alphabets.

We have brought forth an interesting observation regarding the Failure function of the Aho Corasick automaton [2]. This Failure function is intimately connected with the generalized suffix tree of the reverse patterns, and this has provided the basis for our algorithm.

We suggested a new, simple data structure for maintaining node structure, which is the sorted array of minimal size. This data structure is also applicable to other indexing structures that are built lexicographically, such as the suffix tree constructed from a suffix array. Using our novel approach can improve these data structures significantly. Both space and time complexities are kept at the accepted levels, while simplifying implementation and reducing the constants involved.

Our algorithm constructs the Aho Corasick automaton indirectly, using other algorithms for constructing suffix trees and arrays. Therefore, it is important to note that the algorithm's space and time usage depends on the other algorithms used, and may be quite large in practice. It remains an open problem to construct the Aho Corasick automaton directly in $O(n)$ time.

Acknowledgments. We would like to thank Maxime Crochemore, Raffaele Giancarlo, Moshe Lewenstein, Kunsoo Park and Pierre Peterlongo for helpful discussions.

References

[1] Abouelhoda, M.I., S. Kurtz and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004.

[2] Aho, A.V. and M.J. Corasick. Efficient string matching. *Comm. ACM*, 18(6):333–340, 1975.

[3] Andersson, A. and S. Nilsson. A New Efficient Radix Sort. *Proc. 35th IEEE Symposium on Foundations of Computer Science*, pp. 714–721, 1994.

[4] Chen, M.T. and J. Seiferas. Efficient and Elegant Subword-Tree Construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words,* pp. 97–107. Springer-Verlag, 1985.

[5] Farach, M. Optimal suffix tree construction with large alphabets. *Proc. 38th IEEE Symposium on Foundations of Computer Science*, pp. 137–143, 1997.

[6] Gonnet, G., R. Baeza-Yates and T. Sinder. New Indices for text: PAT trees and PAT arrays. In W.B. Frakes and R. Baeza-Yates (Eds.), *Information Retrieval: Data Structures and Algorithms.* Prentice Hall, 1992.

[7] Gusfield, D. Algorithms on strings, trees, and sequences. Cambridge University Press, 1997.

[8] Kärkkäinen, J. and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 03)*, pp. 943–955, 2003. LNCS 2719.

[9] Kim, D.K., J.E. Jeon and H. Park. Efficient Index Data Structure with the Capabilities of Suffix Trees and Suffix Arrays for Alphabets of Non-negligble Size. In A. Apostolico and A. Melucci (Eds.), *String Processing and Information Retrieval (SPIRE 04)*, pp. 138–149, 2004. LNCS 3246.

[10] Kim, D.K., J.S. Sim, H. Park and K. Park. Linear-time construction of suffix arrays. *Symp. Combinatorial Pattern Matching (CPM 03)*, pp. 186–199, 2003. LNCS 2676.

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

[12] Ko, P. and S. Aluru. Space-efficient linear-time construction of suffix arrays. *Symp. Combinatorial Pattern Matching (CPM 03)*, pp. 200–210, 2003. LNCS 2676.

[13] Manber, U. and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

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

[15] Paige, R. and R.E. Tarjan. Three Partition Refinement Algorithms. *SIAM J. Comput.*, 16(6): 973–989, 1987.

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

[17] Weiner, P. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pp. 1–11, 1973.