T U M

I N S T I T U T F Ü R I N F O R M A T I K

Computing Suffix Links for Suffix Trees and Arrays

Moritz G. Maaß

TUM-I0514
Juli 05

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-07-I0514-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Computing Suffix Links for Suffix Trees and Arrays

Moritz G. Maaß
maass@informatik.tu-muenchen.de

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching, Germany

Abstract

We present a new and simple algorithm to reconstruct suffix links in suffix trees and suffix arrays. The algorithm is based on observations regarding suffix tree construction algorithms. With our algorithm we bring suffix arrays even closer to the ease of use and implementation of suffix trees.

Keywords: Pattern Matching, Suffix Links, Suffix Trees, Suffix Arrays

1 Introduction

Historically, suffix links were an invention to facilitate linear-time[1] construction of suffix trees [Wei73, McC76, Ukk95]. It has since been discovered that suffix links have uses of there own (e.g., [Gus97]), most notable for the computation of matching statistics and approximate pattern matching [CL94]. Other applications are finding tandem repeats in linear time [GS04] or the construction of DAWGs [Gus97]. Due to the large size of the suffix tree, suffix links are often discarded or not even constructed. Giegerich and Kurtz [GKS03] have proposed a very space efficient method for top-down construction of suffix trees that does not use suffix links. In Farach's construction method for large alphabets suffix links are not used either [Far97]. Furthermore, recent developments have made the suffix array [MM93] a much more interesting data structure. Kasai et al. [KLA+01] have shown how the suffix array can be used to simulate a bottom-up suffix tree traversal and how to compute the longest common prefix information in linear time (see also [Man04]). Abouelhoda et al. [AKO02, AOK02, AKO04] have enhanced the suffix array so that it can be used with the same asymptotically optimal time bounds as suffix trees in exact matching and other applications. In [AKO04] two methods for suffix link reconstruction are proposed, one with linear-time complexity using (complex) lowest common ancestor (LCA) data structures (see, e.g., [BFC00, BFCP+01, Sad02]), and another simpler one that has complexity $O(n \log n)$. Kim et al. [KJP04] use suffix links on the enhanced suffix array to merge two suffix arrays in linear time for integer alphabets. They also give a linear-time algorithm that reconstructs suffix links. The algorithm does not need constant time LCA data structures. On the other hand, it uses $2n$ lists and bucket sorting, and it is therefore less space efficient than our algorithm, which uses only one additional integer array.

[1] We assume a uniform cost model throughout this paper.

Linear-time algorithms for suffix array construction have been introduced by Kärkkäinen and Sanders [KS03], Kim et al. [KSPP03], Ko and Aluru [KA03]. For practical use, Larsson and Sadakane [LS99], Manzini and Ferragina [MF04], and Burkhardt and Kärkkäinen [BK03] have presented some very fast – but asymptotically non-linear – algorithms which seem to outperform the linear ones (see [ARS+04, PST05] but also [LP04]). Grossi and Vitter have introduced the compressed suffix array [GV00] and another succinct representation, the FM-index, has been developed by Ferragina and Manzini [FM00]. Based on the compressed suffix arrays Sadakane describes a succinct implementation of suffix trees that uses linear space (in bit complexity) and also offers suffix links [Sad04]. The drawback of the compressed suffix arrays is that their practical performance is much worse than that of normal suffix arrays. From the empirical studies in [SS01] and [Kai04] one can expect an increase in the running time by a factor of twenty when comparing normal to compressed suffix arrays.

Our contribution is a very simple, easy-to-implement, and efficient algorithm to reconstruct suffix links on suffix trees and (enhanced) suffix arrays. Under the uniform cost model the algorithm has linear time and space complexity. It is much simpler than the algorithms based on LCA computation because it only does two simple depth first search (DFS) traversals of the tree structure. It is alphabet independent and can thus be used with integer alphabets, for instance, together with Farach-Coltons's suffix tree construction algorithm. Furthermore, it can be seen as a simple enhancement of the enhanced suffix array, making the algorithm [AKO04] run in linear time without the need for LCA or range minimum query (RMQ) data structures.

2 Algorithm

We assume that the reader has basic knowledge in suffix trees, i.e., for an easy understanding, the reader should know the suffix tree construction algorithm by Ukkonen [Ukk95].

In the following, let Σ be an arbitrary alphabet. Note that we do not require a finite alphabet. Let Σ^* denote the set of all finite strings over Σ (including the empty string ε). Let $t = t_1 \cdots t_n \in \Sigma^n$ be a string of length $|t| = n$. If $t = uvw$ with $u, v, w \in \Sigma^*$ then u is a prefix, v a substring, and w a suffix of t. We define the i-th suffix $\mathrm{suff}_i(t) = t_i \cdots t_n$. Following Giegerich and Kurtz [GK97] we define a Σ^+-tree T as a rooted, directed tree with edge labels from Σ^+. For each $x \in \Sigma$, every node in T has at most one outgoing edge whose label starts with x. A Σ^+-tree T is called compact, if all nodes are either the root, leaves or branching. For a node p let $u \in \Sigma^*$ be the string that is constructed by concatenating all edge labels on the path from the root to the node p. We define the path of p as $\mathrm{path}(p) = u$ and the string depth of p as $\mathrm{depth}(p) = |u|$. For a Σ^+-tree T, let its word set $\mathrm{words}(T)$ be all strings u for which there exists a node p with $\mathrm{path}(p) = uv$ for some $v \in \Sigma^*$. The suffix tree $\mathrm{CST}(t)$ of a string t is defined as the compact Σ^+-tree T with $\mathrm{words}(T) = \{u | u \text{ is a substring of } t\}$. For each leaf p of T we define the leaf index $\mathrm{lindex}(p)$ of p to be i if $\mathrm{path}(p) = \mathrm{suff}_i(t)$, i.e., p represents the i-th suffix of t. For node p let $\mathrm{leaves}(p)$ be the leaves in the subtree rooted at p. We let $\perp$ stand for an undefined value (as an undefined value for a pointer to a node).

A suffix link is an auxiliary edge of a Σ^+-tree T. A suffix link points from a node p to a node q which by $\mathrm{path}(q)$ represents the shortest (proper) suffix of $\mathrm{path}(p)$ in T. In suffix trees, usually only suffix links for inner nodes are added. Here, we have the property that $\mathrm{path}(p) = x\mathrm{path}(q)$ for some character $x \in \Sigma^+$.

In the following let T be a suffix tree $\mathrm{CST}(t)$ for string t of length n. Our algorithm constructs suffix links solely based in the structure of the suffix tree and the leaf indices, therefore the size of the alphabet does not matter. The intuitive idea is to construct an array A of size n that contains the branching nodes in the same order as they are encountered by a suffix tree construction algorithm of McCreight [McC76] or Ukkonen [Ukk95]. We can then almost reconstruct the suffix links by setting for each node in A the suffix link to the successor in A. This takes care of almost all cases but those where the "active prefix" [Ukk95] has grown. Thus, we modify the idea to find for each node the index of the leaf that has "caused" the node. From there we find the corresponding branch for the next leaf because we know its leaf index and the branch depth.

For a node q of T let its minimal index be the minimal value of a leaf index of a leaf in the subtree rooted at q. For an inner node p of T let S be the set of minimal indices of its children. We define $\mathrm{cause}(p)$ as the second smallest element in S. Formally,

$$\mathrm{cause}\,(p) =_{def} \min_{2\ q \text{ is a child of } p} \left\{ i \;\middle|\; i = \min_{r \in \mathrm{leaves}(q)} \mathrm{lindex}\,(r) \right\} \quad ,$$

where $\min_2 I$ yields the second smallest element in the set I. Further, for a leaf p we define $\mathrm{branch}(p, d)$ as the ancestor of p at string depth d, i.e.,

$$\mathrm{branch}\,(p, d) =_{def} \begin{cases} q & \text{, if } q \text{ is an ancestor of } p \text{ and } \mathrm{depth}\,(q) = d \\ \perp & \text{, otherwise.} \end{cases}$$

The correctness of our algorithm follows from the next lemma. Note that for every node p of a suffix tree with $\mathrm{path}(p) = xu$ there exists a node q with $\mathrm{path}(q) = u$ to where the suffix link of p will be pointed (see, e.g., Giegerich and Kurtz [GK97]).

Lemma 1 (Suffix Links). *Let p be a non-leaf node of the suffix tree T for a string $t \in \Sigma^n$. The suffix link of p is $\mathrm{branch}(q, \mathrm{depth}(p) - 1)$, where q is the leaf with $\mathrm{lindex}(q) = \mathrm{cause}(p) + 1$.*

Proof. Let $xu = \mathrm{path}(p)$, $u \in \Sigma^*$, $x \in \Sigma$. The suffix link of p must point to a node r with $\mathrm{path}(r) = u$. By definition, $\mathrm{cause}(p)$ is a leaf index from a leaf in the subtree of p. Thus, xu is a prefix of $\mathrm{suff}_{\mathrm{cause}(p)}(t)$ and u is a prefix of $\mathrm{suff}_{\mathrm{cause}(p)+1}(t)$. The length of u is $\mathrm{depth}(p) - 1$. Hence, u is a prefix of $\mathrm{suff}_{\mathrm{lindex}(q)}(t)$ of length $\mathrm{depth}(p) - 1$ and u is represented by an ancestor $r = \mathrm{branch}(q, \mathrm{depth}(p) - 1)$ of q with depth $\mathrm{depth}(r) = \mathrm{depth}(p) - 1$. □

Pseudo code for the algorithm is given in Figure 1. The complexity of the algorithm is given as follows.

Theorem 1 (Correctness and Complexity). *Algorithm $\mathsf{main}(T)$ correctly computes the suffix links for the suffix tree T in time and space $O(n)$.*

Proof. The correctness follows directly from Lemma 1 and the fact that for any two nodes p and q $\mathrm{cause}(p) \neq \mathrm{cause}(q)$. The latter is easy to see as we take the second smallest value from the smallest values from children below. This value is never passed on to a parent.

The complexity follows directly from the algorithm. We perform two DFSs and take constant time per node. The space used is that for two arrays of size at most n and the (call) stack (also of size at most n). □

```
prepare(p, A)
  if p is a leaf then
    Let i be the leaf index of p
    return(i)
  else
    Let d be the depth of p
    min_1 := n + 1
    min_2 := n + 1
    for all children q of p do
      m = prepare(q, A)
      if m < min_1 then
        min_2 := min_1
        min_1 := m
      else if m < min_2 then
        min_2 := m
      end if
    end for
    Set A[min_2 + 1] := p
    return(min_1)
  end if
```

```
compute(p, A, B)
  if p is a leaf then
    Let i be the index of p
    if A[i] ≠ ⊥ and A[i] is not the root then
      Let d be the depth of A[i]
      Set suffix link of A[i] to B[d − 1]
    end if
  else
    Let d be the depth of p
    Set B[d] := p
    for all children q of p do
      compute(q, A, B)
    end for
  end if
```

```
main(T)
  Let r be the root and h the height of T
  Let A be an array of size n initialized to ⊥
  Let B be an array of size h initialized to ⊥
  prepare(r, A)                //bottom-up traversal
  compute(r, A, B)             //top-down traversal
```

Figure 1: Pseudo code for the suffix link reconstruction algorithm. Array A corresponds to cause through the equation $A[\mathrm{cause}(p) + 1] = p$ and array B corresponds to branch through the equation $B[d] = \mathrm{branch}(p, d)$. In $\mathsf{prepare}(p, A)$ we take the minimal leaf indices for each subtree and compute the minimum (which we pass back) and the second minimum (which we store in A). In $\mathsf{compute}(p, A, B)$ we compute branch in B on the fly and use the cause values in A to set the suffix links.

3 Implementation Issues

For suffix trees the implementation is straight forward. For the case that the depth of a node is not stored explicitly, we need an additional array AH to accompany A for storing the string depth of the nodes in A. The height of the tree – if not readily available – can be computed during the first DFS with only small modifications. It is not needed for the asymptotic result because we can simply use n as an upper bound when creating the array B. Since the height of a suffix tree is $O(\log n)$ on average, less memory will be used in practice. Note also, that it might be possible to store the depth in array AH in a smaller field (i.e., a byte).

For suffix arrays we can use the method of Kasai et al. [KLA+01] for the bottom-up traversal ($\mathsf{prepare}(r, A)$). For the top-down traversal ($\mathsf{compute}(r, A, B)$) we need the additional data structures introduced by Abouelhoda et al. [AKO02]. Normally, a suffix tree node is identified by two borders. It is more practical to have a single identifier. This can be generated in an additional array Id through a top-down traversal. For each interval $[l, r]$ we store either l in $\mathsf{Id}[r]$ or r in $\mathsf{Id}[l]$. Using a single bit this can be marked accordingly and we can use the index in Id as an identifier for $[l, r]$. Observe that in a top-down traversal for each node (interval) either l or r is not yet used: The child intervals are always smaller than the parent intervals. If no index were free, we would have used the left and right border for two ancestor intervals $[l, r']$ and $[l', r]$. W.l.o.g., let $[l, r']$ be a child of $[l', r]$, then $r \geq r' \geq r$, i.e.,

$r = r'$. This is a contradiction to $[l, r']$ being an ancestor of $[l, r]$.

Thus, we use an array Id for node (interval) identifiers and an array Sl for suffix links. Using the enhanced suffix array data structures [AKO04] it is also possible to retrieve the depth of a node in constant time. The suffix link creation can thus be implemented to use only one additional temporary array during construction (plus some stack size). If all arrays are implemented naively, the total structure takes five integers (sa, lcp, childtab, Id, Sl) plus one integer (A) per text character and a logarithmic amount of space during construction (the average size of B and the stack). The structure is equivalent to a suffix tree, whose most efficient implementation also takes twenty bytes in the worst case [Kur99]. A more elaborate version would take advantage of the fact that most values in the arrays are small or can be made small by storing relative distances. Abouelhoda et al. [AKO04] report that in this way it is possible to reduce the size of lcp and childtab to one byte per character. The same is possible for Id. As a result we get an implementation using eleven bytes per text character plus some small (i.e., logarithmic in n) amount of additional space. The latter is comparable even to the worst-case size of the suffix tree data structure described in [GKS03].

We conducted some simple testing to compare the above described enhanced suffix array (ESA) with the currently most memory efficient linear-time suffix tree data structure (ILLI) by Kurtz [Kur99]. We used Ukkonen's algorithm [Ukk95] for suffix tree construction and Manzini and Ferragina's algorithm [MF04] for suffix array construction. The latter is not asymptotically linear but has a very good performance in practice [Maa05] (better than asymptotically linear algorithms).

For our tests we used a set genetic sequences available via GenBank (which have the identifiers NC_001460, NC_001454, NC_004001.2 NC_002067, NC_001405, NC_003266, U47924, AC002397, L43967, NC_000912, BA000008, AE002161, NC_000922, NC_003098.1) and the Calgary Corpus[2]. The cumulated results measured on an AMD Athlon XP1800+ with 1544.732 MHz and 1 GB of main memory are shown in Figure 2. With all its new gadgets the suffix array has become fat, fatter even than the suffix tree when we take a look at the maximal memory used during construction. On the other hand, the suffix array has become faster and, as a result, comparable to the suffix tree in our implementation. Only when including the highly repetitive file "pic" from the Calgary Corpus the coding scheme leads to large time and size penalties. We further witness the effect that a larger alphabet size results in a smaller size of the suffix tree and a larger construction time (due to the linked list implementation). As a result, we find that the difference between using a suffix tree or a suffix array has become very small. The rule of thumb that suffix arrays are smaller but slower does not count any more if one wants "full functionality".

4 Conclusion

The last years have seen a major change in the focus of the suffix data structures. The suffix array is becoming more and more the most adequate data structure being preferred to the suffix tree. This is mostly due to the demand for indexing huge texts. Suffix trees and arrays have converged towards each other as a result of the space reduction of suffix trees and the running time enhancement of suffix arrays. We hope to have added another small part to this convergence process. As the difference between both data structures has become very small, it would be interesting whether one could describe their relation and derive fundamental differences that come from the two different concepts of using intervals and nodes.

[2] Available from `ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus`.

Algorithm	Input	Average time construction	search	traversal	Average ratio memory-usage/input-size maximum	index-only
ESA	GB	1.57 s	1.25 s	0.77 s	16.13	12.10
ILLI	GB	1.37 s	0.95 s	0.78 s	13.07	13.07
ESA	CC	11.49 s	1.38 s	0.56 s	16.72	12.48
ILLI	CC	0.48 s	0.39 s	0.24 s	9.97	9.97
ESA	CC\pic	0.35 s	0.29 s	0.22 s	16.41	12.27
ILLI	CC\pic	0.48 s	0.33 s	0.22 s	9.99	9.99

Figure 2: Comparison of the enhanced suffix array and the space efficient suffix tree with genomic data (GB) and the Calgary Corpus (CC). The table gives average values. After the index was constructed we performed 1000 random searches with words from the index text to test the search performance and 100 matching statistic like traversals with the text itself to test the suffix link performance. The maximum memory usage per input character is met during construction. After construction some memory could be freed for the suffix array. Thus, the last column gives the operating size per input character. Note that the text itself contributes another byte to the size. The Calgary Corpus contains a file "pic" with a very high redundancy which lead to a lot of nodes with a depth greater than 255 (the size fitting one byte). The simple size reduction scheme for the lcp array does not work here as more than half of the values stored were larger than 255. This resulted in an extra table with four times the size of the small lcp array.

References

[AKO02] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *Proc. 2nd Workshop on Algorithms in Bioinformatics (WABI)*, volume 2452 of *Lecture Notes in Computer Science*, pages 449–463. Springer, 2002.

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2:53–86, 2004.

[AOK02] Mohamed Ibrahim Abouelhoda, Enno Ohlebusch, and Stefan Kurtz. Optimal exact string matching based on suffix arrays. In A. H. F. Laender and A. L. Oliveira, editors, *Proc. 9th Int. Symp. on String Processing and Information Retrieval (SPIRE)*, volume 2476 of *Lecture Notes in Computer Science*, pages 31–43. Springer, 2002.

[ARS+04] Antonitio, P. J. Ryan, W. F. Smyth, Andrew Turpin, and Xiaoyang Yu. New suffix array algorithms – linear but not fast? In *Proc. 15th Australasian Workshop Combinatorial Algorithms*, pages 148–156, 2004.

[BFC00] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proc. Latin American Theoretical Informatics (LATIN)*, pages 88–94, May 2000.

[BFCP+01] Michael A. Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Least common ancestors in trees and directed acyclic graphs, 2001.

[BK03] Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. 14th Symp. on Combinatorial Pattern Matching (CPM)*, volume 2676 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2003.

[CL94] William I. Chang and Eugene L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12:327–344, 1994.

[Far97] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 137–143, Miami, Florida, USA, October 1997. IEEE. Martin Farach-Colton.

[FM00] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. 41st IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

[GK97] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19:331–353, 1997.

[GKS03] Robert Giegerich, Stefan Kurtz, and Jens Stoye. Efficient implementation of lazy suffix trees. *Software – Practice and Experience*, 33:1035–1049, 2003.

[GS04] Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Computer and System Sciences*, 69:525–546, 2004.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Comp. Science and Computational Biology*. Cambridge University Press, 1997.

[GV00] Roberto Grossi and Jeffry Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd ACM Symp. on Theory of Computing (STOC)*, pages 397–406, 2000.

[KA03] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Proc. 14th Symp. on Combinatorial Pattern Matching (CPM)*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2003.

[Kai04] Hon Wing Kai. *On the Construction and Application of Compressed Text Indexes*. PhD thesis, University of Hong Kong, 2004.

[KJP04] Dong Kyue Kim, Jeong Eun Jeon, and Heejin Park. Merging suffix arrays in linear time for integer alphabets. In *Proc. 15th Australasian Workshop Combinatorial Algorithms*, 2004.

[KLA$^+$01] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In A. Amir and G.M. Landau, editors, *Proc. 11th Symp. on Combinatorial Pattern Matching (CPM)*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2001.

[KS03] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proc. 30th Int. Colloq. on Automata, Languages and Programming (ICALP)*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003.

[KSPP03] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays (extended abstract). In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Proc. 14th Symp. on Combinatorial Pattern Matching (CPM)*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199. Springer, 2003.

[Kur99] Stefan Kurtz. Reducing the space requirement of suffix trees. *Software – Practice and Experience*, 29(13):1149–1171, 1999.

[LP04] Sunglim Lee and Kunsoo Park. Efficient implementation of suffix array construction algorithms. In *Proc. 15th Australasian Workshop Combinatorial Algorithms*, pages 64–72, 2004.

[LS99] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, Dept. of Comp. Science, Lund University, Sweden, 1999.

[Maa05] Moritz Maaß. Matching statistics: Efficient computation and a new practical algorithm for the multiple common substring problem. *To be published in Software – Practice and Experience*, 2005.

[Man04] Giovanni Manzini. Two space saving tricks for linear time LCP computation. Technical Report TR-INF-2004-02-03-INUPMN, Università der Piemonte Orientale, Dipartimento de Inforamtica, 2004.

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

[MF04] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, June 2004.

[MM93] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comp.*, 22(5):935–948, October 1993.

[PST05] Simon Puglisi, W. F. Smyth, and Andrew Turpin. The performance of linear time suffix sorting algorithms. submitted for publication, 2004/2005.

[Sad02] Kunihiko Sadakane. Succint representation of lcp information and improvements in the compressed suffix arrays. In *Proc. 13th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 225–232, 2002.

[Sad04] Kunihiko Sadakane. Compressed suffix trees with full functionality. *To be published in Theoretical Computer Science*, 2004.

[SS01] Kunihiko Sadakane and Tetsuo Shibuya. Indexing huge genome sequences for solving various problems. *Genome Informatics*, 12:175–183, 2001.

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

[Wei73] Peter Weiner. Linear pattern matching. In *Proc. 14th IEEE Symp. on Switching and Automata Theory*, pages 1–11. IEEE, 1973.