Cache-Oblivious String Dictionaries

Gerth Stølting Brodal[*] Rolf Fagerberg[†]

Abstract
We present static cache-oblivious dictionary structures for strings which provide analogues of tries and suffix trees in the cache-oblivious model. Our construction takes as input either a set of strings to store, a single string for which all suffixes are to be stored, a trie, a compressed trie, or a suffix tree, and creates a cache-oblivious data structure which performs prefix queries in $O(\log_B n + |P|/B)$ I/Os, where n is the number of leaves in the trie, P is the query string, and B is the block size. This query cost is optimal for unbounded alphabets. The data structure uses linear space.

1 Introduction

Strings are one of basic data models of computer science. They have numerous applications, e.g. for textual and biological data, and generalize other models such as integers and multi-dimensional data. A basic problem in the model is to store a set of strings such that strings in the set having a given query string P as prefix can be found efficiently. A well-known solution is the trie structure [16], which for an unbounded (or, equivalently, comparison-based) alphabet supports such prefix queries in $O(\log n + |P|)$ time, if implemented right (see e.g. the ternary search trees of Bentley and Sedgewick [6]). In this paper, we study the prefix search problem in the cache-oblivious model.

1.1 Model of Computation The cache-oblivious model is a generalization of the two-level I/O-model [1] (also called the External Memory model or the Disk Access model). The I/O-model was introduced to better model the fact that actual computers contain a hierarchy of memories, where each level acts as a cache for the next larger, but slower level. The vast differences in access times for the levels makes the time for a memory access depend heavily on what is currently the innermost level containing the data accessed. In algorithm analysis, the standard RAM (or von Neumann) model is unable to capture this. The two-level I/O-model approximates the memory hierarchy by modeling two levels, with the inner level having size M, the outer level having infinite size, and transfers between the levels taking place in blocks of B consecutive elements. The cost measure of an algorithm is the number of memory transfers, or I/Os, it makes.

The cache-oblivious model, introduced by Frigo et al. [17], elegantly generalizes the I/O-model to a multi-level memory model by a simple measure: the algorithm is not allowed to know the value of B and M. More precisely, a cache-oblivious algorithm is an algorithm formulated in the RAM model, but analyzed in the I/O-model, with an analysis valid for *any* value of B and M. Cache replacement is assumed to take place automatically by an optimal off-line cache replacement strategy. Since the analysis holds for any B and M, it holds for all levels simultaneously. See [17] for the full details of the cache-oblivious model.

Over the last two decades, a large body of results for the I/O-model has been produced, covering most areas of algorithmics. The cache-oblivious model, introduced in 1999, is younger, but already a sizable number of results exist. One of the fundamental facts in the I/O-model is that comparison-based sorting of N elements takes $\Theta(\text{Sort}(N))$ I/Os, where $\text{Sort}(N) = \frac{N}{B}\log_{M/B}\frac{N}{B}$ [1]. Also in the cache-oblivious model, sorting can be carried out in $\Theta(\text{Sort}(N))$ I/Os, assuming a so-called tall cache $M \geq B^{1+\varepsilon}$ [8, 17]. This tall-cache assumption has been shown to be necessary [9].

Below, we discuss existing I/O-efficient algorithms for problems on strings. For other areas, refer to the recent surveys [2, 22, 25, 26] for the I/O-model, and [4, 7, 12, 22] for the cache-oblivious model.

1.2 Previous String Results Basic string problems with well-known RAM model algorithms include string searching and pattern matching problems, string sorting, and various combinatorial string similarity problems, such as longest common subsequence and edit dis-

[*]BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), Department of Computer Science, University of Aarhus, IT-parken, Åbogade 34, DK-8200 Århus N, Denmark. Supported in part by the Danish Natural Science Research Council. E-mail: gerth@daimi.au.dk.

[†]Department of Mathematics and Computer Science, University of Southern Denmark, DK-5230 Odense M, Denmark. Supported in part by the Danish Natural Science Research Council (SNF). E-mail: rolf@imada.sdu.dk.

tance. We consider (in reverse order) to what extent similar results are known for external memory, that is, in the I/O-model and the cache-oblivious model.

The length of the longest common subsequence and the edit distance between two strings of lengths N_1 and N_2 can be solved by dynamic programming using two simple nested loops with running time $O(N_1 N_2)$ and space usage $O(\min\{N_1, N_2\})$. I/O-efficient versions of these algorithms are straight-forward: computing the $N_1 N_2$ values in rectangular $M \times M$ blocks gives an algorithm using $O(N_1 N_2/(MB))$ I/Os in the I/O-model. A cache-oblivious version with the same I/O bound can be made by using a recursively defined blocked computation order (similar to the recursively defined cache-oblivious matrix algorithms in [17]).

In internal memory, sorting n strings of total length N takes $\Theta(n \log n + N)$ time when the alphabet is unbounded (see e.g. [6]). In external memory, the complexity of sorting strings has not been settled in the I/O-model, hence even less in the cache-oblivious model. Some results (mostly in restricted models) appear in [5]. However, a simple upper bound is $O(\text{Sort}(N))$ I/Os, where N is the total length of the strings sorted, which can be achieved in the I/O-model as well as in the cache-oblivious model, e.g. by building the suffix array [20] over the concatenation of the strings.

The pattern matching algorithms comes in two flavors, with and without preprocessing. A representative of algorithms without preprocessing is the Knuth-Morris-Pratt algorithm [21]. It works by scanning the text and pattern, and accessing an array storing the failure function in a stack-wise fashion. Hence it by construction uses the optimal $O(N/B)$ I/Os for searching a pattern P in a string of length N. This holds in the I/O-model as well as in the cache-oblivious model.

For pattern matching with preprocessing, tries are the central data structure [16]. Tries provide a string dictionary structure over n strings supporting prefix searches in time $O(\log n + |P|)$ where P is the search string/prefix, if the alphabet is unbounded. For constant size alphabets, the time drops to $O(|P|)$. For unbounded alphabets, they can in the RAM model be built in $O(n \log n + N)$ time by direct insertion (or by the method in [6]), where N is the total length of the n strings. For constant size alphabets, the construction time is $O(N)$. Tries also form the basis of suffix trees. The suffix tree of a string S of length N is a compressed trie (alias a blind trie or Patricia [23] trie), i.e. a trie with all unary nodes omitted, which stores all suffixes of the string $S\$$ where $\$$ is a special end-of-string symbol not in the alphabet. It is a powerful data structure, supporting searches for a pattern P in time $O(\log N + |P|)$ for unbounded alphabets, and $O(|P|)$ for constant size alphabets. It is the basis for many combinatorial pattern matching algorithms [18]. A classic result [27] is that suffix trees in the RAM model can be built in $O(N)$ time for constant size alphabets, and $O(N \log N)$ time for unbounded alphabets. Farach [13] extended the former result to integer alphabets by reducing suffix tree construction to integer sorting.

Turning to external memory, the suffix tree algorithm of Farach can be implemented using sorting and scanning steps [14], and hence gives an $O(\text{Sort}(N))$ suffix tree construction algorithm in the I/O-model, as well as in the cache-oblivious model. This also provides the $O(\text{Sort}(N))$ string sorting algorithm mentioned above. Thus, sorting strings or suffixes of strings is feasible in both models. However, *searching* in the result is nontrivial. It can be proven that it is not possible to lay out a trie in external memory such that the search time for prefix searches is $O(\log_B n + |P|/B)$ I/Os in the worst case (see [11] for a lower bound argument). Using other means, Ferragina and Grossi were able to achieve this $O(\log_B n + |P|/B)$ bound in the I/O-model by the string B-tree string dictionary structure [15], thereby providing analogues of tries and suffix trees in the I/O-model. The string B-tree depends rather crucially on the value of B, and no similar result for the cache-oblivious model is known.

Thus, besides the I/O-complexity of string sorting (unknown also in the I/O-model), the central open question within basic string algorithms in the cache-oblivious model is the existence of an I/O-efficient string dictionary structure.

1.3 Our Contribution In this paper, we prove the existence of such an I/O-efficient string dictionary structure.

THEOREM 1.1. *There exists a cache-oblivious string dictionary structure supporting string prefix queries in $O(\log_B n + |P|/B)$ I/Os, where P is the query string, and n is the number of strings stored. It can be constructed in $O(\text{Sort}(N))$ time, where N is the total number of characters in the input. The input can be a set of strings to store, a single string for which all suffixes are to be stored, or a trie, compressed trie, or suffix tree (given as a list of edges of the tree, each annotated with a character). The structure assumes $M \geq B^{2+\delta}$.*

Unlike string B-trees, our structure basically does store a trie over the strings. The essential feature allowing us to get around the lower bound [11] on path traversals in tries (and indeed in general trees) laid out in memory is *redundancy*. Parts of paths in the trie may be stored multiple times, but with only a constant factor blowup in total space. More precisely, we cover the trie by a type of trees we denote *giraffe-trees*. Giraffe-trees may be of independent interest, as they provide a very simple linear space solution to the path traversal problem for trees in external memory, i.e. the problem of storing a tree (allowing redundancy) such that (a copy of) any given path p can be traversed in $O(|p|/B)$ I/Os. A previous solution to a similar problem in the I/O-model was given by Hutchinson et al. [19]. Our solution is simpler and works in the cache-oblivious model.

1.4 Overview of Structure Besides the giraffe-trees, the second essential feature of our structure is a decomposition of the trie into components and sub-components (denoted layers) based on judiciously balancing the progress in scanning the query pattern with the progress in reducing the number of strings left as matching candidates. Each layer is equipped with its own search structure in the form of a blind trie, which allows the search to choose among the giraffe-trees storing the parts of the trie contained in the layer. This giraffe-tree is then used for advancing the search. For the search in the blind tree, we need *look-ahead* in reading the pattern P, i.e. we may read further in P than it actually matches strings in the set. To avoid this ruining the I/O-efficiency, we need a tall-cache assumption in the case $|P| > M$.

The main difficulty we face in using the covering collection of giraffe trees is to determine which giraffe tree to use. To achieve the desired complexity, it is not enough to have a single blind trie to guide the choice. Rather, the searches need to be interleaved. More specifically, the purpose of the components and layers of the construction is to support searches on tries in which ranks do not drop too sharply from parent to child (the permitted drop is a function of the depth of a node). In the general case, the trie is partitioned into components, and each component is further partitioned into layers of doubly exponentially growing height (i.e., height $2 = 2^{2^0}$, $4 = 2^{2^1}, \ldots, 2^{2^i}, \ldots$). As said, each layer is kept in two structures: a blind trie and a covering collection of giraffe trees. A search traverses a component layer by layer until it exits the component (either entering a new component, reaching a leaf, or failing). Within a layer, the search first uses the blind trie and uses the node reached in the blind trie to select the giraffe tree in which to traverse the layer.

The difficulty is that the blind trie search may explore too far forward in the pattern (this can occur if the next layer is not reached). To ensure that this does not cause too much work we need to ensure that the number of I/Os that may be done in traversing the pattern while searching in the blind trie for the given ith layer is at most a constant factor greater than the number of I/Os done in traversing the giraffe trees for the previous layer. More precisely, an ith layer has at most $2^{\varepsilon 2^i}$ leaves. Thus searching the blind trie for an ith layer takes at most $O(2^{\varepsilon 2^i})$ I/Os. Traversing the giraffe trees for the preceding layer takes that most $O(2^{2^{i-1}}/B)$ I/Os. Further, the backtracking is a problem only if the relevant portion of the parent does not fit in main memory; but this portion has length at most 2^{2^i}.

The components and their data structures are laid out in memory in a manner which during searches allow them to be accessed I/O-efficiently in the cache-oblivious model. We do this by associating components with some of the nodes of a binary tree of height $O(\log n)$, and then using the van Emde Boas layout [24] of this tree as a guide to place the structures in memory. Hence, doubly-exponentially increasing values are prominent in the definition of components and their layers. To determine which nodes to associate the components with, we convert the decomposition of the trie into a binary tree of height $O(\log n)$ using a construction algorithm for weighted binary trees.

Our structure needs a tall cache assumption $M \geq B^{1+\delta}$ for the sorting involved in the construction algorithm, and $M \geq B^{2+\delta}$ for the I/O-efficiency of the query algorithm in the case $|P| > M$.

1.5 Preliminaries Let T be any tree. For a node v in T we denote by T_v the subtree rooted at v, and by n_v the number of leaves contained in the T_v. We define the *rank* of v to be $\text{rank}(v) = \lceil \log n_v \rceil$, where $\log x$ denotes the binary logarithm. We *define* the depth of v, $\text{depth}(v)$, to be the number of edges on the path from the root to v (except that if T is a blind trie, $\text{depth}(v)$ will be the string depth, i.e. the length of the string represented by the path from the root to v, including omitted characters).

2 The Data Structure

In this section we describe the main parts of our construction. We use details described later in Sections 3 to 5 as black-boxes.

We will assume that the input to our construction is a trie T given as the list of edges forming its Euler tour. If the input is not of this form, but instead e.g. a set of strings to store, a single string for which all suffixes are to be stored, a trie, a compressed trie, or a suffix tree (given as a list of edges of the tree, each annotated with a character), preprocessing can convert it to an Euler tour, as described in Section 7. We let n denote the number of leaves in T. We note that the construction actually applies to compressed (i.e. blind) tries also, if $\text{depth}(v)$ is defined as the string depth of v.

2.1 The Main Structure First we partition the input trie T into a set of disjoint connected *components*. A component with root v we denote C_v. Each component is again partitioned into a sequence of *layers* $D_v^0, D_v^1, \ldots$, where D_v^i stores the nodes u of C_v with $2^{2^{i-1}} \leq \text{depth}(u) - \text{depth}(v) < 2^{2^i}$. Section 3 gives the details of the definition of the decomposition of T, and proves some of its properties. Figure 1 illustrates a component consisting of four layers.

We then make a slight adjustment at the border between layers. If a leaf l of a tree in layer D_v^i has more than one child in D_v^{i+1}, we will break the edge above l and insert a dummy node l'. The edge between l' and l has string length zero, and l is the only child of l'. We then move l to the next layer D_v^{i+1}, and keep l' in layer D_v^i. After this adjustment, no leaf of any layer D_v^i has more than one edge to nodes in layer D_v^{i+1}.

For each component C_v we will have one data structure for each of the layers D_v^i. Abusing notation slightly, we will use D_v^i to denote both the layer and its data structure. For each tree τ of the forest of layer D_v^i, the data structure D_v^i has a blind trie for τ, and one or more giraffe trees (Section 4) forming a covering of τ. The blind trie is the tree τ compressed by removing all unary nodes, except the root (which can be unary). Leaves are null-ary, hence not removed. The data structure D_v^i consists of all the blind tries followed by all the giraffe-trees of all the coverings, laid out in memory in that order. Each of these trees are stored as a list of edges in BFS order.

Note that the blind tries used are for the layers of a component - i.e. nodes in other components are *not* included in the (sub)trie for which we form blind tries (as illustrated in Figure 1 by the white nodes). In particular, multiway nodes in the original trees may become unary nodes in a component, and hence not appear in a blind trie of a component.

To position all data structures D_v^i, we transform the tree T into a balanced tree T' with at most n leaves, where each node in T' has at most two children. The tree is formed by transforming each component C_v into a binary tree, and then gluing together these trees in the same way as the components were connected in T. That is, all edges connecting components in T will exist in T'. We call these edges *bridge* edges. The remaining edges of T are stored (possibly repeatedly) in the giraffe trees of the final structure.

We now describe the transformation of a component C_v, and the gluing process. For a component C_v we consider all nodes in it with at least one child outside C_v, i.e. nodes with bridge edges below it. Denote such nodes *border nodes* of C_v, and let u be a border node with children $z_1, z_2, \ldots, z_k$ outside C_v, for $k \geq 1$.

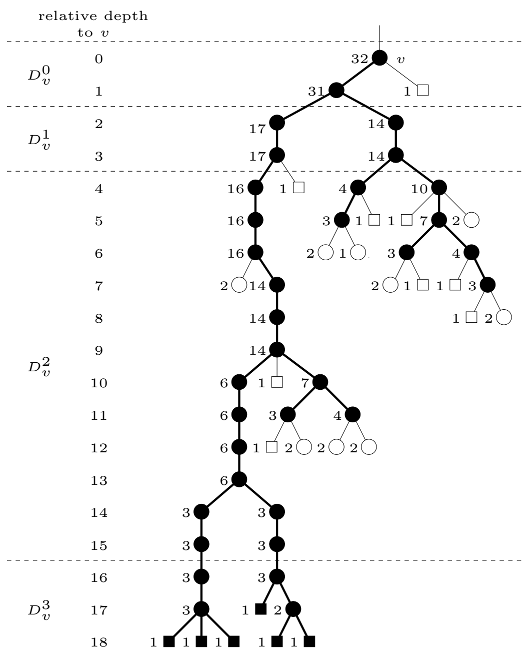

Figure 1: The black nodes show the component $C_v = \bigcup_{i=0}^{3} D_v^i$, whereas white nodes are children of nodes in D_v not being in D_v (children of white nodes are omitted). For each node w is shown the number of leaves n_w in T_w. In the example $\varepsilon = 1$.

Let the weight w_u of u be $\sum_{i=1}^{k} n_{z_i}$, i.e. the number of leaves in T below $z_1, z_2, \ldots, z_k$. We use the list of weights of border nodes as input to the algorithm of Lemma 5.1 (using dummy keys), producing a weighted binary tree with one leaf for each of the border nodes. Then for each border node u with children $z_1, z_2, \ldots, z_k$ outside C_v we make a list of k elements, where element i is the character on the edge in T above z_i, and has the associated weight n_{z_i}. We then use this list as input to the algorithm of Lemma 5.1, producing a weighted binary tree for each border node. The exception is if $k = 1$, where we simply make at tree with a single unary root and one leaf (the algorithm would return a single leaf only). Each bridge edge in T is now the edge above a leaf in a tree produced from a border node.

This collection of trees is then glued together into a single tree T' in the natural fashion: Each leaf of a tree generated from a component corresponds to a border node, and is identified with the root of the tree generated from that border node. Each leaf of a tree generated from a border node corresponds to a bridge edge, and is identified with the root of the tree corresponding to the component to which the lower end of the bridge edge belongs.

We now bound the height of T'. A root-to-leaf path in T corresponds to a root-to-leaf path in T'. By Corollary 3.1 in Section 3, the path in T will go through at most $\lceil \log_2 n \rceil$ components, hence the path in T' will pass at most $O(\log n)$ trees constructed by the algorithm of Lemma 5.1. By the telescoping property $\log x/y + \log y/z = \log x/y$, and by the way the weights used in the trees fit together, from the bound of Lemma 5.1 we get that the length of the path in T' is $O(\log n + \log n/1) = O(\log n)$. It follows that T' has height $O(\log n)$.

2.2 Memory Layout To decide where to store the data structures D_v^i in memory, we first associate the data structure D_v^i with the root in T' of the (sub)tree generated from the component C_v. Note that the node will be associated with D_v^i for all i, and that only some nodes in T' will have associated structures. We then consider the van Emde Boas layout [24] of T'. The van Emde Boas layout of a binary tree of height h lays out the nodes in a linear order by cutting the tree in the middle, producing a toptree of height $\lceil h/2 \rceil$ and many bottom trees of height at most $\lfloor h/2 \rfloor$. The nodes of the top tree is laid out first, and the bottom trees then follows in left to right order. Inside each top or bottom tree, the location of the nodes are determined by using the method recursively. The recursion in the van Emde Boas layout has $\log \log n$ levels, and at each level, we consider trees of at most 2^{2^i} nodes for some i. Each node is part of exactly one tree on each level.

We now use this layout to determine a position for the data structures: For a structure D_v^i associated with the node u in T', we place D_v^i at the position in the order corresponding to the end of the tree which, on the i'th lowest level of the van Emde Boas recursion, contains u.

The resulting data structure is a van Emde Boas layout of T', where the D_v^i structures are interspersed between nodes in the manner described above.

2.3 Searching We now describe the prefix search algorithm, given a search string P. Conceptually, the search will progress along a root-to-leaf path in T'. However, only the edges in the subtrees of T' generated by border nodes will actually be followed directly.

Traversal of the subtrees of T' generated by a component C_v is done by searching in the blind trie for the single tree τ in layer D_v^0 (for $i = 0$, the forest of layer D_v^i is a single tree). This will locate a leaf in τ which represents a path in τ whose common prefix with P is maximum among the paths in τ. The leaf will point to the root of the particular giraffe tree covering this path in the tree. This giraffe tree is then traversed to match against the edges removed when the blind trie was generated from τ. The match ends either at the leaf or at an internal node.

Consider the first case. If the leaf is a leaf of the original trie T, the prefix search is completed. If the leaf has a child in layer D_v^1 (of which there can be at most one by the border adjustment above), it points to the blind trie in the data structure D_v^1 which contains this child (the layer D_v^i for $i \geq 1$ may be a forest, hence have many blind tries in its data structure). If the match can continue along the edge to this child, the search continues in the data structure D_v^1, using the same procedure. If the search cannot continue along this edge, but the leaf in T' has children in other components, it is a border node, and we find the structure for the next component below it by searching in the subtree of T' that was generated by the border node. This search is a plain search in this binary search tree. This search will locate the next component to continue the search with (or the prefix search is found to be completed).

Consider the second case of the giraffe-tree matching ending at an internal node. Again, either the prefix search is complete, or the node is a border node, in which case the search continues in the subtree of T' generated by this node.

3 Tree Decomposition

In this section we describe our partitioning of an arbitrary rooted tree T into a set of disjoint connected components. The component rooted at a node v is denoted C_v. In the following we let $\varepsilon \in (0, 1]$ a constant. The parameter ε influence the size of the components defined, such that a larger value ε causes larger components to be constructed.

The components are identified top-down. First the component C_r containing the root r is identified. Recursively the components are then identified for each subtree T_u rooted at a node $u \notin C_r$ where the parent of u is in C_r.

For a subtree T_v of T we define the component C_v as follows. For a node v recall that n_v denotes the number of leaves contained in the subtree T_v and $\text{rank}(v) = \lceil \log n_v \rceil$. For $i = 1, 2, \ldots$, let *strata* i with respect to v be the nodes u in T_v for which $2^{2^{i-1}} \leq \text{depth}(u) - \text{depth}(v) < 2^{2^i}$, and let strata 0 be those for which $\text{depth}(u) - \text{depth}(v) < 2^{2^0}$. Call a node u in strata i a *candidate* node if $\text{rank}(v) - \text{rank}(u) < \varepsilon 2^i$. The component C_v is the connected component containing v in the subgraph of T_v induced by the candidate nodes. We call the part of C_v contained in strata i (if any) for *layer* i of C_v, and denote it by D_v^i. The component C_v and its layers can be identified in a top-down traversal of T_v. Formally, we define C_v and D_v^i as follows.

Definition 3.1.

$$\begin{aligned}
(3.1)\quad D_v^0 &= \{u \in T_v \mid \mathrm{rank}(u) = \mathrm{rank}(v) \\
&\quad \wedge \mathrm{depth}(u) - \mathrm{depth}(v) < 2^{2^0}\} \\
(3.2)\quad D_v^i &= \{u \in T_v \mid \mathrm{rank}(v) - \mathrm{rank}(u) < \varepsilon 2^i \\
&\quad \wedge 2^{2^{i-1}} \leq \mathrm{depth}(u) - \mathrm{depth}(v) < 2^{2^i} \\
&\quad \wedge (\exists w \in D_v^{i-1} : \mathrm{depth}(w) - \mathrm{depth}(v) = 2^{2^i} - 1 \\
&\quad \wedge u \in T_w)\} \\
(3.3)\quad C_v &= \bigcup_{i=0}^{\infty} D_v^i
\end{aligned}$$

Figure 1 illustrates a component consisting of four layers. The lemma below lists the properties of the decomposition essential for our data structure.

Lemma 3.1.

1. *If a node $u \in C_v$ has a child w with* $\mathrm{rank}(w) = \mathrm{rank}(u)$, *then u and w are in the same component.*
2. *If a node $u \in T$ has only one child w, then u and w are in the same component.*
3. *D_v^i is a forest with at most $2^{\varepsilon 2^i + 1}$ leaves.*
4. *D_v^i contains at most $(2^{2^i} - 2^{2^{i-1}})2^{\varepsilon 2^i + 1}$ nodes.*
5. *For a node $u \in D_v^i$, $u \neq v$, with a child $w \notin C_v$, then* $\mathrm{rank}(v) - \mathrm{rank}(w) \geq \varepsilon 2^i$.

Proof. 1) follows directly from Definition 3.1. If u has only one child w, then $n_w = n_u$ and $\mathrm{rank}(w) = \mathrm{rank}(u)$, i.e. by 1) w and u are in the same component and 2) is true.

To prove 3), let $w_1, \ldots, w_k$ be the leaves of the forest D_v^i (i.e. internal nodes or leaves in T, where no child is in D_v^i). Since $T_{w_1}, \ldots, T_{w_k}$ are disjoint subtrees of T_v, we have $n_{w_1} + \cdots + n_{w_k} \leq n_v$. From $\mathrm{rank}(v) - \mathrm{rank}(w_j) \leq \varepsilon 2^i$ we get $2^{\lceil \log n_{w_j} \rceil} \geq 2^{\lceil \log n_v \rceil}/2^{\varepsilon 2^i}$, and it follows that $n_{w_j} \geq n_v/(2 \cdot 2^{\varepsilon 2^i})$. We conclude that D_v^i contains at most $2^{\varepsilon 2^i + 1}$ leaves, and 3) follows. Since each leaf in D_v^i has at most $2^{2^i} - 2^{2^{i-1}}$ ancestors within D_v^i (including the leaf), we from 3) conclude that D_v^i contains at most $(2^{2^i} - 2^{2^{i-1}})2^{\varepsilon 2^i + 1}$ nodes and 4) follows.

To prove 5) consider $u \in D_v^i$, $u \neq v$, where u has a child $w \notin C_v$. If $i = 0$ then $\mathrm{rank}(v) - \mathrm{rank}(w) \geq 1$ and the lemma follows trivially. If $i \geq 1$ then $\mathrm{rank}(v) - \mathrm{rank}(w) \geq \varepsilon 2^i$. □

From Lemma 3.1(1) it follows that the ranks of the roots of the components on a root-to-leaf path is strictly decreasing.

Corollary 3.1. *On a root-to-leaf path in a tree T with n leaves there are at most $1 + \lceil \log n \rceil$ components.*

It should be noted that if the tree T is a blind trie, then the depth used in Definition 3.1 should refer to the string depth of the nodes, i.e. $\mathrm{depth}(v)$ denotes the sum of the lengths of the edges from the root to v. The properties listed in and the proof of Lemma 3.1 also hold for the case where T is a blind trie. We also need one more modification: If an edge in T crosses from strata i to strata $i+1$, without the upper endpoint being on the lowest level of strata i, we break the edge by inserting a unary node on it at this lowest level.

4 Covering a Tree by Giraffe-Trees

Let T be a rooted tree with n leaves and N nodes in total. This section considers the problem of storing T such that the traversal of a prefix of length p of any predetermined root-to-leaf path can be done using $O(p/B)$ I/Os.

A straightforward solution requiring $O(nN)$ space is to store each of the n root-to-leaf paths in a separate array. A prefix of length p of any predetermined root-to-leaf path can then be traversed using $O(p/B)$ I/Os. Note that the n paths form a cover of all nodes of T. In this section we develope a space efficient solution for this problem. Theorem 4.1 summarizes the result of this section.

Theorem 4.1. *Given a tree T with N nodes, there exists a cache-oblivious covering of T by subtrees (giraffe-trees) where the total space requirement of the covering is $O(N)$, each root-to-leaf path is present in one subtree, and the prefix of length p of a predetermined root-to-leaf path can be traversed in $O(p/B)$ I/Os. Given the Euler tour of T, the covering can be constructed using $O(N/B)$ I/Os.*

A tree is denoted a *giraffe-tree* if at least $N/2$ nodes are ancestors of all leaves, i.e. the tree consists of a long path (neck) of length at least $N/2$ with the remaining nodes are attached as subtrees below the lowest node on the path. Note that a single path is always a giraffe-tree by itself.

Lemma 4.1. *Let T be a giraffe-tree with N nodes stored in BFS layout. Traversing a path of length p starting at the root of T requires $O(p/B)$ I/Os.*

Proof. If $p \leq N/2$, then only the topmost node of the tree is accessed which all have degree one. Since in a BFS layout these nodes are stored consecutively left-to-right it follows that accessing the path requires $O(p/B)$ I/Os.

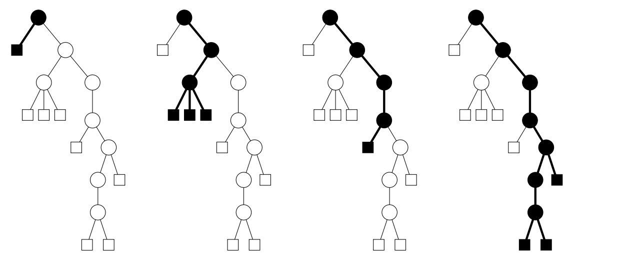
Figure 2: Covering a tree by the four giraffe-trees $T^{1:1}$, $T^{2:4}$, $T^{5:5}$, and $T^{6:8}$.

```
$i \leftarrow 1$
**while** $i \leq n$ **do**
    $j \leftarrow i$
    **while** $j < n$ **and** $T^{i:j+1}$ is a giraffe-tree **do**
        $j \leftarrow j + 1$
    output $T^{i:j}$
    $i \leftarrow j + 1$
```

Figure 3: Greedy algorithm for constructing a cover of T by giraffe-trees.

For the case $p > N/2$, we observe that in a BFS layout the nodes on a path appear in left-to-right order in memory. Following a root-to-leaf path is therefore bounded by the cost of scanning the array storing all nodes in the tree, i.e. $O(N/B) = O(p/B)$ I/Os. □

It should be noted that Lemma 4.1 also holds if a DFS layout or a van Ende Boas layout is used, since both these layouts also ensure that the parent of a node is to the left of the node in the layout.

Let $\ell_1, \ldots, \ell_n$ be the leaves of T in left-to-right order. For $1 \leq i \leq j \leq n$ let $T^{i:j}$ denote the subtree consisting of the nodes on the path from the root to the leaves $\ell_i, \ell_{i+1} \ldots, \ell_j$. The greedy algorithm in Figure 3 constructs a covering of T by a sequence of subtress $T^{i:j}$, where the leaves of T are processed from left-to-right and the next leaf is included in the current subtree until the next leaf forces the subtree under construction not to be a giraffe-tree. The generated subtrees are clearly giraffe-trees and form a covering of T. In Figure 2 is shown the cover generated by the greedy algorithm for a tree with eight leaves.

Lemma 4.2. *The algorithm in Figure 3 constructs a covering of T with giraffe-trees of total size $O(N)$, where N is the number of nodes in T.*

Proof. Let $T^{i:j}$ and $T^{j+1:k}$ be two consecutive giraffe-trees constructed by the greedy algorithm. Observe that from $T^{i:j}$ only nodes on the path to the leaf ℓ_{j+1} can appear in any of the succeeding subtrees constructed after $T^{i:j}$.

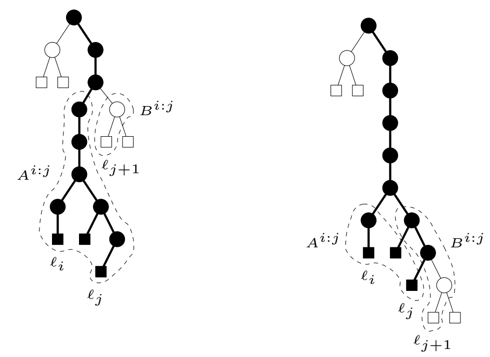

Figure 4: Two cases illustrating how the construction of $T^{i:j}$ is charged to the nodes in $A^{i:j}$ and $B^{i:j}$.

We will charge the construction of $T^{i:j}$ to two sets of nodes $A^{i:j}$ and $B^{i:j}$: $A^{i:j}$ are the nodes in $T^{i:j}$ that are not on the path to ℓ_{j+1}, and $B^{i:j}$ are the nodes in T on the path to ℓ_{j+1} but not on the path to ℓ_i. See Figure 4. By construction $T^{i:j+1}$ is not a giraffe-tree implying $|A^{i:j}| + |B^{i:j}| > |T^{i:j+1}|/2$. It follows that $|T^{i:j}| < |T^{i:j+1}| < 2(|A^{i:j}| + |B^{i:j}|)$. Each node is charged as an $A^{i:j}$ node only once, namely in the last giraffe-tree containing the node. Similarly, each node is charged as a $B^{i:j}$ node at most once, namely in the tree constructed prior to the tree where the node appears as a node on the leftmost path for the first time. For the last tree $T^{i:j}$ constructed, the leaf ℓ_{j+1} does not exist and $A^{i:j} = T^{i:j}$. It follows that $\sum_{T^{i:j}} |T^{i:j}| < \sum_{T^{i:j}} 2(|A^{i:j}| + |B^{i:j}|) \leq 4N$. □

Given an Euler tour of T the algorithm in Figure 3 can be implemented to use $O(N/B)$ I/Os, by performing an Euler tour traversal of T and maintaining the path from the root to the current node on a stack. The output is the Euler tours for each of the trees $T^{i:j}$.

5 Weight Balanced Trees

Lemma 5.1. *Let $x_1 \leq x_2 \leq \cdots \leq x_n$ be a list of n keys in sorted order, and let each key x_i have an associated weight $w_i \in \mathbb{R}_+$. Let $W = \sum_{i=1}^{n} w_i$. A binary search tree in which each key x_i is contained in a leaf of depth at most $2 + 2\lceil \log W/w_i \rceil$ can be constructed cache-obliviously in $O(n)$ time and $O(n/B)$ I/Os.*

Proof. Let the *rank* $\text{rank}(T)$ of a tree T be $\lceil \log w \rceil$, where w is the sum of the weights of its leaves. Let a *link operation* on two trees T_1 and T_2 create a node which has T_1 as its left subtree, T_2 as its right subtree, and the maximum key in T_1 as its search key. For $r = \max\{\text{rank}(T_1), \text{rank}(T_2)\}$, the rank of the new tree is either r or $r + 1$.

The algorithm keeps a stack of search trees. It maintains the invariants that the ranks of trees are strictly decreasing from bottom to top in the stack, and that all keys in a tree are larger than all keys in the trees below it. The algorithm proceeds in n steps, where in step i, a single-leaf tree containing x_i is incorporated into the stack.

We now describe a single step. Let a *link operation on the stack* denote the operation of popping two trees from the stack, linking these and pushing the resulting tree onto the stack.

Let T' be the tree containing only x_i. If the topmost tree has rank larger than rank(T'), or the stack is empty, we simply push T' onto the stack and the step is done. The invariants are clearly maintained.

Otherwise, let U be the lowest tree on the stack for which rank$(U) \leq$ rank(T'). If U is not the tree at the top of the stack, we repeatedly do link operations on the stack until a link involving U is performed. By the invariants, the tree at the top of the stack is now a search tree of rank at most rank$(T') + 1$. Call this rank r. The step finishes with one of three cases: Case *i*) is $r =$ rank$(T')+1$, where we perform zero or more link operations on the stack until the two topmost trees have different ranks or the stack contains a single tree. We then push T' onto the stack. Case *ii*) is $r =$ rank(T'), where we push T' onto the stack and perform one or more link operations on the stack until the two topmost trees have different ranks or the stack contains a single tree. Case *iii*) is $r <$ rank(T'), where we push T' onto the stack and perform one link operation on the stack. We then perform zero or more link operations on the stack until the two topmost trees have different ranks or the stack contains a single tree. In all three cases, the invariants are clearly maintained.

After the last of the n steps, we perform link operations on the stack until a single tree remains.

As each link reduces the number of trees by one, exactly $n - 1$ links are done in total, each of which does $O(1)$ stack operations. Besides the linking, each of the n steps perform $O(1)$ stack operations and one scan step of the input list. The algorithm can therefore be implemented cache-obliviously to run in $O(n)$ time and $O(n/B)$ I/Os.

We now bound the depth of leaves. Let the rank of a node be the rank of the tree it roots. Denote an edge *efficient* if the rank of its upper endpoint is larger than the rank of its lower endpoint. Call an inefficient edge *covered* if it has an efficient edge immediately above it in the tree. It can be verified that in the final tree, all inefficient edges are covered except possibly those incident on a leaf or on the root. From this the depth bound follows: If there are k efficient edges on the path from the leaf containing x_i to the root, we have $\lceil \log w_i \rceil + k \leq \lceil \log W \rceil$, hence $\log w_i + k < 1 + \log W$ and $k < 1 + \log W/w_i$. Since k is an integer, $k \leq \lceil \log W/w_i \rceil$ follows. By the claim, there can be at most $2 + 2k$ edges on the path.

6 Analysis

LEMMA 6.1. *Storing D_v^i uses $O(|D_v^i|)$ space, which is $O(2^{2^{i+1}})$.*

Proof. The space required for the blind trie of D_v^i is $O(2^{\varepsilon 2^i})$, since by Lemma 3.1 (3) the number of leaves in the blind trie is $2^{\varepsilon 2^i + 1}$ and therefore the total number of nodes in the blind trie is $O(2^{2^{i+1}})$. The size of the blind trie is by definition $O(|D_v^i|)$.

By Theorem 4.1 the total space required for the giraffe cover of D_v^i is $O(|D_v^i|)$. By Lemma 3.1 (4) this is bounded by $O((2^{2^i} - 2^{2^{i-1}})2^{\varepsilon 2^i + 1})$. It follows that the total space usage is $O(|D_v^i|)$ which is $O(2^{2^{i+1}})$, since $\varepsilon \leq 1$. □

THEOREM 6.1. *A subtree X of T' of height 2^i in the van Emde Boas layout of T' requires space $O((2^{2^i})^3)$.*

Proof. The recursive layout of X stores the nodes of T' within X and the D_v^j structures for all $v \in X$ and $j \leq i$. Since $|X| = O(2^{2^i})$, we have by Lemma 6.1 that the space required for storing X is $O(2^{2^i} \cdot \sum_{j=0}^{i} 2^{2^{j+1}}) = O((2^{2^i})^3)$. □

THEOREM 6.2. *Prefix queries for a query string P in a string dictionary storing n strings use $O(\log_B n + |P|/B)$ I/Os.*

Proof. The number of I/Os for a prefix search are caused by either accessing the pattern P or accessing the string dictionary data structure.

We first analyse the number of I/Os performed for accessing the query string. If P was scanned left-to-right, the I/Os for accessing P was clearly $\lceil |P|/B \rceil$. Unfortunately, the lookahead in P while searching blind tries cause random I/Os to P. We can without loss of generality assume that we keep the next $\Theta(M)$ unmatched characters of the pattern in memory, i.e. only lookahead of $\Omega(M)$ characters can cause random I/O. Consider the case where the access to a D_v^i causes a lookahead of $\Omega(M)$ during the blind trie search for D_v^i, i.e. $2^{2^i} = \Omega(M)$. The search in D_v^{i-1} matched $\Omega(2^{2^{i-1}})$ characters in P, and the blind trie for D_v^i has size $O(2^{\varepsilon 2^i})$. To charge the random I/Os caused by the blind trie search for D_v^i to the part of the query string matched for D_v^{i-1}, we only need

$$B \cdot 2^{\varepsilon 2^i} = O(2^{2^{i-1}}) \tag{6.4}$$

to be satisfied. Assuming the tall cache assumption $M \geq B^{2+\delta}$ for some constant $\delta > 0$, we from $2^{2^i} = \Omega(M)$ get $B = O((2^{2^i})^{1/(2+\delta)})$. The equality (6.4) is implied by $(2^{2^i})^{1/(2+\delta)} \cdot 2^{\varepsilon 2^i} = O(2^{2^{i-1}})$, which again follows from $1/(2+\delta) \leq 1/2 - \varepsilon$, which is satisfied for sufficiently small ε.

To count the number of I/Os while accessing T' we consider a search path in T'. By Theorem 6.1 each subtree in the recursive van Emde Boas layout of height 2^t fits into $O(1)$ blocks for $t = \lfloor \log\log B^{1/3} \rfloor$. Assuming we always keep in memory the currently traversed height 2^t subtree from the van Emde Boas layout. The number of I/Os following the path becomes $O((\log N)/2^t) = O(\log_B N)$, not counting the I/Os used for accessing the D_v^i data structures not stored in the cached height 2^t tree, i.e. the D_v^i data structures with $i > t$.

To count the I/Os for accessing the $D_v^0, D_v^1, \ldots, D_v^s$ for a component C_v, we now only have to count the number of I/Os for accessing $D_v^{t+1}, D_v^{t+2}, \ldots, D_v^s$ where $s \geq t$. Without loss of generality we assume that each of these D_v^i will need to be read into memory. By Lemma 3.1 (3) and Theorem 4.1 each of these requires $O(1 + 2^{\varepsilon 2^i}/B + p_i/B)$ I/Os where p_i is the length of the path matched in D_v^i. For $\varepsilon \leq 1/2$ we have $2^{\varepsilon 2^i} \leq 2^{2^{i-1}}$, and the scanning of the blind trie for layer i is dominated by the matched part of the query string for layer $i-1$. We have $O(2^{\varepsilon 2^i}/B) = O(p_{i-1}/B)$, and the total number of I/Os is therefore $O(\sum_{i=t+1}^{s}(1+p_i/B))$. Since $(2^{2^{t+1}})^3 = \Omega(B)$ we get $2^{2^{t+3}} = \Omega(B)$. Since $p_i = \Theta(2^{2^i})$ for $i < s$, we in the sum only for $i = t+1$, $i = t+2$ and $i = s$ can have that $p_i/B = o(1)$. For the remaining terms $p_i/B = \Omega(1)$ and the "+1" in the sum can be charged to p_i/B.

Since $(2^{2^{t+1}})^3 = \Omega(B)$, we from Lemma 3.1 (5) have that the rank decreases by at least $\varepsilon 2^k = \Omega(\log B)$ when the search terminates at D_v^k, for $k > t$. It follows that at most $O(\log_B N)$ times we have to charge $O(1)$ additional I/Os. The remaining I/Os are charged to the scanning of P. □

7 Construction Algorithm

Theorem 7.1. *Given an edge list representation of the Euler tour of a trie or blind trie T with N nodes, the cache-oblivious trie data structure can be constructed in $O(\mathrm{Sort}(N))$ time.*

Proof. Due to space restrictions, we only give a sketch of the steps involved. The full number of steps is large, and their description tedious, but only standard tools in the area are used.

To find the decomposition into components and layers, we need to distribute various information upwards and downwards between nodes in (the edge list representation of) T. This can be done by first annotating the nodes with their BFS level [3], then propagating the information upwards and downwards using a priority queue [3] of edges. The information distributed is depth, n_v, and membership of components and layers. The Euler tour of T can then be cut up into Euler tours for the various layers (separating the components using sorting). These Euler tours can then be used for generating the blind tries and giraffe trees.

The many inputs to the algorithm of Lemma 5.1 can be collected using sorting steps. The result can be distributed and glued together using sorting steps.

To compute the memory layout of the structure, we first find the order of the memory positions of the nodes of T' in the van Emde Boas layout. By the implicit navigation formula for van Emde Boas trees [10], and the observation [10] that an in-order traversal of a van Emde Boas layout has scanning cost, the information can be found by simultaneous inorder traversals of the (implicit) van Emde Boas layout and T'.

The nodes in T' to have associated D_v^i structures can be marked with the maximal i of these structures by a sorting step. The height of a node determines how many trees in the van Emde Boas recursion it is a root of. This information can be stored in an $O(\log n)$ size array, and used during a backwards DFS traversal of the van Emde Boas laid out T'. Using the pre-calculated sizes of subtrees of the van Emde Boas recursion (again $O(\log n)$ information), we can during the DFS traversal maintain a current list of pointers to the correct locations. These locations are then output when a marked node in T' is met. The location information is distributed to the data structures by a sorting step. Finally, everything is moved into correct position by a sorting step, and all relevant pointers in the structures are updated using sorting steps. □

We note that if the input is not on the Euler tour form above, but instead e.g. a set of strings to store, a single string for which all suffixes are to be stored, or a trie, compressed trie, or suffix tree (given as a list of edges of the tree, each annotated with a character), preprocessing can convert it to an Euler tour. For the last three input types, this can be done by directly constructing the Euler tour [3]. For the second type of input, we build the suffix array and associated LCP array [14, 20] cache-obliviously in $O(\mathrm{Sort}(N))$ I/Os. Scanning these arrays, we can construct the edges of the compressed trie of the suffixes, by keeping a stack of the right-most path in the trie built so far. This takes $O(N/B)$ I/Os. For the first type of input, we first concatenate the input strings, then build the suffix array and LCP array, which are then pruned for non-relevant entries (distributing the relevancy information using sorting). We then find the edges of the compressed trie as above. If the full trie is wanted, the edges can be expanded, with character information distributed using sorting. Finally, the Euler tour of the tree is calculated.

8 Conclusion

We have given a cache-oblivious dictionary structure for strings performing prefix queries in $O(\log_B n + |P|/B)$ I/Os, where n is the number of leaves in the trie, P is the query string, and B is the block size. This is an optimal I/O-bound for unbounded alphabets. The structure provides analogues of tries and suffix trees in the cache-oblivious model.

It remains an open problem whether a tall cache assumption is required for performing prefix searches in $O(\log_B n + |P|/B)$ I/Os in the cache-oblivious model. In the I/O-model it is not, as demonstrated by string B-trees [15].

Acknowledgements Thanks to Gudmund S. Frandsen for suggesting the name giraffe-tree, and to an anonymous referee for suggestions which improved the presentation.

References

[1] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

[3] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. 34th Ann. ACM Symposium on Theory of Computing*, pages 268–276, 2002.

[4] L. Arge, G. S. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook on Data Structures and Applications.* CRC Press, 2005.

[5] L. Arge, P. Ferragina, R. Grossi, and J. Vitter. On sorting strings in external memory. In *Proc. Ann. ACM Symposium on Theory of Computation*, pages 540–548, 1997.

[6] J. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, 1997.

[7] G. S. Brodal. Cache-oblivious algorithms and data structures. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 3–13. Springer Verlag, 2004.

[8] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages, and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 426–438. Springer Verlag, 2002.

[9] G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. 35th Ann. ACM Symposium on Theory of Computing*, pages 307–315, 2003.

[10] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache-oblivious search trees via binary trees of small height. In *Proc. 13th Ann. ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.

[11] E. Demaine, J. Iacono, and S. Langerman. Worst-case optimal tree layout in a memory hierarchy. Technical Report cs.DS/0410048, arXiv web repository, 2004.

[12] E. D. Demaine. Cache-oblivious data structures and algorithms. In *Proc. EFF summer school on massive data sets*, Lecture Notes in Computer Science. Springer, To appear.

[13] M. Farach. Optimal suffix tree construction with large alphabets. In *38th Ann. Symposium on Foundations of Computer Science*, pages 137–143, 1997.

[14] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

[15] P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.

[16] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, Sept. 1960.

[17] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache oblivious algorithms. In *Proc. 40th Ann. Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[18] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[19] D. Hutchinson, A. Maheshwari, and N. Zeh. An external-memory data structure for shortest path queries. In *Proc. Ann. Combinatorics and Computing Conference*, volume 1627 of *Lecture Notes in Computer Science*, pages 51–60. Springer Verlag, 1999.

[20] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th Int. Colloquium on Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer Verlag, 2003.

[21] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[22] U. Meyer, P. Sanders, and J. F. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science.* Springer Verlag, 2003.

[23] D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, Oct. 1968.

[24] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.

[25] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[26] J. S. Vitter. Geometric and spatial data structures in external memory. In D. Mehta and S. Sahni, editors, *Handbook on Data Structures and Applications.* CRC Press, 2005.

[27] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th Ann. IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.