Improving Suffix Array Locality for Fast Pattern Matching on Disk

Ranjan Sinha
Department of Computer Science
and Software Engineering
The University of Melbourne
Victoria 3010, Australia
rsinha@csse.unimelb.edu.au

Simon J. Puglisi
School of Computer Science
and Information Technology
RMIT University
Victoria 3001, Australia
sjp@cs.rmit.edu.au

Alistair Moffat
Department of Computer Science
and Software Engineering
The University of Melbourne
Victoria 3010, Australia
alistair@csse.unimelb.edu.au

Andrew Turpin
School of Computer Science
and Information Technology
RMIT University
Victoria 3001, Australia
aht@cs.rmit.edu.au

ABSTRACT

The suffix tree (or equivalently, the enhanced suffix array) provides efficient solutions to many problems involving pattern matching and pattern discovery in large strings, such as those arising in computational biology. Here we address the problem of arranging a suffix array on disk so that querying is fast in practice. We show that the combination of a small trie and a suffix array-like blocked data structure allows queries to be answered as much as three times faster than the best alternative disk-based suffix array arrangement. Construction of our data structure requires only modest processing time on top of that required to build the suffix tree, and requires negligible extra memory.

Categories and Subject Descriptors

F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—*Pattern matching*; E.2 [Data Storage Representations]: Contiguous representations; H.3.1 [Information Storage and Retrieval]: Content Analysis and Indexing—*Indexing methods*; H.3.2 [Information Storage and Retrieval]: Information Storage—*File organization*; I.5.4 [Pattern Recognition]: Applications—*Text processing*.

General Terms

Algorithms, experimentation, performance.

Keywords

Pattern matching, suffix tree, suffix array, secondary storage, disk-based algorithm.

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

1. INTRODUCTION

Suffix trees are an important data structure for string processing, and support efficient solutions to many problems involving pattern matching and pattern discovery in large strings, such as those arising in computational biology. A particular problem at which suffix trees excel is *pattern matching*: given an m character pattern $P[0 \ldots m-1]$ and an n character text $T[0 \ldots n-1]$, report the set of positions in T at which P occurs. If a suffix tree data structure is built from T, then patterns can be located in the text in time $O(m + occ)$, where occ is the number of occurrences of the pattern P in T [21, 30]. Suffix trees are also adept at the discovery of *generic patterns*. For example, they can be used to locate all repeating substrings in a text in optimal time [13]. Many more applications of suffix trees have been described [1, 4, 13, 26].

The rapid growth in the size, number and popularity of genomic databases – as well as other types of information – has seen the demand for the virtues of the suffix tree redoubled. A problem with suffix trees however, is that they typically require as much as $12n$ bytes of memory [16] to represent a text of n characters, assuming that characters in T and P are drawn from a small alphabet like the ASCII character set. More space may be required for larger alphabets. This large memory requirement tends to have a consequential effect on the practical speed of access algorithms, as locality of reference during search operations on the tree tends to be poor.

Significant research effort has been focused toward the construction of suffix trees for genome sized texts using secondary storage (disk) [9, 23, 27], which is generally much more abundant than primary memory assumed by traditional suffix tree implementations, but also much slower to access. Another important development has been the evolution of the suffix array data structure. It was recently shown that, when augmented with relatively small auxiliary arrays, the suffix array is equivalent to the suffix tree, in that any algorithm on one data structure can be mapped to the other [1]. The attractiveness of these so-called *enhanced suffix arrays* is that they are completely "array based" and have some benefits in terms of improving the locality of memory references.

The layout of the suffix tree on disk, or the enhanced suffix array in memory, has until now been determined primarily by the construction algorithms used to build them, and not by the applications and algorithms in which they are used. Here we ask the following question: *given a large suffix tree, how should it be arranged on*

disk so that querying, and other algorithms on the tree, are fast in practice? We show that the combination of a small trie in memory, with a suffix array-like data structure on disk, allows queries to be answered up to three times faster than the best alternative disk-based suffix array. Construction of our data structures adds only modest processing time on top of that required to build the suffix tree, and requires negligible extra memory. We stress that our approach is independent of the suffix tree construction algorithm used. Any disk-resident suffix tree on which a depth first traversal can be performed can be transformed into the data structure described here.

The contributions of this paper are summarized as follows:

- A scalable disk-based data structure is introduced, the LOF-SA, that permits efficient and scalable pattern matching in hierarchical memories, because of its high locality of access.
- The LOF-SA merges the suffix and lcp arrays and copies fringe characters from the collection such that each string in the text is represented by a three-field LOF entry kept contiguously on disk.
- The query performance of the LOF-SA on several different types of data is examined, including DNA data and English text, ranging in size from 10 MB to 2 GB, and using query lengths ranging from 6 to 1000 characters; with results compared to current suffix array and suffix tree data structures.

The remainder of this paper is organized in the usual manner. Section 2 establishes notation, and defines a number of key concepts. Section 3 then sets the context for our developments, by describing some of the previous work in the area. Section 4 introduces our data structure, the LOF-SA, and describes how to construct and query it. To evaluate our approach, Sections 5 and 6 first report the methodology used in our experiments, and then use it to compare the LOF-SA to other disk-based suffix tree and suffix array implementations.

2. DEFINITIONS AND NOTATION

Let Σ be a *constant*, *indexed* alphabet consisting of symbols σ_j, $j = 1, 2, \ldots, |\Sigma|$ ordered $\sigma_1, \sigma_2, \cdots, \sigma_{|\Sigma|}$. In this paper we will assume the common case that $|\Sigma| \in 0\ldots255$, where each symbol requires one byte of storage. For example, for DNA data, $|\Sigma| = 4$; and for ASCII text, $|\Sigma| = 128$. Unless otherwise stated we assume Σ is minimal, that is, $\Sigma = \{0, 1, 2, \ldots, |\Sigma| - 1\}$.

We consider a finite, nonempty string $x = x[0\ldots n]$ of $n + 1$ symbols. The first n symbols of x are drawn from Σ and comprise the input. The final symbol, $x[n]$, is a special "end of string" character, \$, defined to be lexicographically smaller than every other character in Σ. For the purposes of calculating space requirements, we assume that n is such that any integer in the range $0\ldots n$ can be stored in one memory word, and that a pointer into x similarly requires four bytes. For $i = 0, \ldots, n$ we write $x[i\ldots n]$ to denote the *suffix* of x of length $n-i+1$, that is $x[i\ldots n] = x[i]x[i+1]\cdots x[n]$. For simplicity we will frequently refer to $x[i\ldots n]$ simply as "suffix i", or "the ith suffix". Similarly, we write $x[0\ldots i]$ to denote the *prefix* of x of length $i + 1$. We write $x[i\ldots j]$ to represent the *substring* $x[i]\cdots x[j]$ of x that starts at position i and ends at position j.

The *trie* T_S, for a set of strings S is a rooted directed tree with the following properties: (a) each edge is labeled with exactly one character; (b) any two edges out of the same node have distinct labels; (c) every member s_i in S maps on to the characters on the path from the root of T_S to exactly one node v of T_S, and conversely the path to every leaf of T_S maps to some string in S. The parent of node v is denoted by *parent*(v); and *occ*(v) refers to the total number of leaves in the subtree rooted at v.

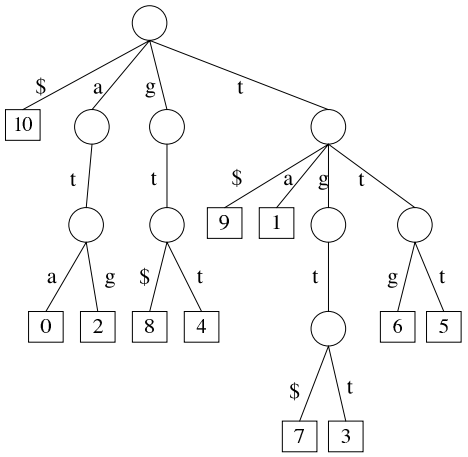


Figure 1: Suffix trie for the string $x =$ atatgtttgt\$.

The *suffix tree* ST_x for string $x[0\ldots n]$ is a rooted, directed tree with exactly $n + 1$ leaves labeled $0\ldots n$. Each internal node has at least two children and each edge is labeled with a nonempty substring of x. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for the leaf labeled i, the concatenation of the edge-labels on the path from the root to the leaf i spells out exactly $x[i\ldots n]$, that is the suffix of x that begins at position i [13]. Thus, ST_x is a compact representation of a trie containing exactly the suffixes of x, where all nodes with one child are merged with their parents. We refer to an ordinary (uncompacted) trie containing the n suffixes of x as a *suffix trie*. The suffix trie for the string $x =$ atatgtttgt\$ (with internal one-child nodes retained, but trailing one-child nodes removed) is shown in Figure 1.

We will also make use of the *suffix array* of x, which we write SA_x or just SA. The suffix array is an array $SA[0\ldots n]$ which contains a permutation of the integers $0\ldots n$ such that $x[SA[0]\ldots n] < x[SA[1]\ldots n] < \cdots < x[SA[i]\ldots n] < \cdots < x[SA[n]\ldots n]$, where string comparison is lexicographic based on the ordering of the alphabet Σ. That is, $SA[j] = i$ iff $x[i\ldots n]$ is the jth smallest suffix of x. The entries in SA_x correspond directly to the leaves of ST_x as they would be encountered during a depth first traversal of ST_x. The key feature of a suffix array compared to a suffix tree is that all the positions where a given pattern occurs in x appear in a contiguous portion of SA_x. This observation neatly and elegantly reduces the pattern matching problem to the question of finding an appropriate interval of SA_x.

A structure that is often used as an adjunct to a suffix array is an *lcp array*, $LCP[0\ldots n]$, where, for $j \in 1\ldots n$, $LCP[j]$ is the length of the *common prefix* shared by the strings indicated by $SA[j-1]$ and $SA[j]$. For the same example string:

	0	1	2	3	4	5	6	7	8	9	10
x	a	t	a	t	g	t	t	t	g	t	\$
SA	10	0	2	8	4	9	1	7	3	6	5
LCP	–	0	2	0	2	0	1	1	3	1	2

More generally, we also define $lcp(y, z)$ as the longest common prefix of strings y and z, so that

$$LCP[j] = lcp(x[SA[j-1]\ldots n], x[SA[j]\ldots n])\,.$$

3. RELATED WORK

The suffix tree was introduced by Weiner [30], who described an $O(n)$ time and space, in-memory construction algorithm. In subsequent years McCreight [21] and later Ukkonen [28] devised simpler construction methods having the same time and space bounds. Much effort has been expended improving the performance in practice of suffix trees, culminating in the work of Kurtz [16]. The focus has largely been on efficiency of construction and/or representation, with the goal usually to reduce the space required by the final index, and the space required during construction.

The most efficient way to locate a pattern in a suffix tree is to traverse it top down until either the pattern exhausts or a mismatch is found. Thus, existential queries are answered in $O(m)$ time – time proportional to the length of the pattern. Returning the positions where the pattern occurs is then a matter of traversing the subtree at which the search ended – the positions of occurrence are the leaves of the subtree, and enumerating them takes a further $O(occ)$ time. Lack of locality of memory reference can slow these suffix tree based algorithms, as there is no sense in which parts of the trie that are logicly connected are physically adjacent. Recently several groups have made impressive progress on disk-based suffix tree construction [9, 23, 27], allowing practical construction of suffix trees on disk.

In the early 1990's Manber and Myers proposed suffix arrays [17] as a space efficient alternative to the suffix tree. A recent survey [24] shows that algorithms for constructing suffix arrays in primary memory have proliferated and matured to a point where it is now several times faster to construct SA_x than ST_x [20, 18]. The current state-of-the-art in *disk based* suffix array construction is the algorithm of Dementiev et al. [11] which is slightly slower than disk based suffix tree construction algorithms [27]. Earlier work on disk-based SA construction is due to Crauser and Ferragina [10], however neither they, nor Dementiev et al., investigate how to actually query the SA on disk after it is built.

Abouelhoda et al. [1] showed that when combined with relatively small auxiliary information, the suffix array provides equivalent functionality to the suffix tree, meaning that any algorithm using suffix trees can be implemented using less space via an *enhanced suffix array*. The most important piece of auxiliary information is the LCP array, described above. The LCP array can be computed in-memory in linear time from SA_x [15, 19], and also as a byproduct of some (earlier) in-memory suffix array construction algorithms [17].

Several algorithms for searching the suffix array in-memory exist [2, 17, 25]. While they offer a speed improvement over the suffix tree in-memory, they are particularly poorly suited to implementation on disk because of the random memory access bought about by the need to dereference SA pointers into the string x in order to compare strings. For this reason, all work to date on pattern matching (and other traversals) in the SA has been under the tacit assumption that the component data structures, including the indexed string, are held in random access memory.

Baeza-Yates et al. [5] investigate the problems associated with searching a suffix array in secondary memory. They propose a method that uses a small memory-resident index (known as SPAT array) to index the disk resident SA in order to reduce disk accesses. The SPAT array stores l characters from each suffix array block. During querying, the SPAT array is binary searched to find the start and end blocks. Depending on l and the size of the distinguishing prefix, both the text and the suffix array may need to be accessed if the query length is longer than l; and all accesses to the text are at arbitrary locations with poor spatial locality. Then, once the start and end blocks are found, they are binary searched, incurring further text accesses. The LOF-SA structure we suggest is also a two-level arrangement, but we achieve fewer disk accesses than the SPAT structure, and hence are able to demonstrate (in Section 5) faster searching times.

4. THE LOF-SA DATA STRUCTURE

The LOF-SA data structure consists of two main components, both of which reside on secondary storage.

4.1 The LOF trie

The first, a small trie which we refer to as the *LOF trie*, is a heavily truncated version of the suffix trie, annotated with leaf count information. Given a *bucket threshold* b, the LOF trie is a pruned suffix trie where each node v is decorated with two integers *start* and *end* that indicate respectively the offset into the SA of the leftmost and rightmost leaves of the subtree rooted at v. That is, $SA[start \ldots end]$ contains precisely the leaves of the subtree at v. The suffix trie is pruned by keeping as terminals all nodes v for which $occ(v) \leq b$ and $occ(parent(v)) > b$, plus all of their parents. This requirement ensures that every terminal node v is a shallowest node for which $occ(v) \leq b$. (Note that this description of the structure of the LOF trie is not the process that we actually use to construct it.) The terminal nodes of the LOF trie partition the SA into non-overlapping intervals, or *buckets*, of size not more than b. All of the suffixes in each bucket share a common prefix of length at least d, where d is the depth in the LOF trie of the node that represents that bucket.

The *start* and *end* values of a node allow the number of leaves in a subtree rooted at node v to be readily determined as $occ(v) = end - start$. However, in our implementation we store *occ* explicitly in each node, along with a byte indicating the number of children at that node. An array of that many items follows, with each item consisting of the symbol value, which is the label on the outgoing edge (1 byte); the *occ* count of the bucket (2 bytes); and a pointer to the child (4 bytes).

4.2 The LOF array

The second part of the LOF-SA, the *LOF array*, is an interleaving of the SA and LCP arrays, plus some extra *fringe* characters from the text, stored as an array of $n+1$ triples $LA_x[i] = \langle L, O, F[0 \ldots f-1] \rangle$, where:

- $LA_x[i].L = LCP_x[i]$;
- $LA_x[i].O = SA_x[i]$; and
- $LA_x[i].F[0 \ldots f-1]$ is a string of $f > 0$ fringe characters.

Because there is a one-to-one correspondence between the entries in SA_x and those in LA_x, the LOF trie partitions LA_x into the same buckets as SA_x.

The F component in each LOF triple allows the number of accesses to the text during pattern matching to be reduced. It represents an "advance preview" of the next f characters of the text, beyond those that are common to the whole bucket. The simplest definition of F would be to take

$$LA_x[i].F[0 \ldots f-1] = x[SA_x[i] + d \ldots SA_x[i] + d + f - 1] \,,$$

so that F is the next f characters after the common bucket prefix d. Below we consider other more useful alternatives that build on the knowledge that is available via the LCP component stored in $LA[x].L$.

The LOF-SA for our running example string is depicted in Figure 2, and can be compared to Figure 1.

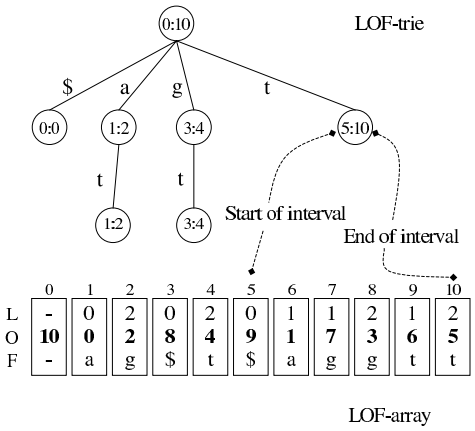


Figure 2: The LOF-SA for the string $x =$ atatgtttgt\$, using $f = 1$.

4.3 LOF-SA construction

Construction of the two LOF-SA components requires SA_x and LCP_x, as well as the string x. Once these two arrays have been constructed (using any suitable method), the LOF array is built in a straightforward way by interleaving the SA and LCP values, and extracting the necessary sequences from the text to fill in the fringe characters. The trie is constructed during a left-to-right scan over the LOF array. The shape of the trie (in fact, the shape of the entire suffix tree [1]) is determined by the LCP values. In our scan we construct explicitly a portion of the suffix trie under the aforementioned restriction that a subtree rooted at a terminal node is not allowed to contain more than b entries. This means that the terminal nodes may have uneven LCP depth, a quality determined by the underlying string. The LOF trie is constructed wholly in-memory and then arranged on disk in a van Emde Boas layout [29]. We experimented with several different layouts for the trie and found the van Emde Boas arrangement to be a good choice.

Efficient reconstruction of the suffix trie from the disk based LOF values is also possible, and is aided by the adjacency of the SA and the LCP values in each LOF entry.

4.4 Pattern Matching

Search in the LOF-SA begins in the LOF trie, and then proceeds to a block of the LOF array. Searching in the trie is done in the usual way: via a top down traversal, using the next character of the pattern in order to determine which child pointer to follow via a simple search. If the pattern exhausts during this traversal, the matching positions are determined by the *start* and *end* offsets into the LOF array stored at the node that provide the matching interval. The match locations can then be enumerated in $O(occ)$ time with a left-to-right scan of the disk-resident LOF array for *occ* positions starting at the *start* offset contained in the trie node – recall that matches with the pattern are contiguous in the LOF array, just as they would be in a suffix array.

If the search reaches a terminal node in the trie and there are still unresolved characters in the pattern, then the activity locus moves into the LOF array bucket corresponding to the LOF trie terminal node that was reached. Pseudo code for the in-bucket search algorithm *SearchLOFArray* is given in Figure 3. This algorithm makes use (line 6) of an auxiliary routine *MatchEntryToPattern*, shown in Figure 4, to determine the length of the match of the pattern with

SearchLOFArray($blockstart, blockend, P, match, m$)
1: $curr \leftarrow blockstart$
2: $match \leftarrow$ the length of the common prefix for this LOF bucket
3: **while** $match \neq m$ **do**
4: **if** $LA[curr].L < match$ **or** $curr = blockend$ **then**
– If the first condition, then already past location of longest *match*, without finding a *match* of m, and if the second condition, LOF block is exhausted. Either way,
5: **return** "No match"
– Otherwise, increase *match* length if appropriate.
6: $match \leftarrow MatchEntryToPattern(P, match, m, curr)$
7: $curr \leftarrow curr + 1$
– Scan through LCP values counting occurrences.
8: $start \leftarrow curr - 1$
9: **while** $curr \leq blockend$ **and** $match \leq LA[curr].L$ **do**
10: $curr \leftarrow curr + 1$
11: **return** $(start, curr - 1)$

Figure 3: Algorithm to search a portion of the LOF Array for the remainder of an already partially matched pattern.

MatchEntryToPattern($P, match, m, curr$)
1: **if** $f = 0$ **or** $LA[curr].L + f < match$ **then**
– Either there are no fringe characters or it is known that P matches all of them, so go straight to the text.
2: $p \leftarrow compare(P[match \ldots m-1], x[LA[curr].O + match \ldots n])$
3: $match \leftarrow match + p$
4: **else**
– Check the pattern against the fringe characters.
5: $p \leftarrow compare(P[match \ldots m-1], LA[curr].F[0 \ldots f-1])$
6: $match \leftarrow match + p$
7: **if** $match = LA[curr].L + f$ **then**
– The pattern matches all the fringe characters, check x.
8: $p \leftarrow compare(P[match \ldots m-1], x[LA[curr].O + match \ldots n])$
9: $match \leftarrow match + p$
10: **return** match

Figure 4: Algorithm for matching a single entry of the LOF array to the partially matched pattern.

the current LOF entry, using the fringe characters contained in each entry of the LOF array. Access is made to the text only when the fringe characters have been exhausted.

The outer while loop (lines 3–7) searches for the first match of the pattern to the entries in $LA[i \ldots j]$ – that is, the leftmost ℓ such that $x[LA[\ell] \ldots LA[\ell] + m] = P$. If a match is found (line 8) then the LCP values (the L field of the LOF entries) are used to guide the search to the end of the matching interval (lines 9–10). This logic exploits the following lemma.

Lemma 1: If

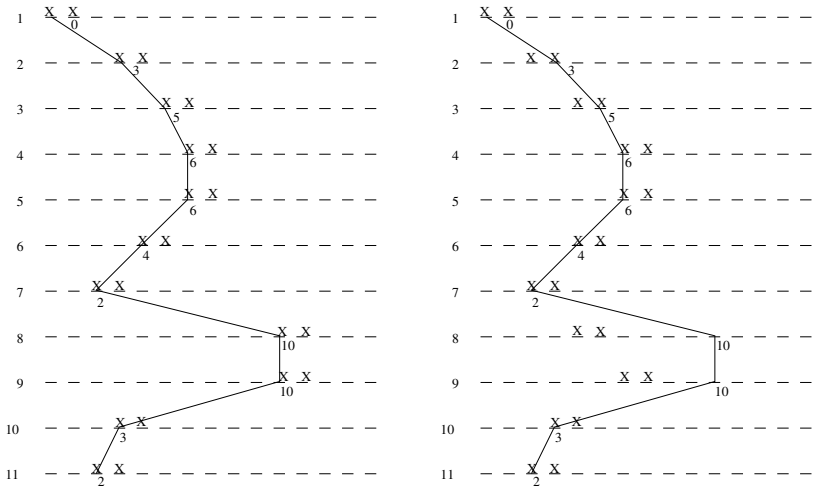
$$lcp(P[1 \ldots m], x[SA[i] \ldots n]) = m$$

and

$$lcp(x[SA[i] \ldots n], x[SA[i+1] \ldots n]) \geq m$$

then

$$lcp(P[1 \ldots m], x[SA[i+1] \ldots n]) = m\,.$$



(a) Right-justified (b) Left-justified

Figure 5: Right and left fringe justification. One LOF bucket is shown, with the LOF entries numbered on the left. The numbers below each line represent the corresponding LCP values, and the "X" symbols show the location of the fringe characters, assuming that $f = 2$. When the fringe is left justified, rightward shifts in the location of the fringe characters are such that no intervening positions are ever unknown.

Having found the start of the matching interval the algorithm of Figure 3 locates the end of the matching interval using a sequential scan of the LOF array, inspecting the LCP values, and avoiding any accesses to the text. In the worst case the complexity of the in-bucket search algorithm is $O(mb + occ)$ where b is the maximum size of the bucket to be searched.

If we recast Lemma 1 into the following lemma, another speed up is possible:

Lemma 2: If

$$k = lcp(P[1\ldots m], x[SA[i]\ldots n]) < m$$

and

$$lcp(x[SA[i]\ldots n], x[SA[i+1]\ldots n]) < k$$

then

$$\forall j \in i\ldots n, lcp(P[1\ldots m], x[SA[i]\ldots n]) \leq k < m\,.$$

This result allows us to rule out the possibly large number of entries remaining in the LOF array, and terminate the outer while loop without performing further matching. The idea is exploited in lines 10–11 of Figure 3. The improvement is only heuristic, but we have found it to be highly effective in practice.

The suggestion to use a LOF trie to narrow the search range in the LOF array is not dissimilar to the bucket table described in the suffix array search algorithm of Manber and Myers [17] and revisited by Abouelhoda et al. [2]. For a given small fixed parameter q, those authors recommend computing a lookup table of size Σ^q so that the section of the suffix array having each possible combination of q symbols as a prefix can be accessed in $O(q)$ time. This means that the actual portion of suffix array searched in detail is, for most inputs, much less than n. Our LOF trie takes matters a step further and guarantees that the section of the LOF array that undergoes secondary search is never larger than the predetermined block parameter b. Use of a trie rather than a fixed-dimensional index array adapts the search procedure gracefully to any skew on the distribution of characters present in the data.

4.5 Right and left justification

We commented earlier that taking the fringe characters to be those following the common block prefix was the simplest thing to do. This approach is appropriate if the LOF bucket is to be binary searched. But the most costly part of the secondary search is bringing the LOF bucket into main memory, and for small blocks, sequential search is just as practical. Use of sequential search opens up two further possibilities for the fringe characters, which we call *right justification* and *left justification*.

In the right-justified (denoted *RJ*) approach, the f distinguishing characters between successive strings, being the characters that extend the LCP values, are copied to each LOF entry as the fringe. During query processing, the bucket is processed sequentially, and each LOF entry can build on the one prior to it. The text is only accessed if all known prefix and fringe characters match the query, and the query is not exhausted. However, this arrangement allows gaps to arise between the fringe characters of successive strings when there are large increases in the LCP values, as shown in Figure 5a. For example, the strings represented by the LOF entries 7 and 8 have a gap of 6 characters, meaning that the text needs to be accessed and checked against the pattern even when the pattern is shorter than the LCP length plus f. Despite the need for occasional *false match checking*, this approach was surprisingly effective.

The alternative (again, only possible when sequential block scanning is being used) is to use a left justified fringe, denoted *LJ*. The fringe characters are more restrained, and advance more sedately from the left in a manner that eliminates gaps between successive strings. The "X"s in Figure 5b show the characters that would be stored in the LOF array as the fringe. The full extent of the LCP match is no longer used (see entry 9), but the gaps are always now at the tail of the prefix, rather than in the middle. The revised arrangement gives certainty of the prefix at all times, and thus delays – and in many cases, completely eliminates – accesses to the text. The drawback is that some amount of backtracking might be required, using an "overshoot" process in the LOF bucket that is still accumulating match confirmation beyond the point at which the match is found, but in practise the additional cost is small, and much more palatable than an access to the text. With a left justified

Collection	Source	Size (MB)	$\|\Sigma\|$
DNA10M	Human Genome Prefix	10	4
DNA20M	Human Genome Prefix	19	4
DNA40M	Human Genome Prefix	38	4
DNA100M	Human Genome Prefix	96	4
DNA200M	Human Genome Prefix	191	4
DNA400M	Human Genome Prefix	381	4
DNA1000M	Human Genome Prefix	954	4
DNA2000M	Human Genome Prefix	1907	4
PROT	Genbank non-redundant database	223	24
WSJ	TREC newswire (no punctuation)	471	62

Table 1: String collections used in our experiments.

fringe, text accesses are typically restricted to long query strings that are not fully answered within the bucket and occasional tail gaps, as occurs at LOF entry 9 in Figure 5b.

5. EXPERIMENTAL METHODOLOGY

This section presents the methodology used in the LOF-SA evaluation described in Section 6 below. The primary purpose of the evaluation is to gain an understanding of the effect of the LOF-SA parameters on query times and on the index size, and to compare the LOF-SA to other leading suffix tree and suffix array structures using realistic data sets.

5.1 Datasets

Experiments were carried out using a range of extracts from three different-domain datasets, as summarized in Table 1.

DNA: The DNA dataset consisted of several large prefixes, ranging in size from 10 million to 2 billion characters (bases), taken from the concatenated human genome [12]. In particular, a contemporary string processing task employed by molecular biologists is *local sequence alignment*, in which a query sequence is matched against a database of possible sequences and the best matches between subsequences of the query and a database sequence are reported. The first phase of many sequence alignment algorithms such as BLAST [3] and its recent variants [8] is an exact matching task in which short substrings of the full query are matched against the text. These results then undergo further processing to find the "best" alignments. The BLAST implementation (at present the most popular tool for the task) spends 80% of the time in performing an alignment doing exact matching of 11 character substrings of the query sequence [8]. While 11 is the "magic" length employed by BLAST, other pattern lengths have been proposed in the literature and are relevant to applications other than sequence alignment. We tested a range of pattern lengths.

PROT: Another common pattern matching task studied in bioinformatics is protein alignment. The process is similar to the DNA task described above, but the alphabet of possible symbols is 24, rather than 4. We worked with a 233 MB subset of the Genbank non-redundant protein set [6, 7].

WSJ: One of the most common uses of pattern matching is in searching text. To capture this application, we included a data file of newspaper articles from the Wall Street Journal subcollection of the TREC corpus [14]. Due to a peculiarity with the TDD software, we removed all punctuation characters, and replaced the space character with the ASCII letter "Z", so that we worked with a pure alphanumeric text. Consequently, we did not allow patterns to contain spaces.

5.2 Queries

Table 2 gives statistics for the pattern sets used for testing. In all cases the sets of patterns were drawn from the collections they were applied to, by selecting 1,000 random location in the text, and taking a sequence of the desired length. Table 2 shows the total number of matches in each text for each set of 1,000 queries. Patterns longer than around 100 characters tend to be unique and thus have only a single answer, but there were a few exceptions noted to this general observation. For example, in the WSJ trials some of the patterns picked out were long runs of blank characters, and these had a large number of matches. Similar variability was observed with the PROT queries. Short patterns almost always had a very large number of occurrences, regardless of the collection.

5.3 Hardware

All tests were conducted on a 3.0 GHz Intel Xeon CPU machine with 4 GB main memory and 1024 kB L2 Cache and a 320 GB Seagate Barracuda 7200.10 disk. The operating system was Fedora Linux running kernel 2.6.9. The compiler was g++ (gcc version 3.4.4) executed with the –O3 option. Prior to each run, the memory buffer was flushed, so as to provide similar initial conditions. Times were recorded with the standard Unix `getrusage` function. All running times given are the average of at least three runs. The machine was under light load, and no other significant tasks were running concurrently.

The decision as to whether any (and if so, which) parts of the data structures and the string were brought into memory was left to the virtual memory paging scheme of the operating system, and in the software version used for the bulk of the experiments below, referred to as the "general purpose" implementation, the data structures and string were always accessed via `fseek` and `fread` operations. Similarly, for the most part, query sequences were run on a "cold start" basis, that is, with no in-advance buffering of disk files caused by untimed pre-querying. A second "optimized" version that first read the string and data structures into memory was also written and executed on a "hot start" basis; it then accessed all of the LOF-SA components via array accesses and so was significantly faster, but was only applicable on strings for which all components could fit into main (or at least virtual) memory. The general purpose version was capable of handling strings up to the address limit of the 32-bit pointers used.

5.4 Baseline systems

In experimental work of this nature there is a clear obligation to test against good competition. The TDD suffix tree implementation, available at `http://www.eecs.umich.edu/tdd`, maintains both the suffix tree and the collection on disk [27]. Another baseline implementation, TRELLIS, is a relatively recent disk-based suffix tree representation that uses significantly more memory for the suffix tree but offers fast construction and search speeds. However, the collection is stored compressed in main memory using a bit-encoding method that is currently applicable only to the DNA collection. A majority of our experiments with TRELLIS kept the collection in main memory, however we also include results with the collection stored on disk. The TRELLIS implementation was obtained from `http://www.benjarath.com/`.

The suffix array implementation maintains both the collection and the suffix array on disk [5]. Preliminary experiments on the DNA collections showed that a block size of $b = 2048$ and an index of $l = 16$ characters in the SPAT array were appropriate choices.

The LOF-SA is a fundamentally disk-based implementation, with both the LOF trie and the LOF array, as well as the text itself, presumed to be stored on disk, and accessed via `fseek` and `fread`

m	DNA100M	DNA200M	DNA400M	DNA1000M	DNA2000M	PROT	WSJ
6	–	–	138,830,605	–	–	–	–
8	–	–	13,807,301	–	–	–	–
10	690,247	1,265,769	2,877,696	5,828,821	11,529,645	144,543	100,751,877
12	–	–	1,258,962	–	–	–	–
16	–	–	240,512	–	–	–	–
20	49,072	91,921	136,620	475,916	924,207	109,462	87,942,548
24	–	–	46,486	–	–	–	–
32	–	–	10,776	–	–	–	–
40	1,977	4,568	4,323	5,963	16,289	336,401	54,949,826
100	1,061	1,078	1,060	1,211	2,049	2,378	1,380,170
200	1,004	1,002	1,018	1,014	1,017	449,454	1,225
400	1,002	1,000	1,001	1,025	1,006	1,096	1,043
1000	1,001	1,000	1,001	1,001	1,002	1,010	1,000

Table 2: Total number of occurrences of 1,000 different random patterns of length m used in these experiments. Note that for the PROT collection, the undefined symbols are denoted by the symbol X and they have not been replaced with random symbols from the alphabet. Thus, for the query lengths 40 and 200 there are one and two patterns respectively with only the symbol X and these have relatively large occurrences.

Collection	Software version	Start	Queries	Bucket threshold b			
				512	1024	2048	4096
DNA100M	optimized	hot	10^6	0.0029	0.0034	0.0045	0.0065
DNA100M	general purpose	hot	10^6	0.071	0.074	0.079	0.087
DNA100M	general purpose	cold	10^6	1.86	1.40	0.80	0.53
DNA100M	general purpose	cold	10^3	21.4	20.1	19.1	19.6
DNA400M	general purpose	cold	10^3	31.5	26.8	26.9	26.9
DNA1000M	general purpose	cold	10^3	60.3	57.2	50.6	46.8

Table 3: Search time (milliseconds per query) as a function of b, for queries of length 100, averaged over either 1,000,000 patterns (first three rows), or over 1,000 patterns (final three rows). The LOF-SA for the DNA100M collection fits within the main memory of the experimental machine, while that for DNA400M and DNA1000M does not; and the implementation tagged as "optimized" makes use of that fact by reading all data structures into arrays and thereafter accessing components of the data structure via array pointers. On the other hand, the "general purpose" implementation makes use of seek and read operations on files, which slows execution even when (as is the case of the DNA100M dataset, and the hot start) the files have probably been buffered into main memory. As a reference point, a standard suffix array implementation requires 0.310 milliseconds per query on the DNA100M file with a hot start and 10^6 queries (equivalent to the second data row in the table), and 409 milliseconds per query on the DNA1000M file and 10^3 queries with a cold start (equivalent to the last data row).

operations. Except where otherwise specified, a bucket threshold of $b = 4096$ and a fringe length of $f = 4$ were used.

6. EXPERIMENTAL RESULTS

There are two key LOF-SA parameters, the bucket threshold b, which determines the maximum number of leaves in each LOF bucket; and the fringe length f. We first study the parameter space and motivate our choice of values for these two parameters.

6.1 Bucket threshold

When the entire LOF-SA structure is held in memory, which is what automatically happens when the text is short, the bucket threshold b is an important determinant of search performance. As shown in the first row of Table 3, $b = 512$ is around two times faster than $b = 4{,}096$ in the optimized in-memory version, because the sequential scanning within the bucket becomes relatively expensive compared to the access cost of getting to the bucket. The second row of Table 3 shows that the general purpose version – where trie node and bucket accesses are handled via `fread` operations to data that is (for this data file, query sequence, and start mode) presumably buffered elsewhere in memory – is as much as 20 times slower than the optimized version, and less sensitive to b.

When disk costs are taken into account in cold start experimentation (that is, without executing a long stream of queries prior to the commencement of timing), the cost relativities change further, as shown in the third and fourth lines of the table. The relatively high cost of a "true" disk access (that is, one for which the target data is not already buffered elsewhere in memory) means that the cost of sequentially processing large buckets is offset by a reduced number of random accesses to trie nodes. Query performance can thus be expected to gradually improve as b increases, until a point is reached at which the trie becomes small enough to become memory-resident. The relative performance between different bucket sizes is then dominated by bucket processing, as is the case for in-memory applications. The disk-dominated timings (large data file, cold start experiment, and shorter query sequence) in the fifth and sixth rows of Table 3 show exactly the expected pattern, with use of $b = 512$ on the larger dataset giving 28% slower query response than $b = 4{,}096$. Note that the size of the trie is inversely related to b, and so $b = 4{,}096$ gives a trie approximately one eighth the size of the $b = 512$ trie.

That is, when long texts are to be indexed, choosing a large value of b – and hence, needing only a small LOF trie – is not necessarily detrimental to overall performance. In the remainder of our experiments, we set $b = 4{,}096$ as a plausible compromise between search

	Right	Left
DNA400M	28 (1.35)	26 (1.04)
PROT	29 (1.60)	28 (1.24)
WSJ	43 (1.72)	35 (1.11)
DNA1000M	45 (1.38)	34 (1.04)

Table 4: Search time in milliseconds per query, as a function of the fringe justification for queries of length 100, using the general purpose implementation, 10^3 queries, and a cold start. The average per-query number of accesses to the text is shown in the parentheses. In these experiments a buffer threshold of $b = 4{,}096$ and a fringe of $f = 4$ were used.

performance and main memory usage by the LOF trie. The LOF array will typically always be disk resident except for small data files. For example, in Table 3 the transition point at which the trie becomes fully memory-resident for the DNA400M file can be seen at approximately $b = 1{,}024$.

6.2 Fringe justification

Two fringe justification approaches were discussed above. Right justification allows gaps to develop between successive LOF entries, and results in the text being accessed in some instances without there being unambiguous evidence of a match. In contrast, the left justified approach eliminates the gaps, and the text is accessed only if the query cannot be entirely consumed by the fringe characters in the bucket.

Table 4 shows that use of the RJ approach reduces accesses to the text to less than 2 per query, and that the left justified approach is even better. These are striking results that clearly show the significance of the fringe characters in terms of reducing the number of expensive text accesses, and were echoed in experiments (not shown here for space reasons) on the other test texts, and with other combinations of parameters.

Note that even when small texts are being indexed, text accesses are relatively costly. Compared to the cost of a sequential access within a memory-resident LOF array, a random access into a memory-resident text is likely to be costly, because the latter will almost certainly involve a cache miss. The effect of the reduction of text accesses is reflected on the search times in Table 4, where the RJ approach is typically slower than the LJ approach even for the small collections. For example, using the optimized-for-memory implementation when processing texts of size 100 MB showed that the LJ approach was still faster than the RJ approach, and that instruction cost of the relatively complex LJ search algorithm is offset by fewer text accesses. Hence, the LJ approach is preferred to the RJ approach for both small and large data, and in both in-memory and disk-based implementations. All remaining experiments in this paper make use of the LJ fringe approach.

6.3 Fringe length

The fringe reduces text accesses at the cost of more space in the LOF array. Table 5 shows the average search time in the DNA400M collection with varying values for f, the fringe length. As expected, increasing the number of fringe characters decreases query times, because the number of text accesses has decreased. In a range of experiments with different texts and query lengths, it was clear that $f = 4$ was a suitable compromise, and increasing it further does not necessarily improve query times. The turning point arises because the increased cost of processing larger buckets is not fully offset by the small additional reductions in text accesses that are achieved by very long fringes.

Attribute	Fringe length f			
	0	2	4	8
Search time (milliseconds)	53	27	26	30
Text accesses	10.52	1.11	1.04	1.01

Table 5: Search time in milliseconds, and average text accesses per query, as a function of the fringe length for a query length of 100 using the DNA400M text, the LJ approach, and $b = 4{,}096$. In all cases the general purpose implementation, 10^3 queries, and a cold start were used.

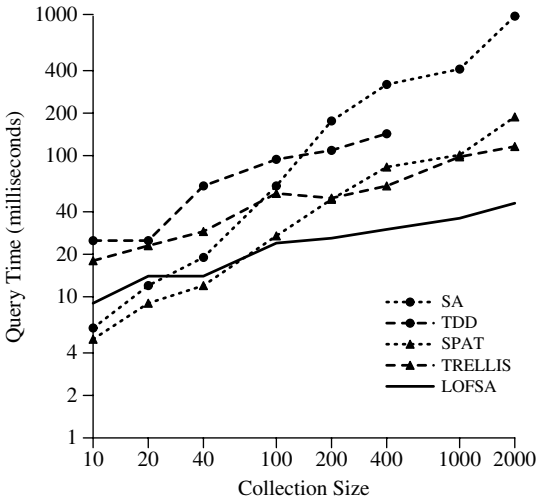


Figure 6: Search time in milliseconds per query, as a function of the collection size for queries of length 100 on various prefixes of the DNA collection. In all cases 10^3 queries and a cold start were used; the LOF-SA version is the general purpose implementation at all data points.

The same effect was measured using the optimized-for-memory implementation, in which increasing the number of fringe characters beyond $f = 4$ did not noticeably improve performance. As $f = 4$ offers the best all-round compromise between space and performance, it is used in the remainder of the experiments.

6.4 The effect of text size

Having set the parameters that determine the structure of the LOF-SA, we now compare it with a range of competitor techniques. To study how query performance varies with text size, we used prefixes of the DNA collection ranging from DNA10M to DNA2000M. Figure 6 shows how the LOF-SA compares to other string indexing methods over this range, measured as the average query time for queries of length 100. (The effect of query length variation is explored in the next subsection.)

The suffix array gives the slowest querying on large inputs (note the logarithmic time scale in Figure 6), with performance greatly hindered by the large number of disk accesses required by the binary search process. The SPAT is over 5 times faster than the suffix array for queries of length 100, a consequence of a much smaller number of disk accesses. The TDD implementation we used was unable to accommodate the larger test files, and it was slow even on the mid-sized ones. The TRELLIS software is two times faster than TDD for DNA400M, the largest file on which the TDD executed. We also experimented with keeping the text on

Index structure	DNA text length n, in millions of bytes			
	200	400	1000	2000
SA	106.63	110.55	115.71	119.65
SPAT	26.58	26.79	27.12	27.83
TDD	37.79	40.51	–	–
TRELLIS (M)	15.69	15.17	16.02	17.36
LOF-SA	2.03	2.04	2.03	2.03
LOF-SA (+TRIE)	10.73	11.36	12.30	12.98

Table 6: Average per-query non-sequential data accesses made, as a function of the collection size, for queries of length 100.

disk for TRELLIS, and observed a slowdown in search times by over 60%.

For all except the very small collections, the LOF-SA provides the fastest querying. It also scales well with increasing collection size, with querying cost growing by a factor of just two over a text size growth of twenty. For the smaller text sizes, the LOF-SA is slightly slower than both the suffix array and the SPAT array. This is a consequence of the relatively larger LOF-SA structure and the buffer threshold b having been set to 4,096 throughout the experiments reported in Figure 6. Smaller access structures would quickly have become fully memory resident on the test machine and the use of a smaller value for b would have allowed faster searching (see Table 3). The optimized in-memory implementation of the LOF-SA (with $b = 256$ and $f = 4$) is typically around four times faster than the suffix array implementation obtained from the Pizza/Chili Corpus web site at `http://pizzachili.dcc.uchile.cl/`.

Table 6 shows the number of potential disk accesses to text and index structures during querying, for the same suite of techniques. For the LOF-SA, the total disk accesses are shown in two rows; the second row includes the accesses to trie nodes, as well as the bucket access and the accesses to the underlying text. Note that in preparing this table, an "access" was counted every time the locus of file operation changed in a non-sequential manner. That is, we counted an access even when (for example) the disk or memory address in question fell within a short distance of the current activity site. These numbers are thus overestimates of the true number of disk seeks, but reflect what would happen if main memory availability was highly constrained.

The LOF-SA incurs one or more of these "disk" accesses at each level of the trie traversal stage; another as the bucket is read; and a further one every time the underlying text is checked. However, if the LOF trie becomes memory-resident, the disk accesses are restricted to the bucket and the collection. That is, when the trie is small, accesses to it are mostly restricted to the first few queries, as a startup cost. Also notable in Table 6 is that the average number of accesses made by the LOF-SA to the text and bucket is stable with increasing collection size, requiring typically one access to a bucket and then one to the text.

In comparison, the standard suffix array incurs two non-sequential accesses at each step of the binary search process – one into the suffix array itself, and another to access the underlying string for the purposes of carrying out the string comparison. Moreover, two binary search processes are needed in the implementation used as a reference point, one to establish the lower end of the range of match positions, and one to establish the upper end of the range. In terms of the metric assessed in Table 6, the suffix array does particularly poorly; and Figure 6 shows that for large datasets non-sequential accesses definitely translate into large running times.

Collection and index structure	Query length m, in characters				
	10	20	40	100	200
File DNA400M					
SA	308	303	317	319	321
SPAT	75	83	87	84	84
TDD	59	132	142	143	145
TRELLIS (M)	237	70	63	61	61
TRELLIS (D)	240	101	95	94	93
LOF-SA	13	26	30	30	30
File PROT					
SA	138	148	143	144	140
SPAT	40	42	42	42	43
TDD	126	131	134	135	135
LOF-SA	19	26	28	29	30
File WSJ					
SA	282	298	292	293	301
SPAT	76	79	79	79	78
TDD	933	1002	979	916	909
LOF-SA	21	33	40	41	40

Table 7: Search time in milliseconds per query, as a function of the query length for the DNA400M, PROT, and WSJ collections. In all cases 10^3 queries and a cold start were used. The LOF-SA version is the general purpose implementation at all data points; it consistently outperforms the other methods, across the spectrum of text types and query lengths.

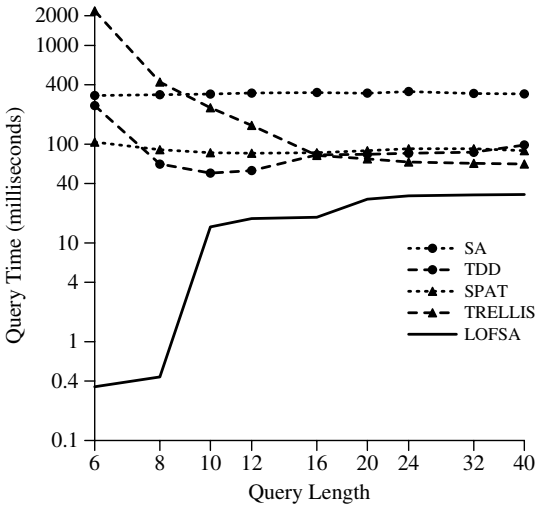


Figure 7: Search time in milliseconds per query, as a function of the query length, for the DNA400M collection. In all cases 10^3 queries and a cold start were used; the LOF-SA version is the general purpose implementation.

Figure 6 and Table 6 provide compelling evidence that the LOF-SA is, as claimed earlier, an excellent data structure for handling what is a fundamental pattern matching problem.

6.5 The effect of query length

In the next round of experiments the length of the patterns being queried was varied, to verify that pattern length did not give rise to performance issues with the LOF-SA. The results from these

b	DNA 400	DNA 1000	PROT 223	WSJ 471
1024	13.53	38.15	19.18	68.37
2048	6.34	17.31	10.03	37.85
4096	3.25	8.44	4.60	20.76
8192	1.51	4.02	2.47	11.00

Table 8: Memory usage of the LOF trie in megabytes, for the three text collections (including two prefixes of DNA), using a range of bucket threshold sizes b.

experiments are shown in Table 7 and in Figure 7. These results demonstrate that LOF-SA requires the least time per query across a very broad range of query lengths, for all the three collections.

Typically, for query lengths over 40, all LOF-SA searches require a bucket access and one access to the text for each answer found. As queries get shorter, an increasing fraction of them are answered using only the LOF trie, without any accesses to the text being required at all. Figure 7 shows this effect, with queries of six and eight symbols handled extremely rapidly, without bucket searching being required – all contents of a set of buckets represent occurrences of the pattern, meaning that the number of occurrences can be reported out of the count fields within the trie node, without needing even any access to a LOF bucket. If the matching text positions are also required, the indicated LOF buckets can be accessed, and the offsets extracted during a sequential scan. Thus, the LOF-SA structure is well suited for both small and long query lengths for the pattern matching problem.

This smooth balance is the strength of the LOF-SA. For small query lengths, the occurrence count can be found from the trie nodes; whereas for longer queries, establishing the presence (or not) of the pattern typically requires one access to a LOF bucket and approximately one access to the text. Nor is it possible to make long queries more expensive via the use of a pathological input text – any long strings that repeat sufficiently many times in the input text automatically get allocated a dedicated node in the LOF trie.

The results in this section also demonstrate that even a short fringe length of $f = 4$ allows the LOF-SA to scale well with query length. Note also that in our experiments the patterns were processed one by one, in a completely random order. The query performance of the LOF-SA can be improved further by processing the queries in batches so that the buckets (and even the text) are accessed in address order. A more principled discussion of batched querying is left for future work.

6.6 Disk accesses

Table 6 showed that the LOF-SA has a largely sequential access pattern, and that if only non-sequential memory accesses are counted, then it is efficient compared to other structures. In this sense, it articulates smoothly from all-in-memory performance, when non-sequential access results in cache misses, to all-on-disk performance, when non-sequential accesses result in disk seeks. In fact the greatest number of non-sequential memory accesses is in the LOF trie, and so establishing the size of the trie – and verifying that it is modest – is also an important aspect of our experiments.

6.7 Memory usage

The bucket threshold b directly determines the size of the trie for the LOF-SA – the larger the value of b, the fewer trie nodes are required, and the smaller the trie. Table 8 shows the relationship between b and total trie size, for all three of the experimental texts. When $b = 4{,}096$, the trie remains smaller than 21 MB, even for the large 2 GB DNA file. Even for the rich-alphabet WSJ file the trie takes less than 5% of the space of the source text. It is this compactness that gives us the assurance to comment, as we did above, that when there is even the slightest amount of main memory available, and a stream of queries is being processed, the trie will migrate into main memory.

Collection	SA	SPAT	TDD	TRELLIS	LOF-SA
DNA400M	1,526	1,529	4,345	11,221	4,581
PROT	889	891	2,773	–	2,672
WSJ	1,883	1,887	13,166	–	5,671

Table 9: Total space requirement in megabytes, of the various index structures tested, not including the string being indexed. The LOF-SA uses $b = 4{,}096$ and $f = 4$.

Table 9 shows the total memory usage incurred by the various index structures on the three collections DNA, PROT, and WSJ, not including the cost of storing the collection itself. The SA technique requires only the suffix array, of four bytes per pointer. From there the sizes grow – the SPAT requires an additional small array as well as the suffix array, and takes a little more space. At the other end of the spectrum, the TDD and TRELLIS implementations both require space for a suffix tree. In between is the LOF-SA, which requires memory for the LOF trie and for the LOF array. To a limited extent the LOF-SA is adaptable to memory constraints, and the size of the LOF trie can be reduced based on the b value, while the size of the LOF array can be reduced based on the fringe length f. Moreover, the fringe characters can be stored bit-compressed for small alphabet collections such as DNA, and we are also exploring more general compression methods applicable to all texts. Even so, there is a minimum space requirement below which the LOF-SA cannot be forced, and that limit is greater than the cost of the suffix array.

The fact that a suffix array requires $5n$ bytes to index a string of n bytes, whereas the LOF-SA (in current implementation) requires more than $13n$ bytes, means that on any given computer there is a defined range of data sizes in which a suffix array can be fully memory-resident, but the LOF-SA cannot. For example, we also experimented using a different machine with just 1 GB of memory, and found that on DNA100M a suffix array could cold-start process a sequence of a hundred thousand 100-symbol queries in 4.6 milliseconds each, whereas the LOF-SA required 7.6 milliseconds under the same conditions. (Worth noting is that both were several orders of magnitude faster than direct string searching over the n-byte sequences using sequential pattern matching approaches.)

That is, the LOF-SA is a structure that supports fast string searching, but its speed is attained via a non-trivial storage space requirement. We reiterate, however, that this space is required on disk, not in main memory, and that even in the cross-over zone, it remains relatively efficient.

6.8 Construction costs

Once the suffix and LCP arrays are available, the LOF array can be created in a single-pass, accessing the underlying text as necessary to obtain the fringe characters. As mentioned earlier, the LOF-SA construction builds on any of the efficient suffix tree construction algorithms that have been developed. On the DNA100M collection, the extraction of the suffix and LCP arrays takes about 94 seconds on the experimental hardware. The construction of the LOF array from the SA and the LCP arrays is linear in the size of the collection and requires 255 seconds for DNA100M and 2551 seconds for DNA1000M. The LOF trie structure can then be con-

structed on top of the LOF array, in a further sequential pass. The time to construct is inversely proportional to the block size b. For example, for the DNA1000M collection, it took 720 seconds with $b = 4{,}096$, and 499 seconds for $b = 8{,}192$.

7. CONCLUSIONS AND FUTURE WORK

We have described the LOF-SA data structure for fast indexed pattern matching on large texts. Our approach allows querying at speeds of up to three times faster than the SPAT suffix array arrangement, and up to twenty times faster than the disk-based TDD suffix tree. The LOF-SA resides almost entirely in external memory, and consists of a small trie that indexes a sequential list of suffix pointers, LCP values, and fringe characters. The underlying source text is also held in secondary memory. The LOF-SA can be thought of as a suffix tree or an enhanced suffix array with improved locality of memory reference – it contains the same information as those data structures, and can be built from either one with little overhead.

Future work will investigate the use of the LOF-SA for tasks other than pattern matching such as repeat detection and approximate matching (see [1, 13, 26]). One would also expect significant speed ups in these applications as the references to adjacent SA/LCP values will incur reduced overheads as a result of their locality in the LOF-SA. We also note that the current LOF-SA arrangement does not incorporate the suffix links required by some applications, and this is another avenue for further investigation. Applying the LOF-SA to sequences composed of words rather than characters is also an obvious area that awaits our attention.

We expect that, in time, external memory algorithms for direct suffix array construction will become faster than their suffix tree counterparts, as has become the case in primary memory. It remains an open problem to efficiently recover LCP information needed for the enhanced suffix array from a disk resident suffix array. It may well be that the only feasible solution is to compute LCP information during suffix array construction.

We also plan to explore ways of reducing the space cost of the LOF-SA, by representing the LCP values and fringe characters in compressed form; and further increasing query speeds by examining different interleaving options for the data within each bucket.

Finally, great strides have been made recently in the area of compressed suffix arrays [22]. It would be interesting to see how the ideas described in this paper might be used to make these compressed data structures more amenable to use on disk.

Acknowledgments. We thank the authors of TDD [27] and TRELLIS [23] for making their code available. This work was supported by the Australian Research Council.

8. REFERENCES

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

[2] M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *Proc. String Processing and Information Retrieval Symp.*, pages 31–43. Springer Verlag, Berlin, Germany, 2002.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Jour. of Molecular Biology*, 215(3):403–410, Oct. 1990.

[4] A. Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, NATO ASI Series F12, pages 85–96. Springer Verlag, Berlin, Germany, 1985.

[5] R. A. Baeza-Yates, E. F. Barbosa, and N. Ziviani. Hierarchies of indices for text searching. *Information Systems*, 21(6):497–514, 1996.

[6] D. Benson, D. J. Lipman, and J. Ostell. Genbank. *Nucleic Acids Research*, 21(13):2963–2965, 1993.

[7] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler. Genbank. *Nucleic Acids Research*, 35:D21–D25, Jan. 2007.

[8] M. Cameron, H. E. Williams, and A. Cannane. Improved gapped alignment in BLAST. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1(3):116–129, 2004.

[9] C. Cheung, J. X. Yu, and H. Lu. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):90–105, 2005.

[10] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, Jan. 2002.

[11] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM Jour. of Experimental Algorithmics*, 2007. To appear.

[12] Ensembl. Ensembl Genome Browser, 2006. http://www.ensembl.org.

[13] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, Cambridge, UK, 1997.

[14] D. Harman. Overview of the second text retrieval conference (TREC-2). *Information Processing and Management*, 31(3):271–289, 1995.

[15] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. Annual Symp. on Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer Verlag, Berlin, Germany, 2001.

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

[17] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.

[18] M. A. Maniscalco and S. J. Puglisi. Faster lightweight suffix array construction. In J. Ryan and Dafik, editors, *Proc. Australasian Workshop on Combinatorial Algorithms*, pages 16–29, 2006.

[19] G. Manzini. Two space saving tricks for linear time LCP computation. In T. Hagerup and J. Katajainen, editors, *Proceedings of 9th Scandinavian Workshop on Algorithm Theory (SWAT '04)*, volume 3111 of *Lecture Notes in Computer Science*, pages 372–383. Springer Verlag, Berlin, Germany, 2004.

[20] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40:33–50, 2004.

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

[22] G. Navarro and V. Mäkinen. Compressed full text indexes. *Computing Surveys*, 39(1), 2007.

[23] B. Phoophakdee and M. J. Zaki. Genome-scale disk-based suffix tree indexing. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 833–844, New York, NY, USA, June 2007. ACM Press.

[24] S. J. Puglisi, W. F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *Computing Surveys*, 39(2):1–31, 2007.

[25] J. S. Sim, D. K. Kim, H. Park, and K. Park. Linear-time search in suffix arrays. In M. Miller and K. Park, editors, *Proc. Australasian Workshop on Combinatorial Algorithms*, pages 139–146, Seoul, Korea, 2003.

[26] B. Smyth. *Computing Patterns in Strings*. Addison-Wesley, Reading, Massachusetts, USA, 2003.

[27] Y. Tian, S. Tata, A. Hankins, and M. Patel. Practical methods for constructing suffix trees. *The VLDB Jour.*, 14(3):281–299, 2005.

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

[29] P. van Emde-Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[30] P. Weiner. Linear pattern matching algorithms. In *Proc. Annual Symposium on Foundations of Computer Science*, pages 1–11, Silver Spring, MD, USA, 1973. IEEE Computer Society Press.