On Performance and Cache Effects in Substring Indexes

Norwegian University of Science and Technology
Technical Report IDI-TR-2007-04

ISSN: 1503-416X

Nils Grimsmo*
nilsgri@idi.ntnu.no

Department of Computer and Information Science
Sem Sælands vei 7-9
NO-7491 Trondheim
NORWAY

2007-05-02

*Supported by the Research Council of Norway under the grant NFR 162349.

Abstract

This report evaluates the performance of uncompressed and compressed substring indexes on build time, space usage and search performance. It is shown how the structures react to increasing data size, alphabet size and repetitiveness in the data. The main contribution is the strong relationship shown between time performance and locality in the data structures. As an example, it is shown that for a large alphabet, suffix tree construction can be speeded up by a factor 16, and query lookup by a factor 8, if dynamic arrays are used to store the lists of children for each node instead of linked lists, at the cost of using about 20% more space. And for enhanced suffix arrays, query lookup is up to twice as fast if the data structure is stored as an array of structs instead of a set of arrays, at no extra space cost.

1 Introduction

The *suffix tree* is a versatile substring index, first introduced by Weiner [Wei73] (see more accessible descriptions by McCreight and Ukkonen [McC76, Ukk95]). It can be used to search for occurrences of patterns in a string, and solve many other problems in combinatorial pattern matching [Gus97]. The suffix tree and similar structures are mostly used in computational biology, where the sequences considered do not have word boundaries, and methods such as inverted lists are not suitable.

1.1 Suffix Tree Definition

A suffix tree for a string T of length n from the alphabet Σ of size σ, is a trie of all $n+1$ suffixes of the string padded with a unique terminal, edge compacted such that every internal node is branching. Since the padded string has $n+1$ suffixes, and the unique terminal ensures that no padded suffix is a prefix of another, the tree has $n+1$ leaf nodes. There are at most n internal nodes, since they all have at least two children. This means that if edges are represented as pointers into the string, the suffix tree needs $\Theta(n)$ space measured in computer words, or $\Theta(n \log n)$ bits, which is slightly more than the optimal $\mathcal{O}(n \log \sigma)$ bits, the space needed to store the string indexed.

In addition to the parent–child edges, each internal (non-root) node has a *suffix link* to another internal node [McC76]. If the edges from the root to an internal node spells $\chi\alpha$, where χ is a string of length 1, this node has a suffix link to the internal node which represents the string α. These links are used in construction algorithms, and to solve some problems, such as *longest common substring* [Gus97].

A suffix tree can be built in $\Theta(n)$ time, when the alphabet size is constant [Wei73] or integer [Far97]. As with any trie, it can be checked if a pattern P of length m is contained in the tree in $\mathcal{O}(m)$ time, if node child lookup is $\Theta(1)$. Since all leaf nodes below the tree position found in such a lookup represent unique matches, and all internal nodes are branching, at most $2z-1$ nodes must then be visited to find all z matches, giving a total time of $\mathcal{O}(m+z)$. The problem of finding all matches of a pattern in a string or a set of strings in known as the *occurrence listing problem.* A suffix tree has asymptotically optimal construction time for integer alphabets, and optimal search time for constant alphabets. Optimal search time for larger alphabets can be achieved with perfect hashing, at the cost of a longer construction time [FKS84].

A *generalised suffix tree* [Gus97] for a set of strings $S = \{T_1, \ldots, T_d\}$, is an edge compacted trie of all suffixes of each string T_i padded with a unique terminator string $\$_i$. A string T_i of length n_i can be added to the tree in $\Theta(n_i)$ time by slightly modifying a suffix tree construction algorithm, and can be removed in $\Theta(n_i)$ time.

1.2 Alternatives to Suffix Trees

A high constant factor in the space needed for suffix trees make them unsuited for solving problems with large strings on commodity computers, as they do not work well on disk [BH04]. The space efficient representation by Kurtz [Kur99] needs 13 bytes per string symbol on average[1]. This has lead to the development of many alternative index structures.

The *suffix array* [MM93] is a simplification of suffix trees, consisting only of an array A with the sorted order of the suffixes of the padded string, such that $T[A[i] \ldots n+1] < T[A[i+1] \ldots n+1]$ for all $1 \leq i \leq n$. The order in A is equal to the or-

[1]When supporting strings up to 500MB

der of the leaf nodes seen in an ordered depth first traversal of the suffix tree for the string. The space usage is $n\lceil \log n \rceil$ bits, plus $n\lceil \log \sigma \rceil$ bits for storing the text. Lookup by binary search is $\mathcal{O}(m \log n)$, which can be improved to $\mathcal{O}(m + \log n)$ by using additional arrays storing longest common prefixes (LCP) between the suffixes indexed [MM93]. After finding the left and right border of suffixes matching the pattern, the hits are read by a sequential scan in the array. Suffix arrays can be constructed in $\Theta(n)$ time [KS03, KSPP03, KA03], but algorithms with higher worst-case bounds are usually faster in practice [MF04, SS05].

A "hybrid" between suffix arrays and trees is the *enhanced suffix array*, which has the same functionality as suffix trees, but uses less space in the worst case. It is a suffix array with additional fields, and can be built in linear time. Abouelhoda et al. [AKO04] show how to use the enhanced suffix array to replace any algorithm doing top–down, bottom–up or suffix link traversal in a suffix tree. The steps of looking up a pattern is similar to what is done in a suffix tree implemented with sibling lists. Listing hits is done as in suffix arrays, by sequential scan between a left and right border. Enhanced suffix arrays are expected to list large numbers of hits much faster than suffix trees in practise, due to the improved locality.

An alternative non-linear suffix tree construction is the *write-only top-down* construction (*wotd*) [GKS03], in which the suffix tree is constructed in $\mathcal{O}(n^2)$ time (expected $\Theta(n \log n)$). Sibling nodes are stored adjacently in memory. This spatial locality makes it faster than linear time suffix trees for some cases.

Many *compressed* substring indexes have been introduced the last ten years. See [NM06] for an introduction, time and space bounds, and an extensive list of references. The family of LZ-indexes [KU96, FM05, Nav03] use structures based on the Ziv-Lempel decomposition of the string [ZL77]. Other indexes are based on suffix arrays, such as the family of *Compressed suffix arrays* [Mäk00, GV00, Sad03, GGV03, MN04a]. The *FM-index* family [FM00, GMN04, FMMN04, MN05] also build on suffix arrays, but uses a different search method. These structures offer various tradeoffs between index size, construction time, and query performance. Many of these structures do not depend upon keeping a copy of the text, and are hence called *self indexes*.

Sadakane [Sad07] has shown that it is also possible to combine a compressed suffix array with a balanced parenthesis representation of a suffix tree and various other structures to get full functionality, with bottom–up, top–down and suffix link traversal of the tree.

1.3 Dynamic Problems

Suffix trees have been equalled by other structures in terms of construction time and search speed, and surpassed on space usage and practical performance when listing many hits. The only problems in which suffix trees have not been matched (at least in asymptotic terms, to the authors knowledge), are problems concerning dynamic sets of strings, in which strings are added to and removed from the set. Generalised suffix trees can be used to solve many of these problems optimally.

Problems with static sets of strings can be solved by using suffix arrays or similar structures, by indexing the concatenation of the strings, separated by a unique symbol $\notin \Sigma$. Also, any dynamic decomposable indexing problem can be solved by using a hierarchy of static indexes of varying sizes, and the cost of a $\mathcal{O}(\log |S|)$ overhead on build time and search performance [OvL80]. Grimsmo [Gri05] shows the performance of hierarchies of suffix arrays compared to a generalised suffix tree.

A problem related to the occurrence listing problem is the *document listing problem* for sets of strings. Given a pattern, find all documents in which it occurs. Muthukrishnan [Mut02] shows how to solve this problem optimally by using a suffix tree with additional information. The technique used can be adapted to suffix arrays and similar structures. An open problem (to the authors knowledge) is solving the document listing problem optimally for dynamic sets of documents.

1.4 Report Overview

Section 2 details how to efficiently implement a suffix tree using dynamic arrays for the parent–child relationship, with a low space overhead compared to sibling lists.

Section 3 describes some of the choices made in the tested implementation of enhanced suffix arrays.

Section 4 features extensive tests of many substring indexes, on build time and search performance, with many types of test data. It is shown how the indexes react to increasing data size, varying alphabet size, and varying text randomness.

Section 5 draws conclusions from the experiments, and gives some guidelines for when the various types of index structures are suitable.

2 Implementation of Suffix Trees

The suffix tree data structure consists of a set of nodes. These nodes are linked together with parent–child edges and suffix links. One of the major choices when implementing suffix trees is how to represent the parent–child relationship. McCreight [McC76] describes the use of linked lists (called sibling lists) and hash tables. His article states that using an array of pointers "would be fast to search, but slow to initialise and prohibitive in size for a large alphabet." This was true in 1976.

2.1 Erroneous Assumptions on Using Arrays for Child Lists

Various authors have made assumptions about the efficiency of different ways to implement suffix trees. Bedathur and Haritsa [BH04] claim that using arrays would result in wasted space, as there would be a lot of null pointers. They say this would be especially severe in the lower parts of the tree. This implies that the authors do not consider the possibility of storing the pointers in an unsorted array and using dynamic arrays, or that they consider such a solution to be inefficient. Tian et al. [TTHP05, page 288] make similar claims, referring to [McC76] and [BH04].

This report presents results showing that using dynamic arrays for storing the parent-child relationship clearly outperforms using sibling lists in terms of speed, at the cost of using slightly more space.

2.2 Parent-Child Relationship

Suffix tree construction is usually described as being $\Theta(n)$, with the assumption that the size of the alphabet can be viewed as a constant. The most common way of implementing suffix trees is using sibling lists. This is a simple solution, which gives a construction time of $\mathcal{O}(n\sigma)$, which is effective for small alphabets, such as DNA. This alphabet factor is most visible in trees for highly random strings from large alphabets, where there are fewer nodes, but a higher average branching factor.

When considering general alphabets (sorting only possible by comparison), the running time of any suffix tree construction algorithm has a worst case $\Omega(n \log n)$, as it has sorting complexity [FCFM00]. Farach [Far97] shows how to construct suffix trees for integer alphabets ($\sigma \leq n$) in $\Theta(n)$ time by recursively constructing a suffix tree for odd numbered suffixes, building a tree for the even numbered from the information found, and then merging these trees.

McCreight [McC76] proposed to use hashing as an alternative to sibling lists. Kurtz [Kur99] shows how to do this effectively. Since there is an initial space overhead for each hash table, a single table is used for all nodes in the tree. The keys in this table are pairs of parent node numbers and edge first symbols, and the values are child node numbers. With this scheme, the expected construction time is $\Theta(n)$, but finding all occurrences when searching can be very slow, as a lookup for children must be done on all possible symbols in Σ. With sibling lists, all z occurrences are found in $\mathcal{O}(m\sigma+z)$, while with a hash map the expected time is $\mathcal{O}(m + z\sigma)$. It is possible to implement a combination of a hash map and a linked list, where the children of a node are linked together, as shown by Grimsmo [Gri05]. The proposed structure uses less space than what is needed for the combination of the sibling lists and the hash table, but it is complex and slow in practice. Simply using both sibling lists and hashing would probably be faster, at the cost of using more space.

Bedathur and Haritsa [BH04] propose storing all child pointers inside the nodes in their disk-based suffix tree construction for very small alphabets ($\sigma = 4$). However, when using this approach, the array storing the pointers must be of a constant size.

2.3 Sibling List Implementation

The implementation by Kurtz [Kur99] is the fastest space efficient implementation of linear time suffix

trees known to the author. Nodes are stored as integer fields in an array, and node references are indexes into this array. The following layout is used for internal nodes:

- *first-child* – Pointer to first node in list of children
- *branch-brother* – Pointer to the next sibling
- *suffix-link* – Pointer to the node $\overline{\alpha}$, if this is node $\overline{\chi\alpha}$.
- *head-position* – The starting position of the second suffix in the string represented in this node.
- *depth* – The depth of the node.

The fields *head-position* and *depth* are used to find the string the incoming edge represents (see [Kur99]). These are used instead of start and end positions because they do not change for a given node during the construction. The number of internal nodes is not known in advance, and are therefore kept in a dynamic array. There are always exactly $n + 1$ leaf nodes, which can be stored in a static array. The only field needed in leaf nodes is the *branch-brother* pointer.

As the space needed is at most $5n$ words for the internal nodes, and n words for the leaf nodes, an upper limit for the total space needed for the suffix tree is $6n$ words. A word must be $\lceil \log n \rceil + 1$ bits to index all nodes and string positions. A total of $24n$ bytes is needed if 32-bit words are assumed, plus n bytes for storing the string. Kurtz shows how to reduce this to $20n$ bytes, by storing *suffix-link* in the *branch-brother* field of the last child. Another trick shown is using *small nodes*, which are internal nodes with fewer fields, exploiting redundant information in the tree. On average, the space usage is about $13n$ bytes, when supporting strings up to 500 MB.

When the number of children per node is high, child lookup is a costly part of tree traversal, and even more so on a computer architecture where cache misses are expensive. When traversing a sibling list, two cache misses are expected per child visited: one for looking up fields in the node, and one for extracting the first symbol on the incoming edge to the child.

2.4 Child Array Implementation

On modern computers, a miss in level 1 cache costs around 20 cycles, while a miss in level 2 cache costs around 200 cycles, more than four times the cost of an integer division [Int06] (and more if pipelined instructions are considered). This implies that using dynamic arrays and copying could be preferable in many applications where linked lists were previously considered the best option. Below follows a description of how to implement suffix trees with child arrays efficiently.

Each internal node refers to a child array. There exist child arrays of a set of pre-defined sizes. Arrays of the same size are stored together in a container, giving allocation and de-allocation in $\Theta(1)$ time by storing free locations in a linked list, with the pointers saved inside the free slots. When a node needs room for more children, a new child array of larger size is allocated, the child pointers copied here, and the old array released.

Table 1 shows the layout of nodes in the used implementation. Two new fields replace *first-child* and *branch-bro* in internal nodes. Which child array size is used is given in *carr-size*, while the position of the child array in the container is given in *carr-pos*. Since there is no *branch-brother* field, leaf nodes do not need to be explicitly represented at all. The reasoning is the same as for a hash table implementation [Kur99]. The fields *in-symb* and *in-ref* listed in the table is the space needed for each node in its parents child array. In a traditional sibling list implementation, a lookup in the string is required for each child considered in lookup on a specific symbol. This is avoided here by storing the first symbol interleaved with the node references. This reduces the number of cache misses.

While an internal node is identified by its index in memory, a leaf node can be identified by its *head-position*. The *depth* can be found by subtracting this from the length of the string. A bit in each node reference is needed to distinguish between internal and leaf nodes. Lookup in the string while traversing the children can also be avoided, by storing the first symbol on incoming edges interleaved with the node references. As relatively few cache misses are expected compared to the number of nodes considered in child traversal, this should prove more efficient than sibling lists.

The term *array doubling* is often used when de-

Table 1: Node layouts with child arrays

Large	**B**	**Small**	**B**	**Leaf**	**B**
carr-size	1	*carr-size*	1		
carr-pos	4	*carr-pos*	4		
suffix-link	4	*dist*	1		
hpos	4				
depth	4				
Child array space usage					
in-symb	1	*in-symb*	1	*in-symb*	1
in-ref	4	*in-ref*	4	*in-ref*	4
Sum	22	**Sum**	11	**Sum**	5

scribing dynamic arrays, but is a bit misleading, as the growth factor can be different from 2. The amortised asymptotic cost of insertion into the array is $\Theta(n)$ for any constant growth factor. If a growth factor of 2 is used, and n values are inserted into a dynamic array, the worst case total number of insertions and re-insertions is about $3n$. If the growth factor is 1.1, the worst case is about $12n$. Because of the cache effects on modern computers, copying is very cheap, and using a low growth factor is affordable.

The total space needed for this implementation, disregarding overhead from dynamic arrays, is $27n$ bytes in the worst. This is $27/20 = 1.35$ times more than for sibling lists. In practice, the space usage is lower, as the number of internal nodes is usually around $0.5n$ to $0.7n$ [Kur99]. The space wasted internally in each child array should also be considered, which comes in addition to the overhead due to the growth factor in the storage for the groups of child arrays. With a growth factor of 1.1, the cost for child arrays is $17 \cdot 1.1 + (5+5) \cdot 1.1^2 \approx 30.8$, while for sibling lists it is $16 \cdot 1.1 + 4 \approx 21.6$. This gives a ratio of $30.8/21.6 \approx 1.43$. This is also lower in practice, as the out-degree of internal nodes often has a very skewed distribution, which can be taken into account when configuring the pre-defined child array sizes.

3 Implementation of Enhanced Suffix Arrays

An enhanced suffix array is also tested in the following experiments. The implementation is the authors translation of the pseudo-code from Abouelhoda et al. [AKO04] into C++. Some choices for the data structures affect the performance, especially query lookup, and are therefore discussed here.

The data structures needed for substring search with an enhanced suffix array is the suffix array itself, the LCP table and the child table (CLD), all of which have the same length. The first choice to be made in an implementation is whether to store the three as separate arrays, or as an array of structs with three fields. As the fields for a given "position" in the enhanced suffix array are often read together during search, the latter choice should give better locality and performance.

Abouelhoda et al. describe how store the LCP table in a byte array, overflowing into another data structure, where lookup is done by binary search. This was not done here, because it gave a considerable slowdown on some of the benchmarks, as they had a high average LCP. On the file `w3c2` (see Table 2), there was a 36% overflow with 1 byte LCP fields, and 11% overflow with 2 bytes.

The CLD table can also be stored in an overflowing byte array [AKO04]. This gave a very low overflow on the tests run here, but still a considerable slowdown during search. This is because the "nodes" close to the root of the virtual tree are the most likely to overflow, as they "span" larger portions of the enhanced suffix array. Even though a small portion of all nodes overflow, a large portion of the nodes traversed during a query lookup are expected to have overflowing CLD fields.

In the implementations tested, all three mentioned fields were stored as four byte integers, resulting in a space usage of $3 \cdot 4n + n = 13n$ bytes including storage for the text itself. If the LCP and CLD fields are stored as overflowing bytes, the expected space usage is $4n + 3 \cdot n = 7n$ bytes. Performance for an implementation where the CLD values were stored in bytes overflowing into a hash table is also given in some of the tests on query lookup in Section 4.7.

For the tests on tree traversal operations in Section 4.9, two additional fields were used. The "left" and "right" suffix link pointers where stored in $2 \cdot 4n$ bytes, and the inverse suffix array used to create them was stored in $4n$ bytes. [AKO04] describes how to store the suffix link pointers in expected $2 \cdot n$ bytes.

4 Experiments

Below follow experiments testing many substring index implementations on various types of text data and queries. The experiments were designed to emphasis the strengths and weaknesses of the various methods. Construction time, space usage and query performance was tested.

4.1 Test Data

Both "real world" and synthetic data was used in the tests. The largest tests from the Canterbury Corpus [cc] and the tests from Manzini and Ferragina's test collection [mf] were used, in addition to some artificial data generated by the author. The results for these tests are given in section 4.3.

To underline the properties of the methods, other synthetic data was also used: Space separated words from a Zipf distribution (Section 4.4), uniform random data with varying alphabet size (Section 4.5), and first order Markov data with a Zipf distribution on the transitions (Section 4.6).

4.2 Tested Implementations

The following implementations were tested:

- *STa* - The author's implementation of suffix trees using child arrays. See Section 2.4.
- *STs* - Using sibling lists. See Section 2.3.
- *Kur* - The suffix tree implementation by Kurtz [Kur99], taken from MUMmer 3.15[2]. It was compiled using the internal flag `STREE_HUGE`, which gives a maximum data size of 500 MB.
- *wotde* - The Write Only Top Town Eager[3] suffix tree construction algorithm [GKS03].
- *DS* - Deep-shallow suffix array construction algorithm[4] [MF04]. Default construction parameters used.
- *esa1*[*b*] - Enhanced suffix array. The implementation is a translation of the pseudocode from [AKO04] into C++. Node fields stored in separate arrays. Initial suffix array built with deep-shallow. Suffix b denotes that CLD values were stored in bytes overflowing into a hash map.
- *esa2*[*b*] - Enhanced suffix array stored as an array of structs.
- *BPR* - The *Bucket Pointer Refinement* suffix array construction algorithm[5] [SS05]. Default construction parameters used.
- *LZ* - LZ-index (*LZ*) [Nav04], implementation by Navarro and Arroyuelo[6].
- *CCSA* - Compressed compact suffix array [MN04a], implementation by Mäkinen and Gonzlez[7].
- *FM* - FM-index (*FM*) [FM05], implementation (version 2) by Ferragina and Venturini[8].
- *AFFM* - Alphabet-Friendly FM-index [FMMN04], implementation (version 2) by Gonzlez[9].
- *RLFM* - Run-Length FM-Index [MN04b], implementation by Mäkinen and Gonzlez[10].
- *SSA* - Succinct suffix array [MN05] (FM family), implementation (version 2) by Mäkinen and Gonzlez[11].

All programs were compiled with gcc 4.1.2 with `-O3` optimisation, and used glibc 2.3.6. They were run on an AMD Athlon 64 3500+ with 512 KB L2 cache, running Debian Sid with Linux 2.6.16. The kernel had the *perfctr* [per] patch, which makes hardware performance counters readable. The PAPI [pap] interface was used to read the variable `PAPI_L2_TCM`, giving level 2 cache misses. The times given are all wall-clock timings, excluding the time for reading files into memory. The memory usage given is `VmRSS` (resident set size) and `VmHWM` (resident set size peak) from `/proc/$pid/status`. (What is used by the tools `top` and `ps`). When running the tests, all initialisation of measurement variables and reading of data was done before each timing was started, and all related calculations were done after it was stopped.

[2] `http://mummer.sourceforge.net/`
[3] `http://bibiserv.techfak.uni-bielefeld.de/wotd/`
[4] `http://www.mfn.unipmn.it/~manzini/lightweight/`
[5] `http://bibiserv.techfak.uni-bielefeld.de/bpr/`
[6] `http://pizzachili.dcc.uchile.cl/indexes/LZ-index/`
[7] `http://pizzachili.dcc.uchile.cl/indexes/Compressed_Compact_Suffix_Array/`
[8] `http://pizzachili.dcc.uchile.cl/indexes/FM-indexV2/`
[9] `http://pizzachili.dcc.uchile.cl/indexes/Alphabet-Friendly_FM-Index/`
[10] `http://pizzachili.dcc.uchile.cl/indexes/Run_Length_FM_index/`
[11] `http://pizzachili.dcc.uchile.cl/indexes/Succinct_Suffix_Array/`

4.3 General Tests

The following tests include data from the Canterbury Corpus [cc] and Manzini and Ferragina's test collection [mf]. In addition, the 35th Fibonacci string is used. Note that the odd numbered[12] Fibonacci strings (counting from 1) give the maximum number of nodes in suffix trees $(2n + 1)$, and give the worst case in many non-linear suffix array construction algorithms [SS05]. The test `a025` is 2^{25} subsequent `a`'s. Table 2 gives some statistics for the tests: Length, alphabet size, LCPs, and first order empirical entropy [NM06].

Table 3 shows the construction speed for the tested methods, given as symbols indexed per second. The rationale for giving this instead of absolute time, is to give easier comparison between the tests shown in the tables, and clearer differentiation between the best methods in the plots that follow. An "x" in the tables denotes that that the test crashed, while "-" denotes that it took too long to complete. "N/A" denotes that the implementation was not designed to handle this alphabet size. Table 4 shows symbols indexed per L2 cache miss. It is expected that this is related to the construction performance.

The suffix trees with child arrays clearly perform better than the sibling lists variants, except on the small DNA tests, where they are slightly slower, and on the Fibonacci string, which has an alphabet of size 2. The advantage comes from the improved locality, as can be read from Table 4. *Kur* and *STs* have similar behaviour, with the former being slightly faster. This is probably because the latter was originally developed to index dynamic sets of documents. *Wotde* has a varying performance, which seems dependent of the average LCP (See Table 2). For tests larger than 5 MB, it is faster than the regular suffix trees only on the random data. The trees using sibling lists seem unsuited for random data with large alphabets. One might have expected that the time and cache performance for the suffix trees was directly proportional to the alphabet size, but it also depends on other properties of the data. A test where only the alphabet size is varied is given in Section 4.5.

The cost of building the enhanced suffix array is rather high. A reason for this is the way data is laid out in memory. The improvement of *esa2* over *esa1* does not show very well in this test on construction performance, as the data fields are built by separate algorithms. The tests in Section 4.7 show that *esa2* is often significantly faster on query lookup. In general, the regular suffix tree with child arrays has faster construction than both the enhanced suffix array and *wotde*.

DS and *BPR* are faster than the linear time suffix trees on most of these tests. The exception is those with very high LCP.

All the compressed structures except *LZ* use a suffix array built with deep-shallow as a starting point for their construction. The build times for the compressed representations seem very affordable. The *LZ* index has fast construction, and is even faster than all non-compressed methods on many of the tests. For all the compressed indexesthere is a strong relationship between time and cache performance. This can be seen when comparing for each method the results on the different tests in Tables 3 and 4.

Query performance is not shown in this test because it depends on too many parameters, such as the length of the query, the number of hits, and the data distribution. Query performance is evaluated in Section 4.7.

Table 5 shows the memory usage for the various implementations, in bytes per symbol after the construction was finished. The peak space usage is shown in table 6. The memory usage of the application seen externally is measured, which could give slightly inconsistent results because of space re-allocation. One method might use all its allocated memory, while one may have allocated more just before it finished.

STa uses 15-25% more memory than *STs*. Both methods use more memory on the DNA tests, probably because of a smaller average branching factor and many internal nodes. Apart from that all non-compressed methods have a rather constant space usage. Both *wotde* and *esa1/2* use slightly less space than the regular suffix trees. Among the construction algorithms for non-compressed structures, *BPR* is the only one using considerable extra space during construction. *DS* is much more space efficient, and has virtually no space overhead.

The *FM* index is a clear winner on space usage on most of the tests. The exception seems to be those with a large alphabet. The other indexes

[12] In the even numbered Fibonacci strings, `a` is always followed by `b`.

Table 2: Test statistics. The three first come from the Canterbury Corpus [cc], while the next 10 come from Manzini and Ferragina's test collection [mf]

		Length	σ	Med. LCP	Avg. LCP	Max. LCP	Entr. (1)	
1	bible.txt	4047393	63	11	14	551	3.27	King James bible
2	E.coli	4638690	4	11	17	2815	1.98	Escherichia coli
3	world192.txt	2473400	94	13	23	559	3.66	CIA world fact book
4	chr22.dna	34553758	5	13	1979	199999	1.88	Human chromo. 22
5	etext99	105277340	146	12	1109	286352	3.57	Project Gutenberg
6	gcc-3.0.tar	86630400	150	21	8603	856970	3.80	gcc 3.0 source files
7	howto	39422105	197	12	268	70720	3.92	Linux HOWTO text
8	jdk13c	69728899	113	113	679	37334	3.54	HTML and Java
9	linux-2.4.5.tar	116254720	256	17	479	136035	3.90	Linux Kernel 2.4.5
10	rctail96	114711151	93	61	282	26597	3.47	Reuters news in XML
11	rfc	116421901	120	19	93	3445	3.40	RFC text files
12	sprot34.dat	109617186	66	32	89	7373	3.93	Swiss Prot database
13	w3c2	104201579	256	114	42300	990053	4.06	HTML from w3c.org
14	fib035	9227465	2	2306866	2435423	5702885	0.59	35th Fibonacci String
15	rand254	104857600	254	3	2.86	6	7.98	Uniform, $\|\sigma\| = 254$
16	a025	33554432	1	16777217	16777216	33554431	0.00	`a` repeated 2^{25} times

Table 3: Symbols indexed per second in thousands.

	STa	STs	Kur	wotde	DS	DSesa	DSesa2	BPR	LZ	CCSA	FM	AFFM	RLFM	SSA
1	1958	1129	1471	1587	**3935**	1657	1832	3430	3201	1011	2327	630	2286	2127
2	1370	1046	1377	1256	3770	1531	1684	3374	**4504**	869	2206	783	2108	2251
3	2335	1263	1745	1857	**4833**	1714	2056	3241	3211	1129	2574	522	2622	2310
4	1212	932	1165	22	2900	1106	1327	2250	**3753**	730	1602	669	1825	1888
5	1332	632	700	127	1736	854	984	1292	**1873**	606	1343	341	1240	1200
6	2227	1044	1273	12	1300	838	923	1813	**2457**	630	1744	350	1122	1036
7	1677	707	791	468	**2763**	1194	1405	2128	2110	796	1795	332	1781	1659
8	3321	2282	**3598**	178	1251	802	884	1341	3553	579	1366	387	965	879
9	2049	883	1032	258	**2520**	1181	1363	2086	N/A	N/A	N/A	N/A	N/A	N/A
10	2302	1198	1447	287	1091	687	756	1039	**2708**	519	1172	371	855	801
11	1842	864	1005	533	**2224**	1038	1202	1463	2212	712	1674	410	1550	1433
12	2029	778	925	499	1904	927	1070	1261	**2445**	658	1523	473	1356	1267
13	**3052**	1509	2138	-	1178	763	857	1484	N/A	N/A	N/A	N/A	N/A	N/A
14	4586	5346	12375	-	91	88	89	267	**21400**	73	491	73	76	76
15	606	30	32	1957	**2470**	925	1101	1218	647	592	x	191	1158	1252
16	4547	5931	14767	-	41708	8393	9494	22805	**55966**	3290	-	1012	9558	10168

Table 4: Symbols indexed per L2 cache miss.

	STa	STs	Kur	wotde	DS	DSesa	DSesa2	BPR	LZ	CCSA	FM	AFFM	RLFM	SSA
1	0.30	0.15	0.13	0.18	**0.48**	0.19	0.22	0.34	0.44	0.15	0.34	0.13	0.34	0.34
2	0.16	0.13	0.13	0.14	0.43	0.17	0.19	0.37	**0.61**	0.15	0.30	0.11	0.31	0.31
3	0.40	0.18	0.16	0.27	**0.67**	0.23	0.27	0.40	0.47	0.18	0.44	0.14	0.44	0.44
4	0.18	0.14	0.15	0.012	0.29	0.13	0.16	0.32	**0.47**	0.11	0.22	0.096	0.23	0.23
5	0.22	0.092	0.088	0.057	0.18	0.11	0.12	0.16	**0.24**	0.088	0.16	0.082	0.15	0.15
6	**0.49**	0.16	0.15	0.00087	0.19	0.12	0.14	0.15	0.34	0.10	0.23	0.11	0.19	0.19
7	0.28	0.091	0.082	0.11	0.29	0.14	0.17	0.24	**0.29**	0.11	0.23	0.11	0.22	0.22
8	**1.2**	0.56	0.54	0.031	0.17	0.11	0.12	0.093	0.43	0.085	0.17	0.090	0.14	0.14
9	**0.44**	0.14	0.13	0.068	0.28	0.15	0.18	0.19	N/A	N/A	N/A	N/A	N/A	N/A
10	**0.50**	0.20	0.19	0.023	0.11	0.080	0.089	0.069	0.33	0.067	0.14	0.067	0.10	0.10
11	**0.36**	0.13	0.13	0.068	0.24	0.13	0.15	0.13	0.30	0.10	0.21	0.10	0.19	0.19
12	**0.39**	0.11	0.11	0.052	0.20	0.12	0.14	0.11	0.31	0.093	0.19	0.093	0.17	0.17
13	**0.97**	0.26	0.26	-	0.16	0.11	0.12	0.12	N/A	N/A	N/A	N/A	N/A	N/A
14	11	10	**18**	-	0.015	0.014	0.014	0.0095	4.1	0.013	0.11	0.013	0.014	0.014
15	0.099	0.0031	0.0033	0.18	**0.34**	0.13	0.16	0.16	0.12	0.12	x	0.089	0.25	0.25
16	23	28	49	-	130	25	17	55	**419**	20	-	4.1	32	32

Table 5: Space usage in bytes per symbol indexed after construction.

	STa	STs	Kur	wotde	DS	DSesa	DSesa2	BPR	LZ	CCSA	FM	AFFM	RLFM	SSA
1	16	13	13	11	5.5	11	11	5.4	6.2	1.8	**0.85**	2.4	1.4	1.4
2	20	17	16	11	5.4	11	11	5.3	4.4	3.0	**0.78**	1.0	1.3	1.0
3	16	13	13	11	5.8	11	11	5.6	7.4	1.9	**1.2**	1.8	1.7	1.9
4	19	16	16	11	5.1	11	11	5.0	2.4	2.5	**0.42**	0.66	0.92	0.69
5	16	13	13	10	5.0	10	10	5.0	2.8	1.9	**0.47**	0.83	0.98	1.0
6	16	13	13	11	5.0	10	10	5.0	2.5	1.1	**0.39**	0.96	0.85	1.1
7	16	13	13	11	5.0	10	10	5.0	2.9	1.6	**0.53**	2.0	0.98	1.1
8	16	13	13	11	5.0	10	10	5.0	1.9	0.70	**0.24**	0.76	0.73	1.2
9	16	13	13	11	5.0	10	10	**5.0**	N/A	N/A	N/A	N/A	N/A	N/A
10	15	12	12	10	5.0	10	10	5.0	2.0	0.98	**0.28**	0.73	0.79	1.1
11	16	13	13	11	5.0	10	10	5.0	2.5	1.2	**0.39**	0.84	0.85	1.0
12	15	13	13	10	5.0	10	10	5.0	2.5	1.4	**0.37**	0.90	0.88	1.1
13	16	13	14	-	5.0	10	10	**5.0**	N/A	N/A	N/A	N/A	N/A	N/A
14	19	16	17	-	5.2	10	10	5.2	1.3	0.64	**0.28**	0.49	0.85	0.69
15	13	9.5	8.4	6.4	5.0	10	10	5.0	5.8	4.0	x	2.5	2.0	**1.5**
16	19	16	17	-	5.1	10	10	5.0	1.1	0.50	-	3.5	0.69	**0.49**

Table 6: Peak space usage during construction.

	STa	STs	Kur	wotde	DS	DSesa	DSesa2	BPR	LZ	CCSA	FM	AFFM	RLFM	SSA
1	16	13	13	11	**5.5**	11	11	11	6.7	8.3	6.6	10	6.7	6.7
2	20	17	16	11	5.4	11	11	10	**4.9**	8.4	6.5	11	6.7	6.7
3	16	13	13	11	**5.8**	11	11	12	7.9	8.6	6.8	9.1	7.0	7.0
4	19	16	16	12	5.1	11	11	10	**4.1**	8.4	6.2	9.8	6.3	6.3
5	16	13	13	10	**5.0**	10	10	10	5.6	8.5	6.1	9.0	6.3	6.3
6	16	13	13	11	**5.1**	10	10	10	5.8	8.5	6.1	8.2	6.3	6.3
7	16	13	13	11	**5.1**	10	10	11	6.6	8.4	6.2	9.7	6.3	6.3
8	16	13	13	11	5.0	10	10	10	**4.2**	8.5	6.1	7.7	6.3	6.3
9	16	13	13	11	**5.0**	10	10	11	N/A	N/A	N/A	N/A	N/A	N/A
10	15	12	12	10	5.0	10	10	10	**4.7**	8.5	6.1	8.0	6.3	6.3
11	16	13	13	11	**5.0**	10	10	10	5.4	8.5	6.1	8.3	6.3	6.3
12	15	13	13	10	**5.0**	10	10	10	5.7	8.5	6.1	8.4	6.3	6.3
13	16	13	14	-	**5.0**	10	10	11	N/A	N/A	N/A	N/A	N/A	N/A
14	19	16	17	-	5.2	11	11	10	**1.4**	8.3	6.3	7.6	6.5	6.5
15	13	9.5	8.4	7.7	**5.0**	10	10	11	20	9.3	x	12	6.3	6.3
16	19	16	17	-	5.1	10	10	10	**1.1**	8.4	-	59	6.3	6.3

from the FM family, *AFFM*, *RLFM* and *SSA*, fare much better there. **FIXME:** [*Fix bug in* FM *for large* σ.] The *LZ* index uses more space than a suffix array on some of the smaller tests, but around half the space on the larger tests.

The working space used during construction varies for the compressed indexes, as seen in table 6. *CCSA* and *AFFM* need around 8-9 bytes per symbol, while the rest of the FM family needs a little more than 6. The working space for the LZ index strongly depends on the properties of the text data, as the tree data structures built utilise its repetitions.

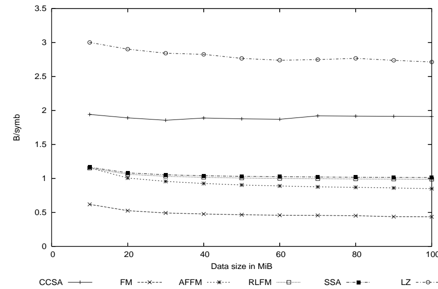


Figure 2: Space usage for compressed structures after construction, Zipf word data, increasing size.

4.4 Test on Increasing Data Size

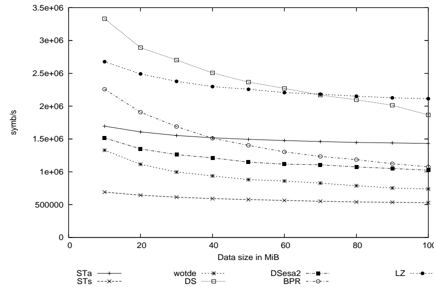
Figure 1 gives the results of indexing increasing amounts of space separated word data from a Zipf distribution (parameter $s = 1.0$). The figure shows symbols indexed per second and per L2 cache miss. Notice that these two quantities are closely related, both here and in the following tests.

The suffix tree *STa* is almost three times faster than *STs*, even though they logically perform the same steps. This is because of a better locality, which can be read from the plot for cache performance. *STa* traverses a contiguous array to lookup the child of a node, while *STs* traverses sibling nodes with "random" memory locations. *DS* and *BPR* here exhibit the most non-linear performance, suggesting than for larger data, linear time suffix array constructions would be faster. *Wotde*, which is also a non-linear method, is faster than *STs*, but slower than *STa*.

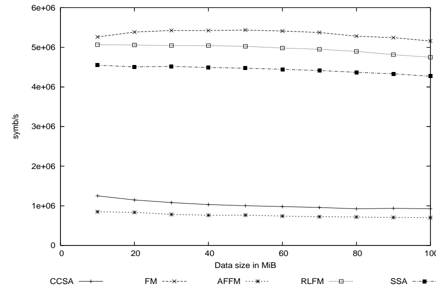
One would expect that the reason *STa* and *STs* show a slightly non-linear behaviour, is the fact that a smaller relative proportion of the tree fits in cache. In the construction algorithms the tree is traversed in a seemingly random pattern along a certain depth. The average depth is the expected longest common prefix of closest pairs between the suffixes added so far. For random data, this is $\log_\sigma n$ [Dev82]. But the cache performance is more linear than the time performance both for *STa* and *STs*. The additional slowdown might be due to memory management overhead.

The *LZ* index shows great indexing performance. It has a more linear behaviour than *DS*, and is faster when the data size reaches 100 MB. For all the other compressed implementations, the time and cache misses for the initial construction of the suffix array by deep-shallow has been subtracted. This is done because this construction in many cases dominates the running time, and made it very hard to interpret the cost of building the compressed representations themselves in Sections 4.5 and 4.6. For *FM*, *RLFM* and *SSA*, the cost of the compression is less than the cost of the initial build. An interesting feature in the plot for cache performance is that *FM*, *RLFM*, and *SSA* are nearly identical. This must be because they have similar access patterns to their data structures. Figure 2 shows the space usage for the compressed methods on this test. *FM* is twice as space efficient as the next method for 100 MB.

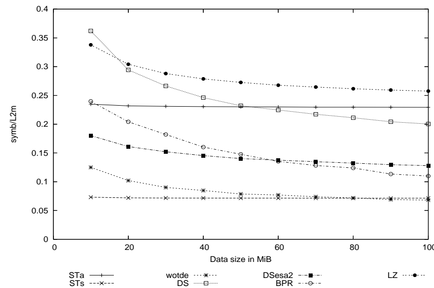
Figure 3 shows a similar test, but with uniform random data ($\sigma = 20$). *DS* and *esa1/2* show approximately the same performance as in the last test, but the other methods do not. Note that *BPR* here seems has less decrease than *DS*, and may have been the fastest method with even more data. *Wotde* is now much faster, while the other suffix trees are slower. Random data gives lower average LCP, which benefits *wotde*, but also a higher average out degree in the nodes, which is bad for the regular suffix trees. The worst effect is seen for the sibling lists, where the performance is halved from what was seen for Zipf data in Figure 1. All compressed structures perform slightly worse on the random data. This is probably because there are fewer regularities in the text, and the index structures grow larger.



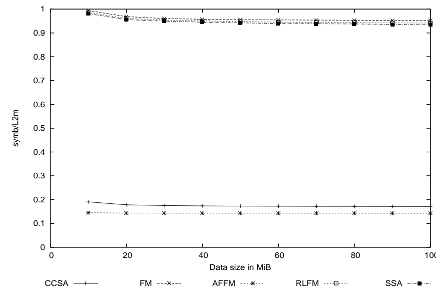
(a) Symbols indexed per second.



(b) Symbols indexed per second.

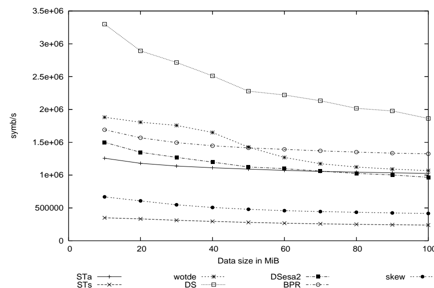


(c) Symbols indexed per L2 miss.

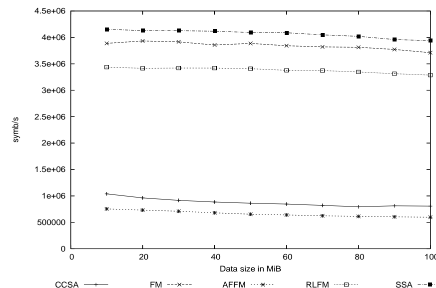


(d) Symbols indexed per L2 miss.

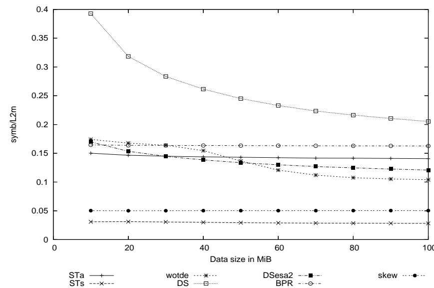
Figure 1: Indexing performance on increasing data size, Zipf word data. Construction of initial suffix array not included for compressed indexes.



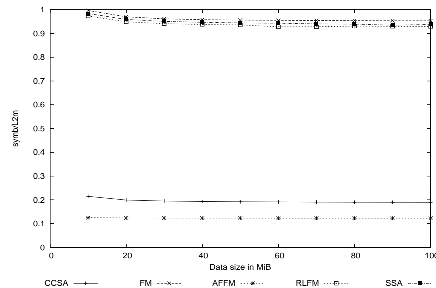
(a) Symbols indexed per second.



(b) Symbols indexed per second.



(c) Symbols indexed per L2 miss.



(d) Symbols indexed per L2 miss.

Figure 3: Indexing performance on increasing data size, random data ($\sigma = 20$). Construction of initial suffix array not included for compressed indexes.

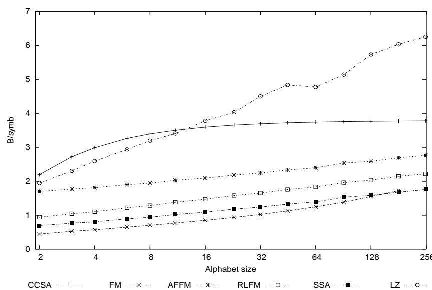


Figure 5: Space usage for compressed structures after construction, on increasing alphabet size.

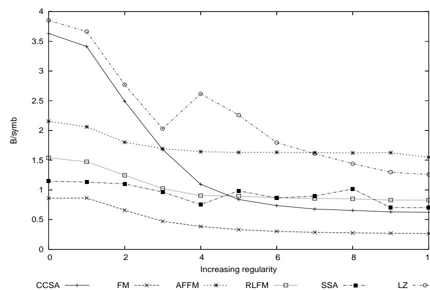


Figure 7: Space usage for compressed structures after construction on Markov data. ($\sigma = 20$)

4.5 Increasing Alphabet Size

Figure 4 shows performance when indexing 10 MB of uniformly random data with an increasing alphabet size.

DS here clearly out-performs the other methods for large alphabets, but has a rather peculiar behaviour. This suffix array construction algorithm combines two different sorting techniques, where one utilises the results from the other. The behaviour seen may be due to the individual performance of these, and the way they are combined. *Wotde* shows great performance as the alphabet increases, because of the decreasing average LCP. The opposite behaviour is seen for the other suffix trees, where an increasing number of children for internal nodes slows down the construction. For *STs* the performance decreases with a factor 30 as the alphabet size goes from 2 to 254. The same effect is seen to a lesser degree for *STa*, with a drop factor of 2.

The construction times of all compressed indexes except *CCSA* seem strongly dependant on the alphabet size, but the amount of cache misses is not. Seemingly, a larger alphabet results in more computation, but as the data accesses are already rather random, there are not more cache misses, even though the data structures are larger. Figure 5 shows the space usage in bytes per symbol. The *FM* index is extremely efficient for small alphabets, and is best of all methods up to an alphabet size of around 128. Remember that this very artificial test gives the worst case space usage for the compressed indexes, as the entropy of the data is maximal for the given alphabet size. The space usage of around 2 bytes per symbol with an alphabet size of 254 for the most space efficient structures is impressive.

4.6 Markov Data

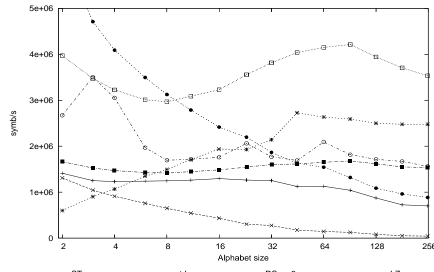
Many of the methods tested have time and space efficiencies which are dependant on the randomness of the data. Less random data gives higher average LCP, degrading the performance of some of the non-linear construction methods. Figure 6 shows results for indexing first order Markov data with a Zipfian transition distribution with varying parameter s. An alphabet size of 20 was used. In a general Zipf distribution, the probability of selecting element k is given as

$$p(k; s, n) = \frac{k^{-s}}{\sum_{i=1}^{n} i^{-s}}$$

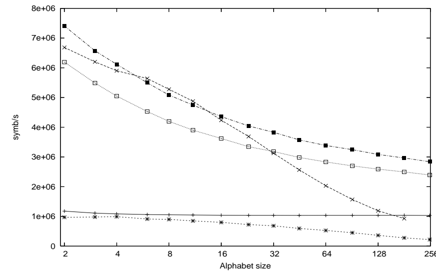
Setting $s = 0$ gives a uniform distribution. For the data used, the average LCP approximately doubled each time the parameter s increased by 1, from 4.7 for $s = 0$, to 2160 for $s = 10$.

Wotde has the greatest dependence on the LCP, and is the only method showing a contiguous drop in performance here. *DS* has a strong peak at $s = 4$, and a total breakdown at $s = 7$, where the average LCP is 324. The peak probably comes from the distribution of work between the two methods deep-shallow combines, as was discussed in section 4.5. The performances of the regular suffix trees increase as the data grows less random, due to lower average branching factors.

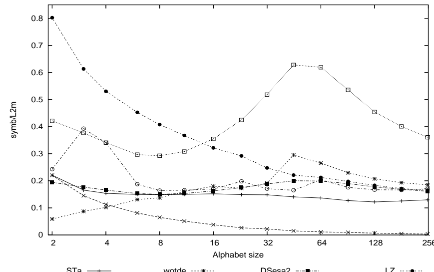
LZ shows an extreme performance on this data, which is very repetitive for large s. The curve for *LZ* continues rising to 35 million symbols per second at $s = 10$, with 70 symbols indexed per L2 miss. *FM*, *RLFM* and *SSA* also show a significant



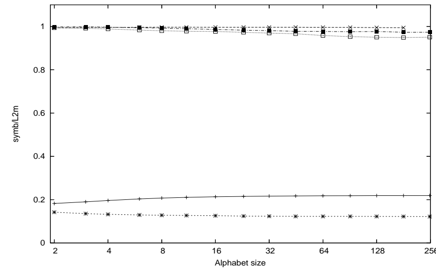
(a) Symbols indexed per second.



(b) Symbols indexed per second.

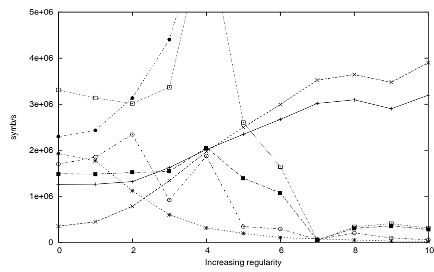


(c) Symbols indexed per L2 miss.

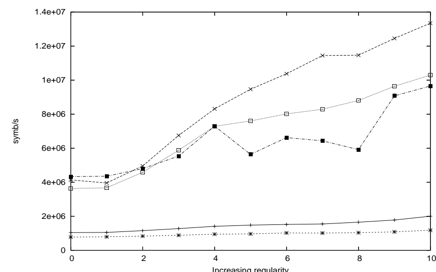


(d) Symbols indexed per L2 miss.

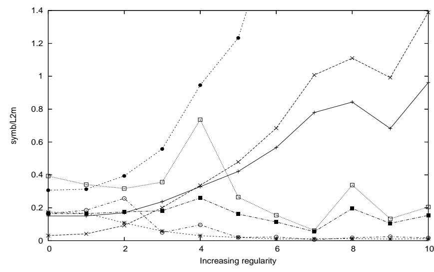
Figure 4: Indexing performance on increasing alphabet size, uniform data. Construction of initial suffix array not included for compressed indexes.



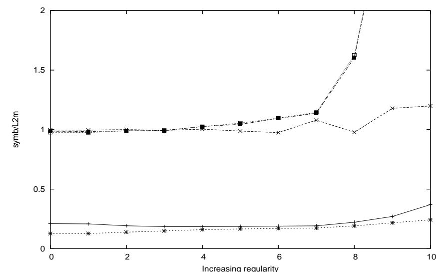
(a) Symbols indexed per second.



(b) Symbols indexed per second.



(c) Symbols indexed per L2 miss.



(d) Symbols indexed per L2 miss.

Figure 6: Indexing performance on first order Markov data with Zipfian transition distribution and increasing parameter s. ($\sigma = 20$). Construction of initial suffix array not included for compressed indexes.

increase in performance with the regularity. Note that for these methods the bulk of the time used is the initial deep-shallow construction, which is subtracted here. To get more robust performance, a linear time suffix array construction algorithm could be used.

Figure 7 shows the space efficiency for the compressed indexes. As in the previous tests, the *FM* index is the most space efficient. *CCSA* and *LZ* also show very good trends. Although the space usage for *LZ* is low, it does not match the extreme time performance that was shown in Figure 6(a). Various tradeoffs between space usage and query performance could be made in the implementation. All the compressed methods have a dependency on higher order entropy in their asymptotic space usage [NM06], but the implementations show this to a varying extent here.

For large parameters s, the data has LCPs higher than what you would see in any real world data of reasonable size. In these pure Markov chain strings, the LCP between almost *all* neighbouring suffixes is very high. The average LCP here should probably be compared with the median LCP seen in table 2. At $s = 6$, the average LCP is 172, and the median LCP 159.

4.7 Query Lookup

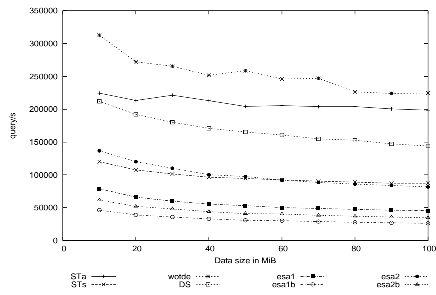
As previously mentioned, many parameters influence query performance. One such parameter is the total data size. Figure 8 shows the number of queries per second and per L2 cache miss, on an increasing amount of Zipf word data ($s = 1, \sigma = 20$). Queries of length 30 were run, and only one hit was reported (to isolate lookup time). The performance drop for the suffix trees is due to the increasing number of nodes seen in downward traversal in the tree, and the number of children considered in child lookup. The latter hits *STs* worse than *STa*. The decrease for the suffix array is due to the increasing number of jumps in the binary search. *Wotde* has the best performance in this test, followed by *STa*.

Esa1 performs rather poorly, and even worse than *STs*, which logically performs the same steps. This is related to the number of cache misses, which is many times as high. A sibling traversal is also performed in *esa1*, but this has bad locality for two reasons: The first is an implementational detail. The information for one "position" in the enhanced suffix array is spread over different arrays, giving unnecessary cache misses. This is implemented differently in *esa2*, which has performance similar to *STs*. The second reason is an inherently bad locality in the enhanced suffix array. For an internal "node" close to the root, the values read to traverse the child nodes is spread over a large area. This is the reason *esa2* has much slower lookup than *STa*, and also a more degrading performance as the amount of data increases. . The variants *esa1b* and *esa2b* have the CLD field stored in bytes overflowing into a hash table (see Section 3). Even though the total overflow is neglible in terms of space, the query lookup performance is rougly halved, because many nodes in the upper part of the tree overflow.

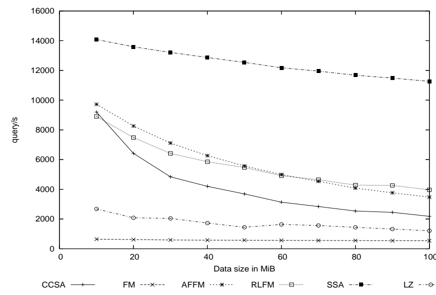
The query lookup performance on the compressed indexes differs greatly. *SSA* is almost 20 times faster than *FM* on 100 MB of data, but 20 times slower than the fastest suffix tree. *FM* has poor time performance, but better cache performance than the other methods. The author does not know the implementation well enough to comment this properly. Note the different scales on the y-axis for compressed indexes. Searching in the compressed structures involves many recursive lookups for each logical step in the search. The values to be read are spread throughout the data structures, giving bad locality.

Figure 9 shows query lookup on data with varying alphabet size. Queries of length 30 were issued on 10 MB of uniform random data. Lookup performance degrades in the suffix tree variants and the enhanced suffix array when the alphabet size grows larger. This is because these methods traverse lists of children in trees, and the average lengths of these lists increase with the alphabet. The effect hits *STs*, *esa1* and *esa2* worst, as they have poor locality in the child traversal.

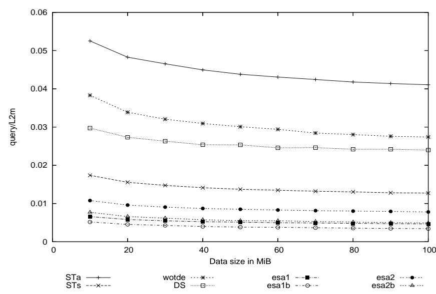
The performances of many of the compressed indexes are very dependant of the alphabet size. Only *CCSA* and *FM* have alphabet independent asymptotic lookup times [NM06]. *CCSA* shows an increase in performance as the alphabet size increases. This is because the average length of the match between the search pattern and the suffixes considered in the binary search decreases, resulting in fewer character comparisons. The implementation tested does not use the trick of keeping track of how many symbols match on the left and right border of the binary search, to reduce the expected



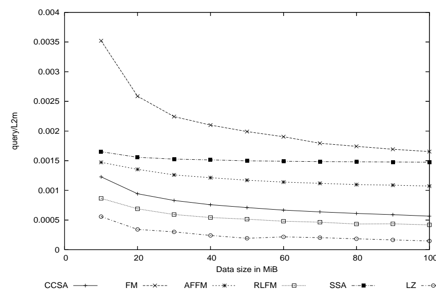
(a) Queries per second.



(b) Queries per second.

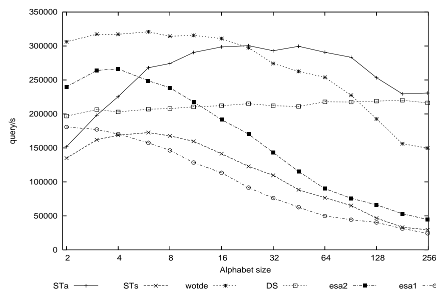


(c) Queries per L2 miss.

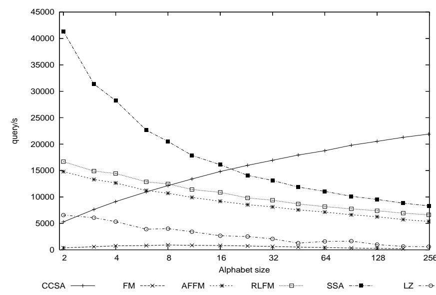


(d) Queries per L2 miss.

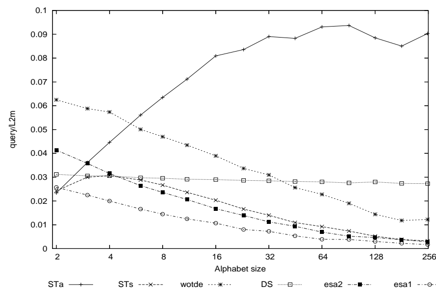
Figure 8: Query lookup on increasing data size, Zipf word data.



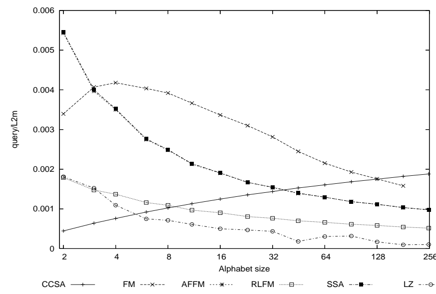
(a) Queries per second.



(b) Queries per second.

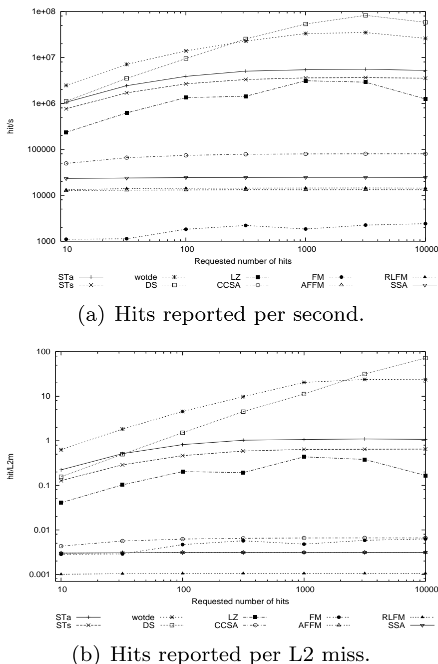


(c) Queries per L2 miss.



(d) Queries per L2 miss.

Figure 9: Query lookup on increasing alphabet size, uniform data.



(a) Hits reported per second.

(b) Hits reported per L2 miss.

Figure 10: Requesting an increasing number of hits. $\log_{10}$ scale on both axis.

number of character comparisons. The methods *AFFM*, *RLFM* and *SSA* all have the same asymptotic lookup cost [NM06], but *SSA* is faster in practise, and has the best query performance of all compressed indexes for small alphabets.

4.8 Reporting Hits

In the previous tests on query performance a single hit was reported for each query, but for some applications the efficiency when listing large numbers of occurrences is more relevant. Figure 10 shows the performance when reporting an increasing number of hits from 100 MB of uniform data with an alphabet size of 20. All search functions were modified to cap the number of hits, and a varying number was requested. The lengths of the queries were set such that the number of hits would be at least what was requested.

Although suffix trees are asymptotically optimal here, both for sibling lists and child arrays, the suffix arrays have an advantage in practise. After finding the left and right border for a match, values are read between these subsequently, giving great spatial locality and performance. *DS* is more than 15 times faster than *STa* at the most. The enhanced suffix array is not included, as it has a performance almost identical to the regular suffix array. *Wotde* is significantly faster than the regular suffix trees in this test, due to a more compact representation and better locality. The slight drop in performance for more than 3000 hits seen for the fastest methods is due to the overhead for the dynamic container used to hold the hits. This could be easily avoided for the suffix arrays, as the number of hits is known after finding the left and right border.

The *LZ* index shows the best performance among the compressed indexes, and delivers hits nearly as fast as a suffix tree with sibling lists. The other compressed structures are 3-5 orders of magnitude slower than the suffix array, but in many applications, thousands of hits per second is sufficient.

Table 7: Tree operations. Showing time in seconds for construction, bottom–up and top–down traverse, and longest common substring search.

(a) 50 MB DNA data.

	ST arr	ST sibl	DS ESA	DS ESA2	CST
Build	**50**	59	82	71	2604
Top–down	14	16	**2.0**	2.5	16
Bottom–up	9.7	9.3	**9.2**	9.3	23
LCSS	24	33	11	**4.0**	313

(b) 50 MB protein data.

	ST arr	ST sibl	DS ESA	DS ESA2	CST
Build	**42**	137	81	70	4788
Top–down	11	14	**1.8**	2.4	17
Bottom–up	6.4	**6.3**	6.7	6.7	24
LCSS	22	76	33	**6.8**	721

4.9 Tree Operations

Suffix trees can be used for many purposes other than substring search. They are used in bioinformatics to find many properties of strings. Gusfield [Gus97] lists numerous applications. Because suffix trees are costly in space, Abouelhoda et al. [AKO04] show how to replace suffix trees with enhanced suffix arrays in all algorithms doing bottom-up, top-down or suffix link traversal in the tree. Sadakane [Sad07] has taken this one step further and shown how to use a succinct representation of a suffix tree on top of any compressed suffix array, supporting the same operations.

Table 7 shows the performance of suffix trees,

enhanced suffix arrays and a compressed suffix tree (*CST*) on various tree operations. The compressed suffix tree implementation by Mäkinen [VGDM07] was used. The longest common sub-string (LCSS) between two strings is found by building a tree for the first string, and then traversing it matching suffixes of the other string, following parent-to-child and suffix links in the tree. (The construction time for the index is excluded.) On LCSS, *esa2* is much faster than *esa1*, because many fields are read in each "node", and these are stored close to each other in *esa2*. In general, *esa2* is as fast or faster than the regular suffix trees on tree traversal.

The compressed suffix tree is very competitive on top–down and bottom–up traversal, where the operations on nodes performed in constant time, but around 100 times slower than *esa2* on LCSS, where they are not [Sad07].

5 Conclusion

It has been shown that performance is strongly dependant of locality also in data structures resident in primary memory. This should be considered when implementing indexes, as can be seen when comparing the performance of *STa* with *STs*, and *esa2* with *esa1*. Which index structures should be chosen for which tasks depends on what is more important of space usage, construction time, query lookup time and reporting hits.

- Suffix trees with dynamic arrays can be as much as 20 times faster on construction than sibling lists for byte sized alphabets, as seen in figure 4(a), and 10 times faster on query lookup, as seen in figure 9. The array representation requires about 20% more space.
- In applications where large numbers of hits must be reported, suffix arrays are strongly preferable over suffix trees, as they list hits 1-2 orders of magnitude faster. See Figure 10(a).
- For fast lookup for small numbers of hits, a suffix tree variant is the most effective index, if you have enough memory. See Figures 8(a) and 9(a).
- The lookup performance of the enhanced suffix array is dependant of the alphabet size, and a regular suffix array may be faster if the alphabet is sufficiently large. See figure 9(a).
- The deep-shallow suffix array construction should in general be chosen over BPR, as it has a much lower space overhead. See Table 6.
- Among the implementations tested here, the FM index is the most space efficient, as long as the alphabet size is not too large. See Table 5.
- The LZ-index is fast on construction and listing hits. As it is more space efficient than a suffix array, it would be the structure of choice in many situations.
- The *wotdeager* suffix tree is fast on lookup and reporting hits, but the construction easily breaks down for non-random data, as seen for some of the benchmarks in table 3.
- Tree traversal algorithms are faster with enhanced suffix arrays than suffix trees. See Table 7.

In general, when designing the layout of data structures, it is important to consider the access patterns in construction and search to maximise locality. A few more computational steps during lookup will often be cheaper than a cache miss.

Acknowledgements.

The author would like to thank all the authors who have made their code publically available, and Øystein Torbjørnsen, Magnus Lie Hetland and Tor Egge for helpful feedback on this article,

References

[AKO04] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms*, 2(1):53–86, 2004.

[BH04] S. J. Bedathur and J. R. Haritsa. Engineering a fast online persistent suffix tree construction. In *Proc. ICDE*, page 720, 2004.

[cc] The canterbury corpus. `http://corpus.canterbury.ac.nz/`.

[Dev82] L. Devroye. Note on the average depth of tries. *Computing*, 28:367–371, 1982.

[Far97] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. FOCS*, pages 137–143, 1997.

[FCFM00] M. Farach-Colton, P. Ferragina, and S. M. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

[FKS84] M.L. Fredman, J. Komlós, and E. Szemerédi. Storing a Sparse Table with 0 (1) Worst Case Access Time. *J. ACM*, 31(3):538–544, 1984.

[FM00] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, page 390, 2000.

[FM05] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[FMMN04] P. Ferragina, G. Manzini, V. Makinen, and G. Navarro. An alphabet-friendly FM-index. *Proc. SPIRE*, pages 150–160, 2004.

[GGV03] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.

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

[GMN04] S. Grabowski, V. Makinen, and G. Navarro. First Huffman, then Burrows-Wheeler: An alphabet-independent FM-index. *Proc SPIRE*, 2004.

[Gri05] N. Grimsmo. Dynamic indexes vs. static hierarchies for substring search. Master's thesis, Norwegian University of Science and Technology, 2005.

[Gus97] D. Gusfield. *Algorithms on strings, trees, and sequences: Computer science and computational biology.* 1997.

[GV00] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proc. STOC*, pages 397–406, 2000.

[Int06] Intel Corporation. *IA-32 Intel Architecture Optimization Reference Manual*, 2006.

[KA03] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. CPM*, 2003.

[KS03] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. ICALP*, 2003.

[KSPP03] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. CPM*, pages 186–199, 2003.

[KU96] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. WSP*, pages 141–155, 1996.

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

[Mäk00] V. Mäkinen. Compact suffix array. In *CPM*, pages 305–319, 2000.

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

[mf] Manzini and Ferragina's test collection. `http://www.mfn.unipmn.it/~manzini/lightweight/`.

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

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

[MN04a] V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *CPM*, pages 420–433, 2004.

[MN04b] V. Mäkinen and G. Navarro. Run-length FM-index. In *Proc. DIMACS*, pages 17–19, 2004.

[MN05] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. In *Proc. CPM*, 2005.

[Mut02] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. SODA*, pages 657–666, 2002.

[Nav03] G. Navarro. The LZ-index: A text index based on the Ziv-Lempel trie. Technical Report TR/DCC-2003-1, Dept. of Computer Science, Univ. of Chile, 2003.

[Nav04] G. Navarro. Indexing text using the ziv-lempel trie. *J. of Discrete Algorithms*, 2(1):87–114, 2004.

[NM06] G. Navarro and V. Mkinen. Compressed full-text indexes. Technical Report TR/DCC-2006-6, Dept. of Comp. Sci., U. of Chile, 2006.

[OvL80] M. H. Overmars and J. van Leeuwen. Some principles for dynamizing decomposable searching problems. Technical Report RUU-CS-80-1, Rijksuniversitet Utrecht, 1980.

[pap] Performance Application Programming Interface. `http://icl.cs.utk.edu/papi/`.

[per] Perfctr Hardware performance counters. `http://user.it.uu.se/~mikpe/linux/perfctr/`.

[Sad03] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. of Algorithms*, 48(2):294–313, 2003.

[Sad07] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems (Online)*, 2007.

[SS05] K. Schürmann and J. Stoye. An incomplex algorithm for fast suffix array construction. In *Proc. ALENEX*, 2005.

[TTHP05] Y. Tian, S. Tata, A. Hankins, and M. Patel. Practical methods for constructing suffix trees. *VLDB J.*, 14(3):281–299, 2005.

[Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(5):249–260, 1995.

[VGDM07] N. Vlimki, W. Gerlach, K. Dixit, and V. Mkinen. Engineering a compressed suffix tree implementation. Accepted to WEA, 2007.

[Wei73] P. Weiner. Linear pattern matching algorithms. In *IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.