The VLDB Journal (2005) 14(3): 281–299
DOI 10.1007/s00778-005-0154-8

REGULAR PAPER

Yuanyuan Tian · Sandeep Tata ·
Richard A. Hankins · Jignesh M. Patel

Practical methods for constructing suffix trees

Received: 14 October 2004 / Accepted: 1 July 2005 / Published online: 26 September 2005

Abstract Sequence datasets are ubiquitous in modern life-science applications, and querying sequences is a common and critical operation in many of these applications. The suffix tree is a versatile data structure that can be used to evaluate a wide variety of queries on sequence datasets, including evaluating exact and approximate string matches, and finding repeat patterns. However, methods for constructing suffix trees are often very time-consuming, especially for suffix trees that are large and do not fit in the available main memory. Even when the suffix tree fits in memory, it turns out that the processor cache behavior of theoretically optimal suffix tree construction methods is poor, resulting in poor performance. Currently, there are a large number of algorithms for constructing suffix trees, but the practical trade-offs in using these algorithms for different scenarios are not well characterized.

In this paper, we explore suffix tree construction algorithms over a wide spectrum of data sources and sizes. First, we show that on modern processors, a cache-efficient algorithm with $O(n^2)$ worst-case complexity outperforms popular linear time algorithms like Ukkonen and McCreight, even for in-memory construction. For larger datasets, the disk I/O requirement quickly becomes the bottleneck in each algorithm's performance. To address this problem, we describe two approaches. First, we present a buffer management strategy for the $O(n^2)$ algorithm. The resulting new algorithm, which we call "Top Down Disk-based" (TDD), scales to sizes much larger than have been previously described in literature. This approach far outperforms the best known disk-based construction methods. Second, we present a new disk-based suffix tree construction algorithm that is based on a sort-merge paradigm, and show that for constructing very large suffix trees with very little resources, this algorithm is more efficient than TDD.

Keywords Suffix tree construction · sequence matching

Y. Tian · S. Tata · J. M. Patel
University of Michigan, 1301 Beal Avenue, Ann Arbor, MI 48109-2122, USA
E-mail: {ytian, tatas, jignesh}@eecs.umich.edu

R. A. Hankins (✉)
Microarchitecture Research Lab, Intel Corp., 2200 Mission College Blvd, Santa Clara, CA 95054, USA
E-mail: richard.a.hankins@intel.com

1 Introduction

Querying large string datasets is becoming increasingly important in a number of life-science and text applications. Life-science researchers are often interested in explorative querying of large biological sequence databases, such as genomes and large sets of protein sequences. Many of these biological datasets are growing at exponential rates– for example, the sizes of the sequence datasets in GenBank have been doubling every 16 months [50]. Consequently, methods for *efficiently* querying large string datasets are critical to the success of these emerging database applications.

A suffix tree is a versatile data structure that can help execute such queries efficiently. In fact, suffix trees are useful for evaluating a wide variety of queries on string databases [26]. For instance, the exact substring matching problem can be solved in time proportional to the length of the query, once the suffix tree is built on the database string. Suffix trees can also be used to solve approximate string matching problems efficiently. Some bioinformatics applications such as MUMmer [16, 17, 37], REPuter [36], and OASIS [41] exploit suffix trees to efficiently evaluate queries on biological sequence datasets. However, suffix trees are not widely used because of their high cost of construction. As we show in this paper, building a suffix tree on moderately sized datasets, such as a single chromosome of the human genome, takes over 1.5 hours with the best-known existing disk-based construction technique [28]. In contrast, the techniques that we develop in this paper reduce the construction time by a factor of 5 on inputs of the same size.

Even though suffix trees are currently not in widespread use, there is a rich history of algorithms for constructing suffix trees. A large focus of previous research has been on linear-time suffix tree construction algorithms [40, 52, 54]. These algorithms are well suited for small input strings where the tree can be constructed entirely in main memory. The growing size of input datasets, however, requires that we construct suffix trees efficiently on disk. The algorithms proposed in [40, 52, 54] cannot be used for disk-based construction as they have poor locality of reference. This poor locality causes a large amount of random disk I/O once the data structures no longer fit in main memory. If we naively use these main-memory algorithms for on-disk suffix tree construction, the process may take well over a day for a single human chromosome.

The large and rapidly growing size of many string datasets underscores the need for fast *disk-based* suffix tree construction algorithms. Theoretical methods for optimal external memory suffix tree construction have also been developed [22], however, the practical behavior of these algorithms has not been explored. A number of recent research investigations have also examined practical suffix tree construction techniques for large datasets [7, 28]. However, these approaches do not scale well for large datasets (such as an entire eukaryotic genome).

In this paper, we present new approaches for *efficiently* constructing large suffix trees on disk. We use a philosophy similar to the one in [28]. We forgo the use of suffix links in return for a much better memory reference pattern, which translates to better scalability and performance for constructing large suffix trees.

The main contributions of this paper are as follows:

1. We present the "Top Down Disk-based" (TDD) approach, which was first introduced in [49]. TDD can be used to efficiently build suffix trees for a wide range of sizes and input types. This technique includes a suffix tree construction algorithm called Partition and Write Only Top Down (PWOTD), and a sophisticated buffer management strategy.
2. We compare the performance of TDD with Ukkonen [52], McCreight [40], and a suffix array-based technique: Deep-Shallow [39] for the in-memory case, where all the data structures needed for building the suffix trees are memory resident (i.e., the datasets are "small"). Interestingly, we show that even though Ukkonen and McCreight have a better worst-case *theoretical* complexity on a random access machine, TDD and Deep-Shallow perform better on modern cached processors because they incur fewer cache misses.
3. We systematically explore the space of data sizes and types, and highlight the advantages and disadvantages of TDD with respect to other construction algorithms.
4. We experimentally demonstrate that TDD scales gracefully with increasing input size. With extensive experimental evaluation, we show that TDD outperforms existing disk-based construction methods. Using the TDD process, we are able to construct a suffix tree on the *entire human genome in 30 h* on a single processor machine! To the best of our knowledge, suffix tree construction on an input string of this size (approximately three billion symbols) has yet to be reported in literature.
5. We describe a new algorithm called ST-Merge that is based on a partition and merge strategy. We experimentally show that ST-Merge algorithm is more efficient than TDD when the input string size is significantly larger than the available memory. However, for most current biological sequence datasets on modern machines with large memory configuration, TDD is the algorithm of choice.

The remainder of this paper is organized as follows: Sect. 2 discusses related work. The TDD technique is described in Sect. 3, and we analyze the behavior of this algorithm in Sect. 4. The ST-Merge algorithm is presented in Sect. 5. Sect. 6 describes the experimental results, and Sect. 7 presents our conclusions.

2 Related work

Linear time algorithms for constructing suffix trees have been described by Weiner [54], McCreight [40], and Ukkonen [52]. (For a discussion on the relationship between these algorithms, see [24].) Ukkonen's is a popular algorithm because it is easier to implement than the other algorithms. It is an $O(n)$, in-memory construction algorithm based on the clever observation that constructing the suffix tree can be performed by iteratively expanding the leaves of a partially constructed suffix tree. Through the use of *suffix links*, which provide a mechanism for quickly traversing across subtrees, the suffix tree can be expanded by simply adding the $i + 1^{st}$ character to the leaves of the suffix tree built on the previous i characters. The algorithm thus relies on suffix links to traverse through all of the subtrees in the main tree, expanding the outer edges for each input character. McCreight's algorithm is a space-economical linear time suffix tree construction algorithm. This algorithm starts from an empty tree and inserts suffixes into the partial tree from the longest to the shortest suffix. Like Ukknonen's algorithm, McCreight's algorithm also utilizes suffix links to traverse from one part of the tree to another. Both are linear time algorithms, but they have poor locality of reference. This leads to poor performance on cached architectures and on disk.

Variants of suffix trees construction algorithms have been considered for disk-based construction [27]. Recently, Bedathur and Haritsa developed a buffering strategy, called TOP-Q, which improves the performance of Ukkonen's algorithm (which uses suffix links) when constructing on-disk suffix trees [7]. A different approach was suggested by Hunt et al. [28] where the authors drop the use of suffix links and use an $O(n^2)$ algorithm with a better locality of memory reference. In one pass over the string, they index all suffixes with the same prefix by inserting them into an on-disk subtree managed by PJama [6], a Java-based object store. Construction of each independent subtree requires a full pass over the string. The main drawback of Hunt's algorithm is that the tree traversal incurs a large number of random accesses during the construction process. A partition and clustering-based approach is described by Schürmann and Stoye in [46], which is an improvement over Hunt et al. This approach uses clustering to better organize disk accesses. A partitioning-based approach was suggested by Clifford and Sergot in [13] to build distributed and paged suffix trees. However, this is an in-memory technique. Cheung et al. [12] have recently proposed an algorithm called *DynaCluster*. This algorithm employs a dynamic clustering technique to reduce the random accesses that are incurred during the tree traversal. Every cluster contains tree nodes that are frequently referenced by each other. In this paper, we compare our suffix tree construction methods with TOP-Q [7], Hunt's [28] method, and DynaCluster [12], and show that in practice our methods for constructing suffix trees are more efficient.

A top-down suffix tree construction approach has been suggested in [3]. In [25], Giegerich, Kurtz, and Stoye explore the benefits of using a lazy implementation of suffix trees. In this approach, the authors argue that one can avoid paying the full construction cost by constructing the subtree only when it is accessed for the first time. This approach is useful either when a small number of queries are posed or only short queries are posed against a string dataset. When executing a large number of (longer) queries, most of the tree must be materialized, and in this case, this approach will perform poorly.

Previous research has also produced theoretical results on understanding the average sizes of suffix trees [9, 48], and theoretical complexity of using sorting to build suffix trees. In [21], Farach describes a linear time algorithm by constructing odd and even suffix trees, and merging them. In [22], the authors show that this algorithm has the same I/O complexity as sorting on the DAM model described by Vitter and Shriver [53]. However, they do not differentiate between random and sequential I/O. In contrast, our approach makes careful choices in order to reduce random I/O, and incurs mostly sequential I/O.

Suffix arrays are closely related to suffix trees, and can be used as an alternative to suffix trees for many string matching tasks [1, 11, 14, 42]. A suffix tree can also be constructed by first building a suffix array. With the help of an additional longest common prefix (LCP) array, a suffix array can be converted into a suffix tree in $O(n)$ time. Theoretical linear time suffix array construction algorithms have been proposed in [31, 33, 34]. There has also been considerable interest in practical suffix array construction algorithms. The Deep-Shallow algorithm proposed in [39] is a space-efficient internal memory suffix array construction algorithm. Although its worst-case time complexity is $\Theta(n^2 \log n)$, it is arguably the fastest in-memory method in practice. In [31, 32, 38], algorithms for constructing LCP arrays in linear time are proposed.

The long interest of the algorithmic community in optimal external memory suffix array construction algorithms has led to the external DC3 algorithm recently proposed by Dementiev et al. [18]. This external construction method is based on the Skew algorithm [31]. The Skew algorithm is a theoretically optimal suffix array construction algorithm, and uses a merge-based approach. This method recursively reduces the suffix array construction using a two-thirds–one-thirds split of the suffix array. Each recursive call first sorts the larger array, and the smaller array is sorted using the ordering information in the larger array. The arrays are merged to produce the final array. The external DC3 algorithm extends the in-memory Skew algorithm with the help of the STXXL library [47]. The STXXL library is a C++ template library that enables containers and algorithms to process large amounts of data that do not fit in main memory. It also improves performance by supporting multiple disks and overlapping I/O with CPU computation (see [47] for details). The external DC3 algorithm [18] is theoretically optimal and superior to the previous external suffix array construction methods in practice. We draw some comparisons between our methods and the external DC3 algorithm in Sect. 6.5, and show that in practice TDD is faster than the external DC3 algorithm.

TDD uses a simple partitioning strategy. However, a more sophisticated partitioning method was recently proposed by Carvalho et al. [10], which can complement our existing partitioning method.

3 The TDD technique

Most suffix tree construction algorithms do not scale due to the prohibitive disk I/O requirements. The high per-character space overhead of a suffix tree quickly causes the data structures to outgrow main memory, and the poor locality of reference makes efficient buffer management difficult.

We now present a new disk-based construction technique called the "Top-Down Disk-based" technique, hereafter referred to simply as TDD. TDD scales much more gracefully than existing techniques by reducing the main-memory requirements through strategic buffering of the largest data structures. The TDD technique consists of a suffix tree construction algorithm, called PWOTD, and the related buffer management strategy described in the following sections.

3.1 PWOTD algorithm

The first component of the TDD technique is our suffix tree construction algorithm, called Partition and Write Only Top Down (PWOTD). This algorithm is based on the *wotdeager* algorithm suggested by Giegerich et al. [25]. We improve on this algorithm by using a partitioning phase which allows one to immediately build larger, independent subtrees in memory. (A similar partitioning strategy was proposed in [46].) Before we explain the details of our algorithm, we briefly discuss the representation of the suffix tree.

The suffix tree is represented by a linear array, just as in *wotdeager*. This is a compact representation using 8.5 bytes

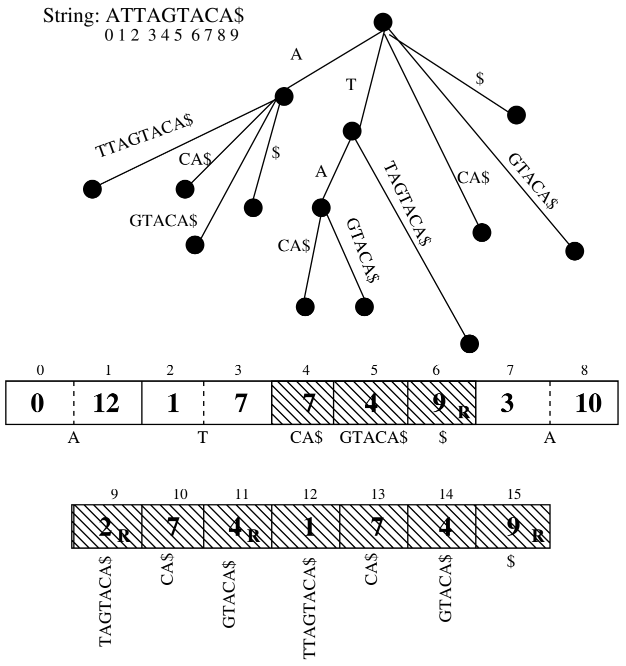

Fig. 1 Suffix tree representation (leaf nodes are shaded, the right-most child is denoted with an R)

```
Algorithm PWOTD(String,prefixlen)
Phase 1:
Scan the String and partition Suffixes based
on the first prefixlen symbols of each suffix
Phase 2: Do for each partition:
1. START BuildSuffixTree
2. Populate Suffixes from current partition
3. Sort Suffixes on first symbol using Temp
4. Output branching and leaf nodes to the Tree
5. Push the nodes pointing to an unevaluated range
   onto the Stack
While Stack is not empty
    6. Pop a node
    7. Find the Longest Common Prefix (LCP) of
       all the suffixes in this range by checking
       the String
    8. Sort the range in Suffixes on the first
       symbol using Temp
    9. Write out branching nodes or leaf nodes to Tree
    10.Push the nodes pointing to an unevaluated range
       onto the Stack
11. END
```

Fig. 2 The TDD algorithm

per indexed symbol in the average case with 4 byte integers. Fig. 1 illustrates a suffix tree on the string ATTAGTACA$ and the tree's corresponding array representation in memory. Shaded entries in the array represent leaf nodes, with all other entries representing non-leaf nodes. An *R* in the lower right-hand corner of an entry denotes a right-most child. Note that leaf nodes are represented using a single integer, while non-leaf nodes use two integers. (The two entries of a non-leaf node are separated by a dashed line in the figure.) The first entry in a non-leaf node is an index into the input string; the character at that index is the starting character of the incoming edge's label. The length of the label can be deduced by examining the children of the current node. The second entry in a non-leaf node points to the first child. For example, in Fig. 1, the non-leaf node represented by the entries 0 and 1 in the tree array has four leaf children located at entries 12, 13, 14, and 15, respectively. The parent's suffix starts at index 0 in the string, whereas the children's suffixes begins with the indexes 1, 7, 4, and 9, respectively. Therefore, we know the length of the parent's edge label is $min\{1, 7, 4, 9\} - 0 = 1$. Note that the leaf nodes do not have a second entry. The leaf node requires only the starting index of the label; the end of the label is the string's terminating character. See [25] for a more detailed explanation.

The PWOTD algorithm consists of two phases. In the first phase, wepartition the suffixes of the input string into $|A|^{prefixlen}$ partitions, where $|A|$ is the alphabet size of the string and *prefixlen* is the depth of the partitioning. The partitioning step is executed as follows. The input string is scanned from left to right. At each index position i the *prefixlen* subsequent characters are used to determine one of the $|A|^{prefixlen}$ partitions. This index i is then written to the calculated partition's buffer. At the end of the scan, each partition will contain the suffix pointers for suffixes that all have the same prefix of size *prefixlen*. Note that the number of partitions ($|A|^{prefixlen}$) is much smaller than the length of the string.

To further illustrate the partition step, consider the following example. Partitioning the string ATTAGTACA$ using a *prefixlen* of 1 would create four partitions of suffixes, one for each symbol in the alphabet. (We ignore the final partition consisting of just the string terminator symbol $.) The suffix partition for the character A would be {0, 3, 6, 8}, representing the suffixes {ATTAGTACA$, AGTACA$, ACA$, A$}. The suffix partition for the character T would be {1,2,5} representing the suffixes {TTAGTACA$, TAGTACA$, TACA$}. In phase two, we use the *wotdeager* algorithm to build the suffix tree on each partition using a top down construction.

The pseudo-code for the PWOTD algorithm is shown in Fig. 2. While the partitioning in phase one of PWOTD is simple enough, the algorithm for *wotdeager* in phase two warrants further discussion. We now illustrate the *wotdeager* algorithm using an example.

3.1.1 Example illustrating the wotdeager algorithm

The PWOTD algorithm requires four data structures for constructing suffix trees: an input string array, a suffix array, a temporary array, and the suffix tree. For the discussion that follows, we name each of these structures *String*, *Suffixes*, *Temp*, and *Tree*, respectively.

The Suffixes array is first populated with suffixes from a partition after discarding the first *prefixlen* characters.Using the same example string as before, ATTAGTACA$ with *prefixlen*= 1, consider the construction of the Suffixes array for the T-partition. The suffixes in this partition are at positions 1, 2, and 5. Since all these suffixes share the same prefix, T, we add one to each offset to produce the new Suffix array {2, 3, 6}. The next step involves sorting this array of suffixes based on the first character. The first characters of each suffix are {T, A, A}. The sorting is done in linear time using an algorithm called *count-sort* (for a constant alphabet size). In a single pass, for each character in the alphabet, we count the number of occurrences of that character as the first character of each suffix, and copy the suffix pointers into the Temp array. We see that the count for A is 2 and the count for T is 1; the counts for G, C, and $ are 0. We can use these counts to determine the character group boundaries: group A will start at position 0 with two entries, and group T will start at position 2 with one entry. We make a single pass through the Temp array and produce the Suffixes array sorted on the first character. The Suffixes array is now {2, 6, 3}. The A-group has two members and is therefore a branching node. These two suffixes completely determine the subtree below this node. Space is reserved in the Tree to write this non-leaf node once it is expanded, then the node is pushed onto the stack. Since the T-group has only one member, it is a leaf node and will be immediately written to the Tree. Since no other children need to be processed, no additional entries are added to the stack, and this node will be popped off first.

Once the node is popped off the stack, we find the longest common prefix (LCP) of all the nodes in the group {3, 6}. We examine position 4 (G) and position 7 (C) to determine that the LCP is 1. Each suffix pointer is incremented by the LCP, and the result is processed as before. The computation proceeds until all nodes have been expanded and the stack is empty. Fig. 1 shows the complete suffix tree and its array representation.

3.1.2 Discussion of the PWOTD algorithm

Observe that the second phase of PWOTD operates on subsets of the suffixes of the string. In *wotdeager*, for a string of n symbols, the size of the Suffixes array and the Temp array needed to be $4 \times n$ bytes (assuming 4 byte integers are used as pointers). By partitioning in the first phase, the amount of memory needed by the suffix arrays in each run is just $(4 \times n)/(|A|^{prefixlen})$ on average. (Some partitions might be smaller and some larger than this fig. due to skew in real world data. Sophisticated partitioning techniques can be used to balance the partition sizes [10].) The important point is that partitioning decreases the main-memory requirements for suffix tree construction, allowing independent subtrees to be built entirely in main memory. Suppose we are partitioning a 100 million symbol string over an alphabet of size 4. Using a $prefixlen = 2$ will decrease the space requirement of the Suffixes and Temp arrays from 400 MB to approximately 25 MB each, and the Tree array from 1200 to approximately 75 MB. Unfortunately, this savings is not entirely free. The cost of the partitioning phase is $O(n \times prefixlen)$, which increases linearly with *prefixlen*. For small input strings where we have sufficient main memory for all the structures, we can skip the partitioning phase entirely. It is not necessary to continue partitioning once the Suffixes and Temp arrays fit into memory. For even very large datasets, such as the human genome, partitioning with *prefixlen* more than 7 is not beneficial.

3.2 Buffer management

Since suffix trees are an order of magnitude larger in size than the input data strings, suffix tree construction algorithms require large amounts of memory, and may exceed the amount of main memory that is available. For such large datasets, efficient disk-based construction methods are needed that can scale well for large input sizes. One strength of TDD is that its data structures transition gracefully to disk as necessary, and individual buffer management polices for each structure are used. As a result, TDD can scale gracefully to handle large input sizes.

Recall that the PWOTD algorithm requires four data structures for constructing suffix trees: *String*, *Suffixes*, *Temp*, and *Tree*. Figure 3 shows each of these structures as separate, in-memory buffer caches. By appropriately allocating memory and by using the right buffer replacement policy for each structure, the TDD approach is able to build suffix trees on extremely large inputs. The buffer management policies are summarized in Fig. 3 and are discussed in detail below.

The largest data structure is the Tree buffer. This array stores the suffix tree during its intermediate stages as well as the final computed result. The Tree data structure is typically 8-12 times the size of the input string. The reference pattern to Tree consists mainly of sequential writes when the children of a node are being recorded. Occasionally, pages are revisited when an unexpanded node is popped off the stack. This access pattern displays very good temporal and spatial locality. Clearly, the majority of this structure can be placed

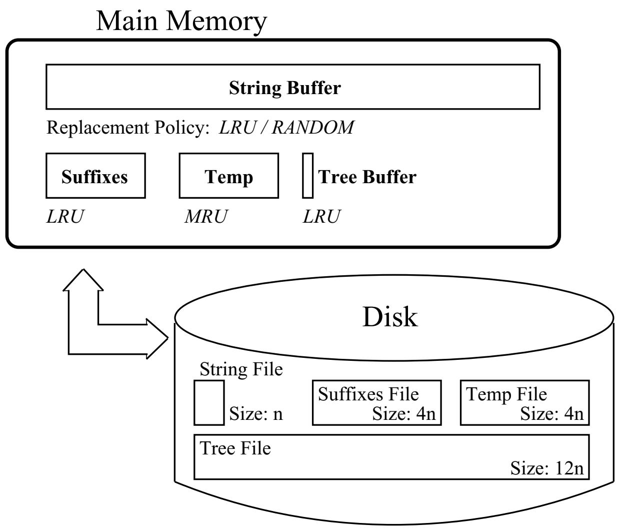

Fig. 3 Buffer management schema

on disk and managed efficiently with a simple LRU (*L*east *R*ecently *U*sed) replacement policy.

The next largest data structures are the Suffixes and the Temp arrays. The Suffixes array is accessed as follows: first a sequential scan is used to copy the values into the Temp array. The count phase of the count-sort is piggybacked on this sequential scan. The sort operation following the scan causes writes back into the Suffixes array. However, there is some locality in the pattern of writes in the Suffixes array, since the writes start at each character-group boundary and proceed sequentially to the right. Based on the (limited) locality of reference, one expects LRU to perform reasonably well. The Temp array is referenced in two sequential scans: the first to copy all of the suffixes in the Suffixes array, and the second to copy all of them back into the Suffixes array in sorted order. For this reference pattern, replacing the most recently used page (MRU) works best.

The String array has the smallest main-memory requirement of all the data structures, but the worst locality of access. The String array is referenced when performing the count-sort and to find the longest common prefix in each sorted group. During the count-sort all of the portions of the string referenced by the suffix pointers are accessed. Though these positions could be anywhere in the string, they are always accessed in left to right order. In the function to find the longest common prefix of a group, a similar pattern of reference is observed. In the case of this find-LCP function, each iteration will access the characters in the string, one symbol to the right of those previously referenced. In the case of the count-sort operation, the next set of suffixes to be sorted will be a subset of the current set. This is a fairly complex reference pattern, and there is some locality of reference, so we expect LRU and RANDOM to do well. Based on evidence in Sect. 6.4, we see that both are reasonable choices.

3.3 Buffer size determination

To obtain the maximum benefit from buffer management policy, it is important to divide the available memory amongst the data structures appropriately. A careful apportioning of the available memory among these data structures can affect the overall execution time dramatically. In the rest of this section, we describe a technique to divide the available memory among the buffers.

If we know the access pattern for each of the data structures, we can devise an algorithm to partition the memory to minimize the overall number of buffer cache misses. Note that we need only an access pattern on a string representative of each class, such as DNA sequences, protein sequences, etc. In fact, we have found experimentally that these access patterns are similar across a wide-range of datasets (we discuss these results in detail in Sect. 6.4.) An illustrative graph of the buffer cache miss pattern for each data structure is shown in Fig. 4. In this figure, the *X*-axis represents the number of pages allocated to the buffer as a percentage of the total size of the data structure. The *Y*-axis shows the number of cache misses. This figure is representative of biological sequences, and it is based on data derived from actual experiments in Sect. 6.4.

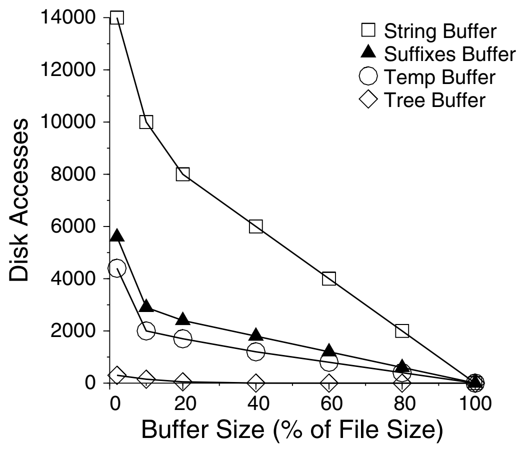

Fig. 4 Sample page miss curves

As we will see at the end of Sect. 3.3.1, our buffer allocation strategy needs to estimate only the relative magnitudes of the slopes of each curve and the position of the "knee" towards the start of the curve. The full curve as shown in Fig. 4 is not needed for the algorithm. However, it is useful to facilitate the following discussion.

3.3.1 TDD heuristic for allocating buffers

We know from Fig. 4 that the cache miss behavior for each buffer is approximately linear once the memory is allocated beyond a minimum point. Once we identify these points, we can allocate the minimum buffer size necessary for each structure. The remaining memory is then allocated in order of decreasing slopes of the buffer miss curves.

We know from arguments in Sect. 3.2 that references to the String have poor locality. One can infer that the String data structure is likely to require the most buffer space. We also know that the references to the Tree array have very good locality, so the buffer space it needs is likely to be a very small fraction of its full size. Between Suffixes and Temp, we know that the Temp array has more locality than the Suffixes array, and will therefore require less memory. Both Suffixes and Temp require a smaller fraction of their pages to be resident in the buffer cache when compared to the String. We exploit this behavior to design a heuristic for memory allotment.

We suggest the minimum number of pages allocated to the Temp and Suffixes arrays to be $|A|$. During the sort phase, we know that the Suffixes array will be accessed at $|A|$ different positions which correspond to the character group boundaries. The incremental benefit of adding a page will be very high until $|A|$ pages, and then one can expect to see a change in the slope at this point. By allocating at least $|A|$ pages, we avoid the penalty of operating in the initial high miss-rate region. The TDD heuristic chooses to allocate a minimum of $|A|$ pages to Suffixes and Temp first.

We suggest allocating two pages to the Tree array. Two pages allow a parent node, possibly written to a previous page and then pushed onto the stack for later processing, to be accessed without replacing the current active page. This saves a large amount of I/O over choosing a buffer size of only one page.

The remaining pages are allocated to the String array upto its maximum required amount. If any pages are left over, they are allocated to Suffixes upto its maximum requirement. The remaining pages (if any) are allocated to Temp, and finally to Tree.

The reasoning behind this heuristic is borne out by the graphs in Fig. 4. The String, which has the least locality of reference, has the highest slope and the largest magnitude. Suffixes and Temp have a lower magnitude and a more gradual slope, indicating that the improvement with each additional page allocated is smaller. Finally, the Tree, which has excellent locality of reference, is nearly zero. All curves have a knee which we estimate by choosing minimum allocations.

3.3.2 An example allocation

The following example demonstrates how to allocate the main memory to the buffer caches. Assume that your system has 100 buffer pages available for use and that you are building a suffix tree on a small string that requires 6 pages. Further assume that the alphabet size is 4 and that 4 byte integers are used. Assuming that no partitioning is done, the Suffixes array will need 24 pages (one integer for each character in the String), the Temp array will need 24 pages, and the Tree will need at most 72 pages. First we allocate 4 pages each to Suffixes and Temp. We allocate 2 pages to Tree. We are now left with 90 pages. Of these, we allocate 6 pages to the String, thereby fitting it entirely in memory. From the remaining 84 pages, Suffixes and Temp are allocated 20 and fit into memory, and the final 44 pages are all given to Tree. This allocation is shown pictorially in the first row of Fig. 5.

Similarly, the second row in Fig. 5 is an allocation for a medium-sized input of 50 pages. The heuristic allocates 2 pages to the Tree, 4 to the Temp array, 44 to Suffixes, and 50 to the String. The third allocation corresponds to a large string of 120 pages. Here, Suffixes, Temp, and Tree are allocated their minimums of 4, 4, and 2, respectively, and the rest of the memory (90 pages) is given to String. Note that the entire string does not fit in memory now, and portions will be swapped into memory from disk when they are needed.

Observe from Fig. 5 that when the input is small and all the structures fit into memory, most of the space is occupied by the largest data structure: the Tree. As the input size increases , the Tree is pushed out to disk. For very large strings that do not fit into memory, everything but the String is pushed out to disk, and the String is given nearly all of the memory. By first pushing the structures with better locality of reference onto disk, TDD is able to scale gracefully to very large input sizes.

Note that our heuristic does not need the actual utility curves to calculate the allotments. It estimates the “knee” of each curve using the algorithm, and assumes that the curve is linear for the rest of the region.

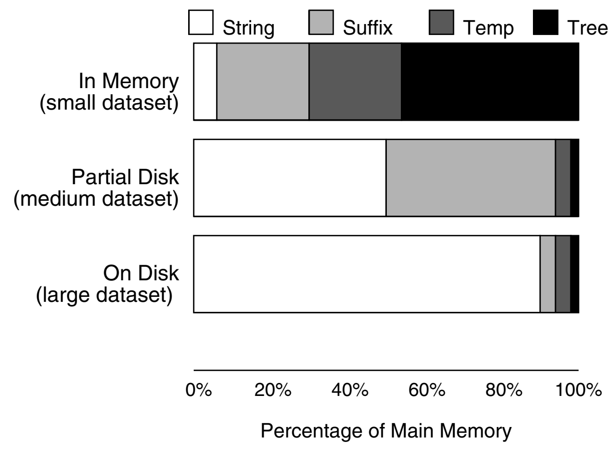

Fig. 5 Buffer allocation for different data structures: Note how other data structures are gradually pushed out of memory as the input string size increases

4 Analysis

In this section, we analyze the advantages and the disadvantages of using the TDD technique for various types and sizes of string data. We also describe how the design choices we have made in TDD overcome the performance bottlenecks present in other proposed techniques.

4.1 I/O benefits

Unlike the approach of [7] where the authors use the in-memory $O(n)$ algorithm (Ukkonen) as the basis for their disk-based algorithm, we use the theoretically less efficient $O(n^2)$ *wotdeager* algorithm [25]. A major difference between the two algorithms is that Ukkonen’s algorithm sequentially accesses the string data and then updates the suffix tree through random traversals, while our TDD approach accesses the input string randomly and then writes the tree sequentially. For disk-based construction algorithms, random access is the performance bottleneck as on each access an entire page will potentially have to be read from disk; therefore, efficient caching of the randomly accessed disk pages is critical.

On first appearance, it may seem that we are simply trading some random disk I/O for other random disk I/O, but the input string is the smallest structure in the construction algorithm, while the suffix tree is the largest structure. TDD can place the suffix tree in very small buffer cache as the writes are almost entirely sequential, which leaves the remaining memory free to buffer the randomly accessed, but much smaller, input string. Therefore, our algorithm requires a much smaller buffer cache to contain the randomly accessed data. Conversely, for the same amount of buffer cache, we can cache much more of the randomly accessed pages, allowing us to construct suffix trees on much larger input strings.

4.2 Main-memory analysis

When we build suffix trees on *small* strings (i.e., when the string and all the data structures fit in memory), no disk I/O is incurred. For the case of in-memory construction, one would expect that a linear time algorithm such as Ukkonen or McCreight would perform better than the TDD approach, which has a worst-case complexity of $O(n^2)$. However, one must consider more than just the theoretical complexity to understand the execution time of the algorithms.

Traditionally, in designing disk-based algorithms, all accesses to main memory are considered equally good, as the disk I/O is the performance bottleneck. However, for programs that incur little disk I/O, the performance bottleneck shifts to the main-memory hierarchy. Modern processors typically employ one or more data caches for improving access time to memory when there is a lot of spatial and/or temporal locality in the access patterns. The processor cache is analogous to a database's buffer cache, the primary difference being that the user does not have control over the replacement policy. Reading data from the processor's data cache is an order of magnitude faster than reading data from the main memory. Furthermore, as the speed of the processor increases, so does the main-memory latency (in terms of number of cycles). As a result, the latency of random memory accesses will only grow with future processors.

Linear time algorithms such as Ukkonen and McCreight require a large number of random memory accesses due to the linked list traversals through the tree structure. In Ukkonen, a majority of cache misses occur after traversing a suffix link to a new subtree and then examining each child of the new parent. The traversal of the suffix link to the sibling subtree and the subsequent search of the destination node's children require random accesses to memory over a large address space. Because this span of memory is too large to fit in the processor cache, each access has a very high probability of incurring the full main-memory latency. Similarly, McCreight's algorithm also traverses suffix links during construction, and incurs many cache misses. Furthermore, the rescanning and scanning steps used to find the extended locus of the head of the newly added suffix result in more random accesses. Using an array-based representation [35], where the pointers to the children are stored in an array with an element for each symbol in the alphabet, can reduce the number of cache misses. However, this representation uses a lot of space, potentially leading to higher execution time. In previous work, both McCreight [40] and TOP-Q [7] argue for the linked list based implementation as being a better choice.

Observe that when using the linked list implementation, as the alphabet size grows, the number of children for each non-leaf node will increase accordingly. As more children are examined to find the right position to insert the next character, the number of cache misses also increases. Therefore, Ukkonen's method will incur an increasing number of processor cache misses with an increase in alphabet size. Similarly, with McCreight's algorithm, an increase in alphabet size leads to more cache misses.

For TDD, the alphabet size has the opposite effect. As the branching factor increases, the working set of the Suffixes and Temp arrays quickly decreases, and can fit into the processor cache sooner. The majority of read misses in the TDD algorithm occur when calculating the size of each character group (in line 8 of Fig. 2). This is because the beginning character of each suffix must be read, and there is little spatial locality in the reads. While both algorithms must perform random accesses to main memory, incurring very expensive cache misses, there are three properties about the TDD algorithm that make it more suited for in-memory performance: (a) the access pattern is sequential through memory, (b) each random memory access is independent of the other accesses, and (c) the accesses are known a priori. A detailed discussion of these properties can be found in [25]. Because the accesses to the input data string are sequential through the memory address space, hardware-based data prefetchers may be able to identify opportunities for prefetching the cache lines [29]. In addition, techniques for overlapping execution with main-memory latency can easily be incorporated in TDD.

The Deep-Shallow algorithm of [39] is a space-efficient in-memory suffix array construction technique. It differentiates the cases of sorting suffixes with a short common prefix from sorting suffixes with a long common prefix. These two cases are called "shallow" sorting and "deep" sorting, respectively. The Bentley-Sedgewick multikey quick sort [8] is used as the shallow sorter, and a combination of different algorithms are used in the deep sorter. The memory reference pattern is different in the case of each algorithm, and a thorough analysis of the reference pattern is very complicated. This complex combination of different sorting strategies at different stages of suffix array construction turns out to perform very well in practice.

4.3 Effect of alphabet size and data skew

In this section, we consider the effect of alphabet size and data skew on TDD.

There are two properties of the input string that can affect the execution time of TDD: the size of the alphabet and the skew in the string. The average case running time for constructing a suffix tree on a Random Access Machine for *uniformly random input strings* is $O(n \log_{|A|} n)$, where $|A|$ is the size of the input alphabet and n is the length of the input string. (A uniformly random string can be thought of as a sequence generated by a source that emits each symbol in sequence from the alphabet set with equal probabilities, and the symbol emitted is independent of previous symbols.) The suffix tree has $O(\log_{|A|} n)$ levels [19], and at each level i, the suffixes array is divided into $|A|^i$ equal parts ($|A|$ is the branching factor, and the string is uniformly random.) The count-sort and the find-LCP (line 7 of Fig. 2) functions are called on each of these levels. The running time of count-sort is linear. To find the longest common prefix for a set of suffixes from a uniformly distributed string, the expected number of suffixes compared before a mismatch is slightly over 1. Therefore, the find-LCP function would return after just one or two comparisons most of the time. In some cases, the actual LCP is more than 1 and a scan of the entire suffixes is required. Therefore, in the case of uniformly random data, the find-LCP function is expected to run in constant time. At each of the $O(log_{|A|}n)$ levels, the amount of computation performed is $O(n)$. This gives rise to the overall average case running time of $O(n \log_{|A|} n)$. The same average case cost can be shown to hold for random strings generated by picking symbols independently from the alphabet with *fixed* non-uniform probabilities. [4] shows that the height of trees on such strings is $O(\log n)$, and a linear amount of work is done at each level, leading to an average cost of $O(n \log n)$.

The longest common prefix of a set of suffixes is actually the label on the incoming edge for the node that corresponds to this set of suffixes. The average length of all the LCPs computed while building a tree is equal to the average length of the labels on each edge ending in a non-leaf node. This average LCP length is dependent on the distribution of symbols in the data. Real datasets, such as DNA strings, have a skew that is particular to them. By nature, DNA often consists of large repeating sequences; different symbols occur with more or less the same frequency but certain patterns occur more frequently than others. As a result, the average LCP length is higher than that for uniformly distributed data.

Figure 6 shows a histogram for the LCP lengths generated while constructing suffix trees on the SwissProt protein database [5] and the first 50 MB of Human DNA from chromosome 1 [23]. Notice that both sequences have a high probability that the LCP length will be greater than 1. Even

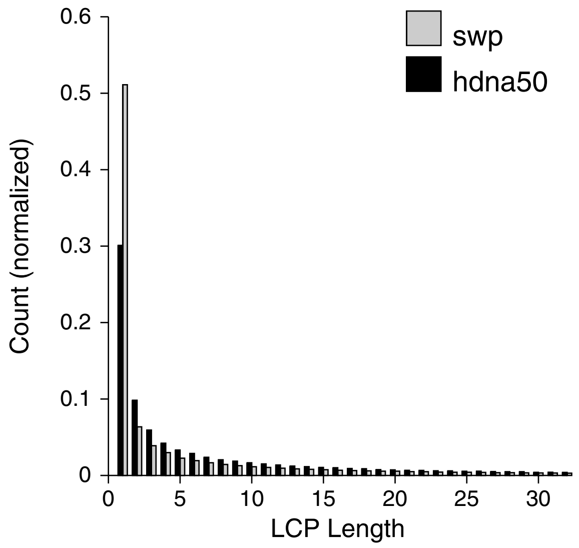

Fig. 6 LCP histogram: This figure plots the histogram until an LCP length of 32. For the DNA dataset, 18.8% of the LCPs have a length greater than 32, and for the protein dataset 13.8% of the LCPs have a length greater than 32

among biological datasets, the differences can be quite dramatic. From the figure, we observe that the DNA sequence is much more likely to have LCP lengths greater than 1 compared with the protein sequence (70% versus 50%). It is important to note that the LCP histograms for the DNA and protein sequences shown in the figure are not representative of all DNA and protein sequences, but these particular results do highlight the differences one can expect between input datasets.

For data with a lot of repeating sequences, the find-LCP function will not be able to complete in a constant amount of time. It will have to scan at least the first l characters of all the suffixes in the range, where l is the length of the actual LCP. In this case, the cost of find-LCP becomes $O(l \times r)$ where l is the length of the actual LCP, and r is the number of suffixes in the range that the function is examining. As a result, the PWOTD algorithm will take longer to complete.

TDD performs worse on inputs with many repeats such as DNA. On the other hand, Ukkonen's algorithm exploits these repeats by terminating an insert phase when a similar suffix is already in the tree. With long repeating sequences like DNA, this works in favor of Ukkonen's algorithm. Unfortunately, this advantage is not enough to offset the random reference pattern which still makes it a poor choice for large input strings when using cached architectures.

The size of the input alphabet also has an important effect. Larger input alphabets are an advantage for TDD because the running time is $O(n \log_{|A|} n)$, where $|A|$ is the size of the alphabet. A larger input alphabet size implies a larger branching factor for the suffix tree. This in turn implies that the working size of the Suffixes and Temp arrays shrinks more rapidly—and could fit into the cache entirely at a lower depth. For Ukkonen, a larger branching factor would imply that on an average, more siblings will have to be examined while searching for the right place to insert. This leads to a longer running time for Ukkonen. The same discussion also applies to McCreight's algorithm. There are hash-based and array-based approaches that alleviate this problem [35], but at the cost of consuming much more space for the tree. A larger tree representation naturally implies that for the in-memory case, we are limited to building trees on smaller strings.

Note that the case where Ukkonen's and McCreight's methods will have an advantage over TDD is for short input strings over a small alphabet size with high skew (repeat sequences). TDD is a better choice in all other cases. We experimentally demonstrate these effects in Sect. 6.

5 The ST-merge algorithm

The TDD technique works very well so long as the input string fits into available main memory. In Sect. 6, we show that if the input string does not fit completely in memory, accesses to the string will incur a large number of random I/O. Consequently, for input strings that are significantly larger

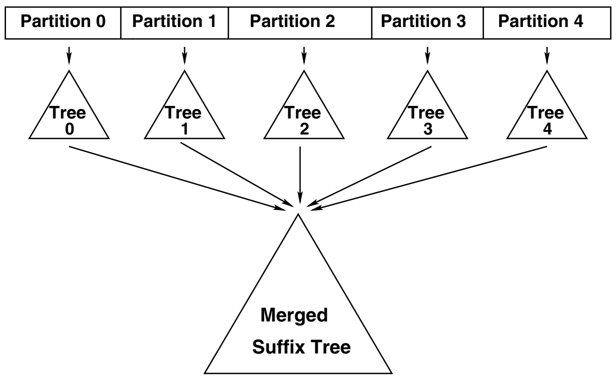

Fig. 7 The scheme for ST-merge

```
Algorithm ST-Merge
Phase 1
1. Partition the string into $k$ partitions
2. Build trees on each partition using TDD
Phase 2
1. NodeSet={}
2. For each tree to be merged
        3. Add the root of the tree to NodeSet
4. End For
5. NodeMerge(NodeSet, NULL)
```

Fig. 8 The ST-merge algorithm

```
NodeMerge(NodeSet,ParentEdge)
1. If ParentEdge == NULL
    2. Create a new node N for the merged tree
3. Else
    4. Create a new node N at the end of ParentEdge
5. Group the edges from the nodes in the NodeSet
    by the first character on each edge using a sort.
6. For each edge group
    7. If the group contains one edge
        8. Copy the edge and the subtree below
            from the corresponding source tree to
            node N of the merged tree
    9. Else
        10. EdgeSet={edges in the edge group}
        11. EdgeMerge(EdgeSet, N)
    12. End If
13.End For
```

Fig. 9 The NodeMerge subroutine

than the available memory the performance of TDD will rapidly degrade. In this section, we present a merge-based suffix tree construction algorithm that is more efficient than TDD when the input data string does not fit in main memory.

The ST-Merge algorithm employs a divide-and-conquer strategy similar to the external sort-merge algorithm. It is outlined in Fig. 7 and shown in detail in Fig. 8. While the ST-Merge algorithm can have more than one merge phase (as with sort-merge), here we only present a two-phase algorithm which has a single merge phase. (As with external sort-merge, in practice, this two-phase method is often sufficient with large main-memory configurations.) At a high-level, the ST-Merge algorithm works as follows: To construct a suffix tree for a string of size n, the algorithm first partitions the set of n suffixes into k disjoint subsets. Then a suffix tree is built on each of these subsets. Next, the intermediate trees are merged to produce the final suffix tree.

Note that the partitioning step of ST-Merge can be carried out in any arbitrary way–in fact, we could randomly assign a suffix to one of k buckets. However, we choose to partition the suffixes such that a given subset will contain only contiguous suffixes from the string. As we will discuss in detail in Sect. 5.1, using this partition strategy, we have a very high locality of access to the string when constructing the trees on each partition.

In the merging phase, the references to the input string have a more clustered access pattern, which has a better locality of reference than TDD. In addition, the ST-Merge method permits a number of merge strategies. For example, all the trees could be merged in a single merge step, or alternatively trees can be merged incrementally, i.e., trees are merged one after another. However, the first approach is preferable as it reduces the number of intermediate suffix trees that are produced (which may be written to the disk).

For building the suffix trees on the individual partitions, the ST-Merge algorithm simply uses the PWOTD algorithm. The subsequent merge phase is more complicated, and is described in detail below.

There are two main subroutines used in the merge phase: *NodeMerge* and *EdgeMerge*. The merge algorithm starts by merging the root nodes of the trees that are generated by the first phase. This is accomplished by a call to NodeMerge. EdgeMerge is used by NodeMerge when it is trying to merge multiple nodes that have outgoing edges with a common prefix. The NodeMerge and EdgeMerge subroutines are shown in Figs. 9 and 10, respectively.

The NodeMerge algorithm merges the nodes from the source trees and creates a merged node as the ending node of the parent edge in the merged suffix tree. Note that the parent edge of the merged node is NULL only when the roots of the source trees are merged. The NodeMerge algorithm first groups all the outgoing edges from the source nodes according to the first character along each edge, so that edges from each group share the same starting alphabet. If the alphabet set size is $|A|$, there are at most $|A|$ groups of edges. As the edges of each node are already sorted, replacement selection sort or count-sort can be used to generate the groups. Next, the algorithm examines each edge group. If the edge group contains only one edge, then it implies that this edge along with the subtree below is a branch of the merged node in the merged suffix tree. In this case, the algorithm simply copies the entire branch from the

```
EdgeMerge(EdgeSet, ParentNode)
1. Find the longest common prefix LCP of edges in EdgeSet
2. Create a new edge E, labeled with the LCP,
   as one outgoing edge of ParentNode
3. NodeSet={}
4. For each edge in EdgeSet
    5. If LCP is a proper prefix of the edge
        6. Create a new node to the corresponding tree,
           which breaks the edge at the end of LCP
        7. Add the new node to NodeSet
    8. Else
        9. Add the ending node of the edge to NodeSet
    10. End if
11. End For
12. NodeMerge(NodeSet, E)
```

Fig. 10 The EdgeMerge subroutine

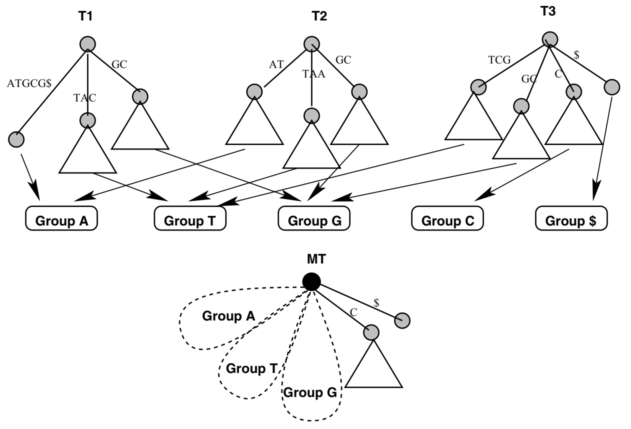

Fig. 11 Example of trees being merged: T1, T2, and T3 are three source trees to be merged. The final merged tree is MT. The triangle below a node represents the subtree under that node. The algorithm starts by calling NodeMerge on the trees T1–T3, which creates a root node for MT and groups the edges of the source trees according to the first character of each edge. This step produces five groups. Group A, T, and G all contain more than one edge, so EdgeMerge is called for each of these groups. Whereas group C and $ only have one edge, so the corresponding branches are copied to MT

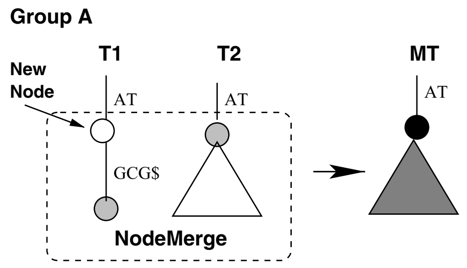

Fig. 12 EdgeMerge for group-A: We first create one outgoing edge from MT's root node, and label it with the LCP of the edges in group A. As the edge from T1 is longer than the LCP, we insert a new node in the middle of the long edge of T1 to split it into two edges labeled AT and GCG$, respectively. Then, NodeMerge is called on the newly created node in T1 and the node in T2 at the end of the label AT. NodeMerge then produces a node at the end of the edge AT in MT, as well as the subtree below it

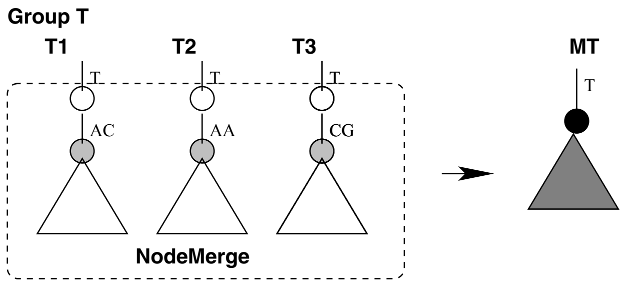

Fig. 13 EdgeMerge for group-T: The LCP of the edges in this group is a proper prefix of every edge, so we insert a node at the end of the LCP into every edge. The newly created nodes are then merged by making a call to NodeMerge

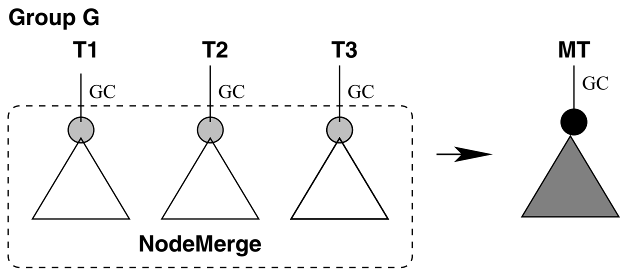

Fig. 14 EdgeMerge for group-G: All the edges are the same in this group. Consequently, the corresponding nodes ending at these edges are merged by making a call to NodeMerge

source tree to the merged tree. If a group contains more than one edge, the algorithm creates a new outgoing edge of the merged node. This step is carried out by calling EdgeMerge.

Note that NodeMerge will never need to merge a leaf node with an internal node. If such a case arose, it would mean that the suffix represented by the leaf node is a prefix of another suffix. This cannot happen since we add a terminating symbol to the end of the string to prevent this very case! The EdgeMerge algorithm merges together multiple edges that start with the same symbol. It first finds the longest common prefix (LCP) of the set of edges. Then, it creates a new edge in the result tree and labels it with the LCP. If any of the source edges have labels longer than the LCP, the edges are artificially split by inserting a node after the LCP. All the nodes ending at LCP now can be merged together with a call to NodeMerge, since they are all at the end of edges labeled identically.

A detailed example of ST-Merge is shown in Figs. 11–15.

5.1 Comparison with TDD

In this section, we present an analysis of the ST-Merge algorithm and discuss its relative advantages and disadvantages.

The main advantage for ST-Merge comes from the way it accesses the disk. In the partition and build phase, the algorithm accesses only a small portion of the string corresponding to that partition (the suffixes at the end of each

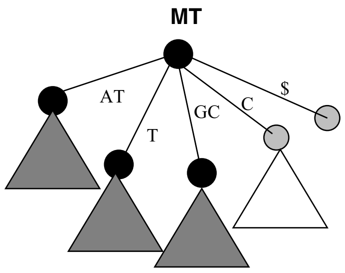

Fig. 15 The result of the merge

partition may require accesses that spill across the partition boundary). This ensures that most accesses to the string are in memory if the buffer for the String is at least the size of the partition. This can be much smaller than the whole string, and can therefore save a large amount of I/O. In fact, the first phase of the algorithm typically takes an order of magnitude less time than TDD. In the second phase, the input trees and the output tree are all sequentially accessed. So, each tree only requires a small buffer. The remaining memory is allocated to the string. Compared to TDD, the accesses to the string in the second phase of ST-Merge have more spatial locality of reference. This is because the accesses to the string (driven by the trees from phase 1) result in a smaller working set.

The decision of how many partitions to use in the first phase can be made using a simple formula. Suppose that M is the total amount of memory available. Let n be the size of the input string. The number of partitions to be used in the first phase is given by $k = \lceil \frac{n \times f}{M} \rceil$, where f (> 1) is an adjustment multiplication factor to account for overhead associated with the memory required for the auxiliary data structures, which are proportional in size to the input string. When the amount of main memory is greater than the string size, partitioning does not provide much benefit, and we simply use TDD.

Now, we examine the worst-case complexity of the merge algorithm. The first phase is $O(n^2)$ in the worst case. The second phase has two components: the cost of merging the nodes, and the cost of merging the edges. In the worst case, each node in the output tree ($O(n)$ nodes) is a result of merging k nodes from the source trees. This involves sorting at most $|A| \times k$ edges. Any sorting algorithm can be used to group the edges–a count-sort can do this in $O(|A| \times k)$ time. Therefore, the cost of merging the nodes is $O(n \times k)$ (assuming a constant-sized alphabet). The cost of merging the edges is the sum of the lengths of the edges of the source trees. This is because each symbol on an edge is considered at most once. In the worst case, the length of an edge is $O(n)$. This yields a worst-case cost of $O(n^2)$. Adding the three components, the worst-case complexity of ST-Merge is $O(n^2)$.

Next, we derive a loose bound for the average case complexity assuming that the string is generated by a Bernoulli source (i.e., the characters are drawn from the alphabet independently with fixed probabilities). The first phase takes $O(n \log \frac{n}{k})$, with k partitions each taking time $O(\frac{n}{k} \log \frac{n}{k})$. The cost of merging the edges is $O(n \log \frac{n}{k})$ on average, since the number of edges in the source trees totals $O(k \times \frac{n}{k})$, and the average length of the LCP is $O(\log \frac{n}{k})$ [4]. The worst-case complexity of merging the nodes serves as an upper bound for the average case cost. Adding the three components, the average cost of merging is $O(nk + n \log \frac{n}{k})$. As $k = \Theta(n)$, this is $O(n^2)$. Note that in practice with large main-memory configurations, k is usually a small number, since $k = \lceil \frac{n \times f}{M} \rceil$, where M is the size of the memory.

It is important to note that since ST-Merge writes a set of intermediate trees (the trees generated for each partition in the first phase) and merges them together for the final tree, the amount of data it writes is approximately twice the amount written by TDD (assuming that phase 2 requires only a single pass). However, this disadvantage is offset by the fact that the amount of memory required by the string buffer is smaller for ST-Merge and this results in less random I/O. The exact effect of these two factors depends on the ratio of the size of the string to the amount of memory available. In Sect. 6.6, we compare the execution times of TDD and ST-Merge.

6 Experimental evaluation

In this section, we present the results of an extensive experimental evaluation of the different suffix tree construction techniques. First, we compare the performance of TDD with Ukkonen's algorithm [52] and Kurtz's implementation [35] of McCreight's algorithm [40] for constructing in-memory suffix trees. For the in-memory case, we also compare these algorithms with an indirect approach that builds a suffix array first and converts the suffix array to a suffix tree. The suffix array method we choose is the Deep-Shallow algorithm [39], which is a fast, lightweight, in-memory suffix array construction algorithm. Then we compare TDD with Hunt's algorithm [28] for disk-based construction performance. We also evaluate the external DC3 algorithm [18], which is a fast disk-based suffix array construction technique. Finally, we examine the performance of ST-Merge and TDD when the input string is larger than the available memory.

6.1 Experimental setup and implementation

Our TDD algorithm uses separate buffer caches for the four main structures: the string, the suffixes array, the temporary working space for the count-sort, and the suffix tree. We use fixed-size pages of 8 K for reading and writing to disk. Buffer allocation for TDD is done using the method described in Sect. 3.3. If the amount of memory required is less than the size of the buffer cache, then that structure is loaded into the cache, with accesses to the data bypassing the buffer cache logic. TDD was written in C++ and compiled with GNU's g++ compiler version 3.2.2 with full optimizations activated.

For an implementation of Ukkonen's algorithm, we use the version from [55]. It is a textbook implementation based on Gusfield's description [26] and is written in C. The algorithm operates entirely in main memory, and there is no persistence. The suffix tree representation uses 32 bytes per node.

For the McCreight's algorithm we use the implementation that is part of the MUMmer software package [51]. This version of McCreight's algorithm is both space- and time-efficient, and the tree representation requires 10.1 bytes on average per input character.

The implementation of the Deep-Shallow suffix array construction algorithm is from [15]. Since this algorithm only constructs a suffix array, to build a suffix tree we augmented this method with a method for converting the suffix array to a suffix tree. For the remainder of this section, we refer to this Deep-Shallow implementation for constructing suffix trees as Deep-Shallow*. The conversion from suffix arrays to suffix trees requires the construction of an LCP array. For this implementation, we used the GetHeight algorithm proposed in [32]. We implemented a simple linear algorithm for converting a suffix array to a suffix tree as described in [2].

Our C++ implementation of Hunt's algorithm is from the OASIS sequence search tool [41], which is part of a larger project called Periscope [43]. The OASIS implementation uses a shared buffer cache instead of the persistent Java object store, PJama [6], described in the original proposal [28]. The buffer manager employs the CLOCK replacement policy. The OASIS implementation performed better than the implementation described in [28]. This is not surprising since PJama incurs the overhead of running through the Java Virtual Machine.

To compare TDD with a disk-based suffix array construction method, we used the external DC3 algorithm [18]. For the external DC3 suffix array construction algorithm, we use the code provided in [20]. The external DC3 algorithm from [20] can support multiple disks, but for all the disk-based methods including DC3, we used only one disk.

For the disk-based experiments that follow, unless stated otherwise, all I/O is to raw devices; i.e., there is no buffering of disk blocks by the operating system, and all reads and writes to disk are synchronous (blocking). This provides an unbiased accounting of the performance for disk-based construction as operating system buffering will not (positively) affect the performance. Therefore, our results present the worst-case performance for the disk-based construction methods. Using asynchronous writes is expected to improve the performance of our algorithm over the results presented. Each raw device accesses a single partition on one Maxtor Atlas 10K IV drive. The disk drive controller is an LSI 53C1030, Ultra 320 SCSI controller.

All experiments were performed on an Intel Pentium 4 processor with 2.8 GHz clock speed and 2 GB of main memory. This processor includes a two-level cache hierarchy. There are two first-level caches, named L1-I and L1-D, that cache instructions and data, respectively. There is also a single L2 cache that stores both instructions and data. The L1 data cache is an 8 KB, four-way set-associative cache with a 64 byte line size. The L1 instruction cache is a 12 K trace cache, four-way set associative. The L2 cache is a 512 KB, eight-way, set-associative cache, also with a 128 byte line size. The operating system was Linux, kernel version 2.4.20.

The Pentium 4 processor includes 18 event counters that are available for recording microarchitectural events, such as the number of instructions executed [30]. To access the event counters, the *perfctr* library was used [44]. The events measured include: clock cycles executed, instructions and micro-operations executed, L2 cache accesses and misses, Translation Lookaside Buffer (TLB) misses, and branch mispredictions.

6.2 Implications of 64-bit architectures

The implementation that we use for the evaluation presented in this section, is based on a 32-bit architecture. However, our code can easily be adapted to use 64-bit addressing. In this section, we briefly examine the impact of using 64-bit architectures, which can directly address more than 4 GB of physical memory.

We first investigate the memory requirement of the data structures used in our algorithms. There are two types of pointers in the data structures. The first type is a *string pointer*, which points to a position in the input string. The second type of pointer is a *node pointer*, which points to another node in the suffix tree. For the pointer to the string position, a 64-bit integer representation is needed only when the string size is larger than 4G (2^{32}) symbols. For the pointers to nodes, a 64-bit integer representation is needed only if the number of array entries in the suffix tree structure is more than 4G. Note that if the string has less than 4G symbols, and the suffix tree has more than 4G entries, then we can use a 32-bit representation for the string pointer and a 64-bit representation for the node pointer.

A non-leaf node in the suffix tree (the *Tree* structure shown in Fig. 1) has one string pointer and one node pointer, whereas a leaf node simply has one string pointer. In our tree representation, in addition to the tree array, we have 2 bits per entry in the tree array to indicate whether the entry is a leaf or a non-leaf, and whether the entry is the rightmost sibling (see Fig. 1 for details). The bit overhead is not affected by the changes to the pointer representation.

With a 32-bit representation for both string and node pointers, the size of a non-leaf node is 8 bytes, and the size of a leaf-node is 4 bytes. Going to a 64-bit representation adds four bytes for each pointer type that is affected.

In addition to the actual suffix tree (the *Tree* structure shown in Fig. 1), the suffix tree construction algorithm also uses two additional arrays, namely the *Suffixes* and *Temp* arrays. Both of them only contain string pointers. The size of the entries for both these arrays is 4 bytes with a 32-bit representation.

Note that TDD uses a partitioning method to construct the suffix trees (see Sect. 3 for details). This partitioning method constructs disjoint suffix trees based on the first few

Table 1 Main-memory data sources

Data source	Description	Symbols (10^6)
dmelano	D. Melanogaster Chr. 2 (DNA)	20
guten95	Gutenberg project, Year 1995 (English text)	20
swp20	Slice of SwissProt (Protein)	20
unif4	4-char alphabet, uniform distrib.	20
unif40	40-char alphabet, uniform distrib.	20
unif80	80-char alphabet, uniform distrib.	20

Table 2 Execution time details for Deep-Shallow*: Time spent by the algorithm in the three phases—suffix array construction (SA), LCP array construction (LCP), and suffix array to suffix tree conversion (Conv)

Data source	SA (s)	LCP (s)	Conv (s)	Total (s)
unif4	9.32	9.34	5.09	24.03
dmelano	10.65	9.69	7.25	27.59
swp20	9.57	9.22	4.86	23.65
unif40	7.87	10.61	3.98	22.46
guten95	9.31	8.1	4.58	21.78
unif80	7.53	9.98	3.67	21.18

symbols of the suffixes (the *prefixlen* variable in Fig. 2). Since each disjoint suffix tree only contains node pointers that point to nodes within the subtree, even when the *total* number of entries in the system is more than 4G, as long as each subtree has less than 4G entries, the node pointers can continue to use 32-bit representation.

6.3 Comparison of the in-memory algorithms

To evaluate the performance of the TDD technique for in-memory construction, we compare with the $O(n)$ time algorithms of Ukkonen and McCreight, and the Deep-Shallow* algorithm. We do not evaluate Hunt's algorithm in this section as it was not designed as an in-memory technique.

For this experiment, we used six different datasets: chromosome 2 of Drosophila Melanogaster from GenBank [23], a slice of the SwissProt dataset [5] containing 20 million symbols, and the text dataset from the 1995 collection from project Gutenberg [45]. The DNA dataset has an alphabet size of 5 (4 nucleotides, and the character 'N' for unknown positions). The protein dataset has an alphabet size of 23 (for the 20 amino acids, one character for representing unknown, and two characters to represent combinations), and the text dataset uses an alphabet of size 61 (all uppercase characters, numbers, and punctuation marks). We also chose three strings that contain uniformly distributed symbols from an alphabet of size 4, 40, and 80. The datasets used in this experiment are summarized in Table1.

Figure 16 shows the execution time breakdown for four algorithms, grouped by the datasets. In order, we present the times for TDD, Ukkonen, McCreight, and Deep-Shallow*. Note that since this is the in-memory case, TDD reduces to the PWOTD algorithm. In these experiments, all data structures fit into memory. The total execution time is decomposed into the time executing the following microarchitectural events (from bottom to top): instructions executed plus resource related stalls, TLB misses, branch mispredictions, L2 cache hits, and L2 cache misses (or main-memory reads).

From Fig. 16, we observe that the L2 cache miss component is a large contributor to the execution time for all algorithms. All algorithms show a similar breakdown for the small alphabet sizes of DNA data (unif4 and dmelano). When the alphabet size increases from 4 symbols to 20 symbols for *swp20*, then to 40 symbols for *unif40*, and finally to 80 symbols for *unif80*, the cache miss component of the suffix link based algorithms (Ukkonen and McCreight) increases dramatically, while it remains low for TDD. The reason for this, as discussed in Sect. 4.2, is that these algorithms incur a lot of cache misses while following the suffix link to a new portion of the tree, and in traversing all the children when trying to find the right position to insert the new entry. The suffix array-based method, Deep-Shallow*, does not exhibit this increase.

We observe that for each dataset, TDD outperforms the implementation of Ukkonen's algorithm that we use, and the performance difference increases with the alphabet size. This behavior was expected based on discussions in Sect. 4.3. TDD is faster than Ukkonen's method by a factor of 2.5 (*dmelano*)–16 (*unif80*). TDD also outperforms McCreight's algorithm for *swp20*, *unif40*, *guten95*, and *unif80* by a factor of 2.7, 6.2, 1.5, and 10.9, respectively. On the other two datasets, *unif4* and *dmelano*, the performance is nearly the same. Interestingly, the suffix array-based method, Deep-Shallow*, performs roughly as well as TDD. For the Deep-Shallow* algorithm, Table 2 shows the actual times spent in each of the three phases of the algorithm.

Collectively, these results demonstrate that despite having a $O(n^2)$ time complexity, the TDD technique significantly outperforms the implementations of the linear time algorithms of Ukkonen and McCreight on cached architectures. It does not, however, have any significant advantage over the suffix array-based Deep-Shallow* algorithm.

We must caution the reader, however, that this superior performance of TDD is not guaranteed in all cases. There

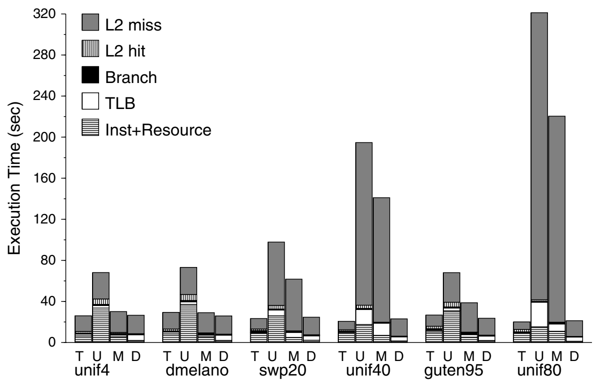

Fig. 16 In-memory execution time breakdown for TDD, Ukkonen, McCreight, and Deep-Shallow*

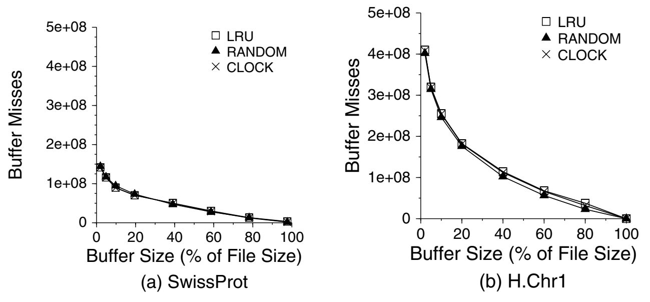

Fig. 17 String buffer

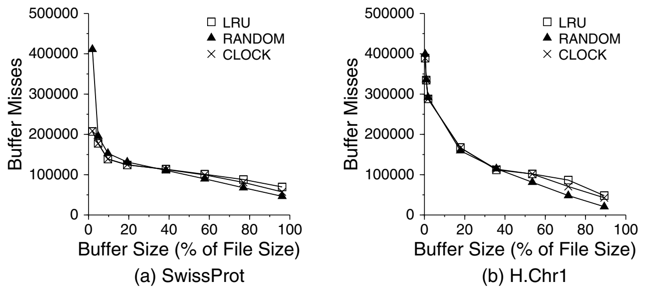

Fig. 18 Suffix buffer

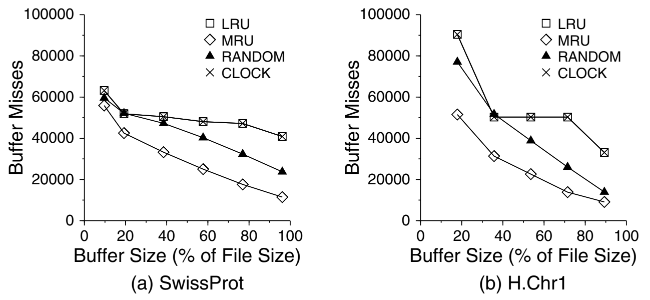

Fig. 19 Temp buffer

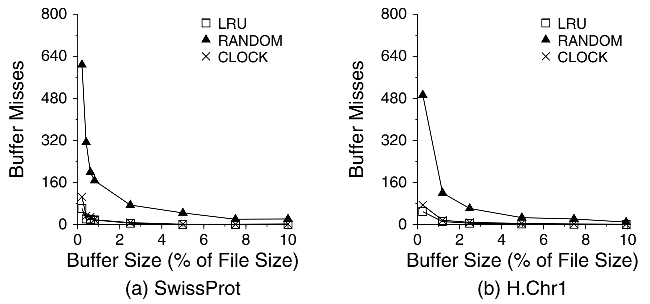

Fig. 20 Tree buffer

may be inputs with a small alphabet size and a high amount of skew on which Ukkonen or McCreight could outperform TDD, despite being less cache-efficient.

6.4 Buffer management with TDD

In this section, we evaluate the effectiveness of various buffer management policies on TDD. For each data structure

Table 3 The on-disk sizes of each data structure

Data structure	SwissProt (size in pages)	Human DNA) (size in pages)
String	6,250 (50 MB)	6,250 (50 MB)
Suffixes	1,250 (10 MB)	6,250 (10 MB)
Temp	1,250 (10 MB)	6,250 (50 MB)
Tree	4,100 (32.8 MB)	16,200 (129.6 MB)

used in the TDD algorithm, we analyze the performance of the LRU, MRU, RANDOM, and CLOCK page replacement polices over a wide range of buffer cache sizes. To facilitate this analysis over the wide range of variables, we employed a buffer cache simulator. The simulator takes as input a trace of the address requests into the buffer cache and the page size. The simulator outputs the disk I/O statistics for the desired replacement policy. For all the results shown here, except for the Temp array, MRU performs the worst by far and is not shown in the figures that we present in this section.

To generate the address request traces, we built suffix trees on the *SwissProt* database [5] and a 50 Mbp slice of the *Human Chromosome*-1 database [23]. A *prefixlen* of 1 was used for partitioning in the first phase. The total size of each of the arrays for these datasets is summarized in Table 3.

6.4.1 Page size

In order to determine the page size to use for the buffers, we conducted several experiments. We observed that larger page sizes produced fewer page misses when the alphabet size was large (protein datasets, for instance). Smaller page sizes seemed to have a slight advantage in the case of input sets with smaller alphabets (like DNA sequences). We observed that a page size of 8192 bytes performed well for a wide range of alphabet sizes. In the interest of space, we omit the details of our page-size study. For all the experiments described in this section we use a page size of 8 KB.

6.4.2 Buffer replacement policy

The results showing the effect of the various buffer replacement policies for the four data structures are presented in Figs. 17–20. In these figures, the *X*-axis is the buffer size (shown as a percentage of the original input string size), and the *Y*-axis is the number of buffer misses that are incurred by various replacement policies.

From Fig. 17, we observe that for the String buffer LRU, RANDOM, and CLOCK all perform similarly. Of all the arrays, when the buffer size is a fixed fraction of the total size of the structure, the String incurs the largest number of page misses. This is not surprising since this structure is accessed the most and in a random fashion. RANDOM and LRU are both good choices for the String buffer.

In the case of the Suffixes buffer (shown in Fig. 18), all three policies perform similarly for small buffer sizes. In the case of the Temp buffer, the reference pattern consists of one linear scan from left to right to copy the suffixes from the Suffixes array, and then another scan from left to right to copy the suffixes back into the Suffixes array in the sorted order. Clearly, MRU is the best policy in this case as shown by the results in Fig. 19. It is interesting to observe that the space required by the Temp buffer is much smaller than the space required by the Suffixes buffer to keep the number of misses down to the same level, though the array sizes are the same.

For the Tree buffer (see Fig. 20), with very small buffer sizes, LRU and CLOCK outperform RANDOM. However, this advantage is lost for even moderate buffer sizes. The most important observation to be made here is that despite being the largest data structure, it requires the smallest amount of buffer space, and takes a relatively insignificant number of misses for any policy. Therefore, for the Tree buffer, we can choose to implement the cheapest policy–the RANDOM replacement policy.

6.5 Comparison of disk-based algorithms

In this section, we first compare the performance of our technique with the technique proposed by Hunt et al. [28], which is currently considered the best practical disk-based suffix tree construction approach. We also compare the performance of TDD with the DC3 suffix array construction method [20]. Note that the DC3 method only constructs a suffix array and not the suffix tree. However, these results provide a lower bound on the cost of constructing a disk-based suffix tree using a suffix array construction method.

For this experiment, we used seven datasets which are described in Table 4. The construction times for the three algorithms are shown in Table 5.

From Table 5, we see that in each case TDD significantly outperforms Hunt's algorithm. On the TrEMBL dataset, TDD is faster by a factor of 7.4. For Human Chromosome-1, TDD is faster by a factor of 5.5. For a large text dataset like the Gutenberg Collection, TDD is nearly 10 times faster! For the largest dataset, the human genome, Hunt's algorithm did not complete in a reasonable amount of time. TDD finishes in less than 30 h. The 3 billion symbols of the human genome can be in memory if we use 4 bits per symbol, which

Table 4 On-disk data sources

Data source	Description	Symbols (10^6)
swp	Entire UniProt/SwissProt (Protein)	53
H.Chr1-50	50 Mbps slice of Human Chromosome-1 (DNA)	50
guten03	2003 Directory of Gutenberg Project (English Text)	58
trembl	TrEMBL (Protein)	338
H.Chr1	Entire Human Chromosome-1 (DNA)	227
guten (English Text)	Entire Gutenberg Collection	407
HG	Entire Human Genome (DNA)	3,000

Table 5 On-disk performance comparison

Data source	Symbols (10^6)	Hunt (min)	TDD (min)	Speed-up	DC3 (min)
swp	53	13.95	2.78	5.0	12.60
H.Chr1-50	50	11.47	2.02	5.7	12.67
guten03	58	22.5	6.03	3.7	13.78
trembl	338	236.7	32.00	7.4	102.78
H.Chr1	227	97.50	17.83	5.5	74.57
guten	407	463.3	46.67	9.9	120.53
HG	3, 000	—	30 h	—	—

is what was used to obtain the number in Table 5. The reason why TDD performs better is that Hunt's algorithm traverses the on-disk tree during construction, while TDD does not. During construction, a given node in the tree is written at most once in TDD. In addition, the careful management of the buffer sizes and the separate buffer replacement policies help reduce the disk I/O costs for TDD even further.

Next, we compare TDD with the fastest known disk-based suffix array construction algorithm—the disk-based DC3 algorithm [18]. These results are shown in Table 5. From Table 5, we can see that TDD is more than twice as fast as the external DC3 method in all cases. For HG, DC3 did not complete successfully. When the cost of building the LCP array and converting the suffix array to a suffix tree is added, the cost of this approach will be even higher (the number for the external DC3 algorithm in Table 5 only includes the time to build the suffix array). The suffix array construction algorithm works by recursively splitting the set of suffixes into a "two thirds" array (for suffixes starting at positions i such that $i \ mod\ 3 \neq 0$) and a "one thirds" array (for suffixes starting at positions i such that $i \ mod\ 3 = 0$). The larger array is sorted using radix sort, essentially giving lexicographic names to triples of symbols in the suffix. If there are two suffixes that cannot be distinguished by radix sort at this level, then an additional level of recursion is used where the lexicographic name is derived from three times as many symbols, and so on. The smaller array is sorted using the information from the "two thirds" array and then merged to this larger array using a fairly simple merge algorithm. In this algorithm, a large amount of random I/O is incurred during the radix sort. In addition, the amount of random I/O quickly increases as the recursion proceeds to a deeper level. This can happen very frequently with biological sequences where long repeats are common and deeper recursion is required to sort suffixes with longer LCPs.

6.5.1 Comparison of TDD with TOP-Q

Recently, Bedathur and Haritsa have proposed the TOP-Q technique for constructing suffix trees [7]. TOP-Q is a new low overhead buffer management method which can be used with Ukkonen's construction algorithm. The goal of the TOP-Q approach is to invent a buffer management technique that does not require modifying an existing in-memory construction algorithm. In contrast, TDD and Hunt's algorithm [28] take the approach of modifying existing suffix tree construction algorithms to produce a new disk-based suffix tree construction algorithm. Even though the research focus of TOP-Q is different from TDD and Hunt's algorithm, it is natural to ask how the TOP-Q method compares to these other approaches.

To compare TDD with TOP-Q, we obtained a copy of the TOP-Q code from the authors. This version of the code supports building suffix tree indices only on DNA sequences. As per the recommendation in [7], we used a buffer pool of 880 MB for the internal nodes and 800 MB for the leaf nodes (this was the maximum memory allocation possible with the TOP-Q code). On 50 Mbp of Human Chromosome-1, TOP-Q took about 78 min. By contrast, under the same conditions, TDD took about 2.1 min: faster by a factor of 37. On the entire Human Chromosome-1, TOP-Q took 5800 min, while our approach takes around 18 min. In this case, TDD is faster by 2 orders of magnitude!

6.5.2 Comparison of TDD with DynaCluster

The DynaCluster algorithm [12] is based upon Hunt's algorithm and tries to group nodes that are frequently referenced by each other into one cluster. The clusters are recursively created in a top-down fashion and a depth-first order. By using a dynamic clustering technique, DynaCluster reduces the random accesses to the suffix tree during construction time. However, just as in TOP-Q, DynaCluster is also inherently disadvantaged because they use clustering to improve what is a highly random reference pattern (on a large structure) to start with.

This is highlighted in the following comparison of I/O costs. In one of the their experiments in [12], the authors constructed the suffix tree for Human Chromosome-1 (224 MB) with a total of 864 MB of available memory. The I/O cost of this experiment is more than 800 s on their experimental platform. For computing the I/O costs, the authors used simulated disk numbers. Based on their method and the parameters in their paper (30 MB/s transfer rate, 8 KB pages), 800 s translates to 3 million disk reads/writes. For the same dataset and with identical parameters, TDD incurs 0.5 million disk accesses, which is around a sixth of that incurred by DynaCluster. This directly translates to a clear advantage for TDD.

6.6 Constructing suffix trees on very large inputs

In the previous section, we saw that TDD outperformed the other methods. The ST-Merge algorithm has advantages over TDD when the input string size is much larger than the main memory available ($\frac{n}{M} \gg 1$). When the input string fits in memory, ST-Merge is the same as the TDD algorithm.

Figure 21 shows the execution times of TDD and ST-Merge when the data string is much larger than the available main memory. To keep the running times for this experiment measurable, for this experiment only, we limited the total memory available to the algorithms to 6 MB, and varied the size of the input string from 10 to 80 MB. The other

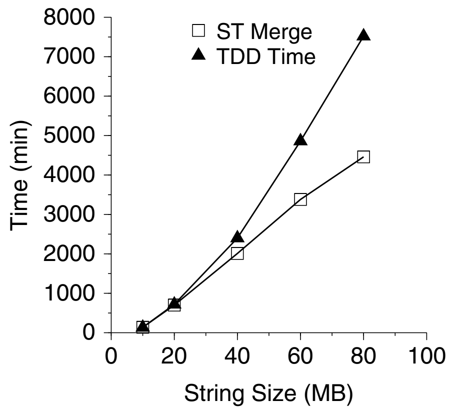

Fig. 21 Execution times : TDD and ST-merge

experimental conditions are the same as before. We note that our main motivation for using a small amount of main memory for this experiment is primarily to keep this experiment manageable. As can be seen in Fig. 21, even in this "scaled down" setting the execution time for the algorithms is very large–using a larger dataset with a larger amount of memory would have taken many days or weeks for each run. The "scaled down" setting exposes the behavior of these algorithms, while keeping the run times for the algorithms reasonable.

From Fig. 21, we observe, that when the input data string is significantly larger (about three times or more) than the main-memory size, the ST-Merge algorithm starts to outperform the TDD algorithm. We also observe that as the ratio $\frac{n}{M}$ increases, the ST-Merge algorithm has a larger advantage over TDD. This is expected because TDD incurs an increasingly larger penalty from the random I/O on the string. Consequently, for very large datasets, in which case the input string is significantly larger than the available main memory, ST-Merge is clearly the algorithm of choice.

7 Conclusions and future work

Practical methods for suffix tree construction on large character sequences have been virtually intractable. Existing approaches have excessive memory requirements and poor locality of reference and therefore do not scale well for even moderately sized datasets.

We first compare different algorithms used for constructing suffix trees in-memory. We demonstrate that our method (which is essentially PWOTD for the in-memory case) has an advantage over Ukkonen's algorithm by a factor of 2.5–16. It is also better than McCreight in some cases by up to a factor of 10. We argue that PWOTD wins over Ukkonen and McCreight because of superior cache performance. We also show that PWOTD is competitive with the suffix array based Deep-Shallow* algorithm and takes nearly the same time on various inputs.

To address the problem of disk-based suffix tree construction and unlock the potential of this powerful indexing structure, we have introduced the "Top Down Disk-based" (TDD) technique. The TDD technique includes the suffix tree construction algorithm (PWOTD), and an accompanying buffer management strategy.

Extensive experimental evaluations show that TDD scales gracefully as the dataset size increases. The TDD approach lets us build suffix trees on large frequently used sequence datasets such as UniProt/TrEMBL [5] in a few minutes. The TDD approach outperforms a popular disk-based suffix tree construction method (the Hunt's algorithm) by a factor of 5–10. In fact, to demonstrate the strength of TDD, we show that using slightly more main-memory than the input string, a suffix tree can be constructed on the *entire Human Genome in 30 hours on a single processor machine!* These input sizes are significantly larger than the datasets that have been used in previously published approaches.

In this paper, we also compare TDD with a recently proposed disk-based suffix array construction method [18], and show that TDD also outperforms this method.

Even though TDD far outperforms the existing suffix tree construction algorithms, TDD degrades in performance when the input data string is much larger than the amount of main memory. To address this case, we have also proposed a new merge-based suffix tree algorithm called ST-Merge. The TDD algorithm can be seen as a special case of the ST-Merge algorithm when the number of merge partitions is equal to one. We have implemented ST-Merge, and demonstrated its benefits over TDD for constructing suffix trees on very large string datasets.

As part of our future work, we plan on making ST-Merge and TDD more amenable to parallel execution. We believe that these algorithms are extremely parallelizable due to the partitioning phase that they employ. We are also exploring the benefits of using multiple disks, and of overlapping I/O and computation.

Acknowledgements This work was completed while Richard A. Hankins was a Ph.D. student at the University of Michigan. This research was supported by the National Science Foundation under grant IIS-0093059, and by a research gift donation from Microsoft. We would like to thank the reviewers of this paper for their careful study of our previous manuscript, and their constructive suggestions. Their comments have helped strengthen this paper. We would like to thank Srikanta Bedathur and Jayant Haritsa for providing us a copy of their TOP-Q code. We thank Stefan Kurtz for providing us with an implementation of McCreight's algorithm. We also thank Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders for providing us the source code for their DC3 algorithm. In addition, we also thank Giovanni Manzini and Paolo Ferragina for providing us the source code of their Deep-Shallow algorithm.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms **2**, 53–86 (2004)
2. Aluru, S.: Suffix Trees and Suffix Arrays, Handbook of Data Structures and Applications. CRC Press (2004)
3. Andersson, A., Nilsson, S.: Efficient implementation of suffix trees. Software: Pract Exp. **25**(2), 129–141 (1995)
4. Apostolico, A., Szpankowski, W.: Self-alignments in words and their applications. J. Algorithms **13**(3), 446–467 (1992)
5. Apweiler, R., Bairoch, A., Wu, C.H., Barker, W., Boeckmann, B., Ferro, S., Gasteiger, E., Huang, H., Lopez, R., Magrane, M., Martin, M.J., Natale, D., O'Donovan, A.C., Redaschi, N., Yeh, L.L.: Uniprot: The universal protein knowledgebase. Nucl. Acids Res. **32**(D), 115–119 (2004)
6. Atkinson, M., Jordan, M.: Providing orthogonal persistence for java. In Proceedings of the 12th European Conference on Object-Oriented Programming, pp. 383–395 (1998)
7. Bedathur, S.J., Haritsa, J.R.: Engineering a fast online persistent suffix tree construction. In: Proceedings of the 20th International Conference on Data Engineering, pp. 720–731 (2004)
8. Bentley, J.L., Sedgewick, R.: Fast algorithms for sorting and searching strings. In Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 360–369 (1997)
9. Blumer, A., Ehrenfeucht, A., Haussler, D.: Average sizes of suffix trees and DAWGs. Discrete Applied Mathematics **24**(1), 37–45 (1989)
10. Carvalho, A., Freitas, A., Oliveira, A., Sagot, M.-F.: A parallel algorithm for the extraction of structured motifs. In Proceedings of the 2004 ACM Symposium on Applied Computing, pp. 147–153 (2004)
11. Cheng, L.-L., Cheung, D., Yiu, S.-M.: Approximate string matching in DNA sequences. In: Proceeings of the 8th International Conference on Database Systems for Advanced Applications, pp. 303–310 (2003)
12. Cheung, C.-F., Yu, J.X., Lu, H.: Constructing suffix tree for gigabyte sequences with megabyte memory. IEEE Transactions on Knowledge and Data Engineering **17**(1), 90–105 (2005)
13. Clifford, R., Sergot, M.J.: Distributed and paged suffix trees for large genetic databases. In: Proceedings of 14th Annual Symposium on Combinatorial Pattern Matching, pp. 70–82, 2003.
14. Crauser, A., Ferragina, P.: A theoretical and experimental study on the construction of suffix arrays in external memory and its applications. Algorithmica **32**(1), 1–35 (2002)
15. Deep-Shallow Suffix Array and BWT Construction Algorithms. http://www.mfn.unipmn.it/∼manzini/lightweight/.
16. Delcher, A., Kasif, S., Fleischmann, R., Peterson, J., White, O., Salzberg, S.: Alignment of whole genomes. Nucleic Acids Res. **27**(11), 2369–2376 (1999)
17. Delcher, A., Phillippy, A., Carlton, J., Salzberg, S.: Fast algorithms for large-scale genome alignment and comparision. Nucleic Acids Res. **30**(11), 2478–2483 (2002)
18. Dementiev, R., Kärkkäinen, J., Mehnert, J., Sanders, P.: Better external memory suffix array construction. In: Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (2005)
19. Devroye, L., Szpankowski, W., Rais, B.: A note on the height of suffix trees. SIAM J. Comput. **21**(1), 48–53 (1992)
20. External Memory Suffix Array Construction Project. http://i10www.ira.uka.de/dementiev/esuffix/docu/index.html.
21. Farach, M.: Optimal suffix tree construction with large alphabets. In: Proceedings of the 38th Annual Symposium on Foundations of Computer Science, pp. 137–143. IEEE Computer Society (1997)
22. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. Journal of The ACM **47**(6), 987–1011 (2000)
23. GenBank, NCBI, 2004. http://www.ncbi.nlm.nih.gov/GenBank.
24. Giegerich, R., Kurtz, S.: From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. Algorithmica **19**(3), 331–353 (1997)
25. Giegerich, R., Kurtz, S., Stoye, J.: Efficient implementation of lazy suffix trees. Soft. Pract. Exp. **33**(11), 1035–1049 (2003)
26. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press (1997)
27. Heumann, K., Mewes, H.-W.: The hashed position tree (HPT): A suffix tree variant for large data sets stored on slow mass storage devices. In: Proceedings of the 3rd South American Workshop on String Processing, pp. 101–115 (1996)

28. Hunt, E., Atkinson, M.P., Irving, R.W.: A database index to large biological sequences. The VLDB J. **7**(3), 139–148 (2001)
29. Intel Corporation. The IA-32 Intel Architecture Optimization Reference Manual. Intel (Order Number 248966) (2004)
30. Intel Corporation. The IA-32 Intel Architecture Software Developer's Manual: System Programming Guide, vol. 3. Intel (Order Number 253668) (2004)
31. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Proceedings of the 13th International Conference on Automata, Languages and Programming, pp. 943–955 (2003)
32. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, pp. 181–192 (2001)
33. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, pp. 186–199 (June 2003)
34. Ko, P., Aluru, S.: Space efficient linear-time construction of suffix arrays. In: Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, pp. 200–210 (June 2003)
35. Kurtz, S.: Reducing space requirement of suffix trees. Soft: Pract. Exp. **29**(13), 1149–1171 (1999)
36. Kurtz, S., Choudhuri, J.V., Ohlebusch, E., Schleiermacher, C., Stoye, J., Giegerich, R.: REPuter: The manifold applications of repeat analysis on a genomic scale. Nucleic Acids Res. **29**, 4633–4642 (2001)
37. Kurtz, S., Phillippy, A., Delcher, A., Smoot, M., Shumway, M., Antonescu, C., Salzberg, S.: Versatile and open software for comparing large genomes. Genome Bio. **5**(R12) (2004)
38. Manzini, G.: Two space saving tricks for linear time LCP array computation. In Proceedings of the 9th Scandinavian Workshop on Algorithm Theory, pp. 372–383 (2004)
39. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica **40**(1), 33–50 (2004)
40. McCreight, E.M.: A space-economical suffix tree construction algorithm. Journal of The ACM **23**(2), 262–272 (1976)
41. Meek, C., Patel, J.M., Kasetty, S.: Oasis: An online and accurate technique for local-alignment searches on biological sequences. In Proceedings of 29th International Conference on Very Large Data Bases, pp. 910–921 (2003)
42. Navarro, G., Baeza-Yates, R., Tariho, J.: Indexing methods for approximate string matching. IEEE Data Eng. Bull. **24**(4), 19–27 (2001)
43. Patel, J.M.: The role of declarative querying in bioinformatics. OMICS: J. Integr. Biol. **7**(1), 89–92 (2003)
44. Pettersson, M.: Perfctr: Linux performance montioring counters driver. http://user.it.uu.se/∼mikpe/linux/perfctr.
45. Project Gutenberg. http://www.gutenberg.net.
46. Schurmann, K.-B., Stoye, J.: Suffix-tree construction and storage with limited main memory. Technical Report 2003-06, Univeristy of Bielefeld, Germany (2003)
47. STXXL Library. http://i10www.ira.uka.de/dementiev/stxxl.shtml.
48. Szpankowski, W.: Average-Case Analysis of Algorithms on Sequences. John Wiley and Sons (2001)
49. Tata, S., Hankins, R.A., Patel, J.M.: Practical suffix tree construction. In: Proceedings of 30th International Conference on Very Large Data Bases, pp. 36–47 (2004)
50. The Growth of GenBank, NCBI, 2004. http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html.
51. The MUMmer Software. http://www.tigr.org/software/mummer/.
52. Ukkonen, E.: Constructing suffix-trees on-line in linear time. In Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture: Information Processing, pp. 484–492 (1992)
53. Vitter, J.S., Shriver, M.: Algorithms for parallel memory: Two-level memories. Algorithmica **12**, 110–147 (1994)
54. Weiner, P.: Linear pattern matching algorithms. In Proceedings of the 14th Annual Symposium on Switching and Automata Theory, pp. 1–11 (1973)
55. Yona, S., Tsadok, D.: ANSI C implementation of a suffix tree. http://cs.haifa.ac.il/∼shlomo/suffix_tree.