When Rebuilding Suffix Arrays is Faster than Adding and Removing from Dynamic Suffix Arrays

Gail Carmichael

December 20, 2009

1 Introduction

Indexing structures for textual data has become important for supporting fast searches of large databases. Suffix trees [8] were once the structure of choice for their linear time construction, despite the large space requirements. In fact, much attention has been paid to disk-based suffix trees, designed particularly for use with DNA sequence databases (e.g. [9]).

Suffix arrays [7] gained interest when introduced in the early nineties for their smaller space consumption, but weren't widely adopted until several linear-time construction algorithms were later published almost simultaneously [4, 5, 6]. A disadvantage of suffix arrays over suffix trees is that they have traditionally been static, and so must be recreated from scratch when the text they are built on changes. However, a fully dynamic suffix array that supports adding and removing characters in any position has recently been proposed by Salson et al [10]. The dynamic suffix array relies on the similarities between the suffix array and the Burrows-Wheeler Transform [1], and is based on a method designed to maintain a dynamic Burrows-Wheeler Transform [11]. It supports the longest common prefix array used in many algorithms that operate on suffix arrays, but also requires other supplemental structures to support dynamic maintenance.

The dynamic suffix array was experimentally shown to be efficient, especially for few insertions of a large number of characters at once. These experiments give some sense of when it is faster to opt for rebuilding a standard suffix array instead of deleting and inserting characters from the dynamic version.

This paper summarizes results from a more detailed set of experiments that are designed to provide a better idea of when to use static suffix arrays instead of dynamic. Background information on static and dynamic suffix arrays can be found in Section 2, and the experiment results follow in Section 3. The results are summarized in Section 4 along with suggestions for further experimentation.

0	1	2	3	4	5	6	7	8	9
B	A	N	A	N	A	R	A	M	A

		p	lcp
0	A	9	0
1	AMA	7	1
2	ANANARAMA	1	1
3	ANARAMA	3	3
4	ARAMA	5	1
5	BANANARAMA	0	0
6	MA	8	0
7	NANARAMA	2	0
8	NARAMA	4	2
9	RAMA	6	0

Figure 1: An example of a suffix array for the text BANANARAMA. The position array is p and the longest common prefixes are in lcp. The first column gives the indices into the p and lcp arrays. The second column gives the suffix referred to by p.

2 Background

A suffix array is, conceptually, a structure that stores all of the suffixes of a body of text in sorted order. A suffix includes all of the characters found at and after a given index; it does not refer to the suffix of a single word. A list of indices referring to the starting points of the suffixes – also known as the position array – is stored in the suffix array so that the suffixes appear in the list in lexicographically sorted order. To search for a substring of the original text is now a matter of searching the suffixes using a binary search strategy to see if the query string appears as a prefix of one of the suffixes.

There are additional lists that can supplement the list of suffix positions to support certain operations more efficiently. The most commonly used is a list of longest common prefixes (or lcp's). The lcp for each suffix is the number of consecutive characters, starting at the beginning, it shares with the prefix appearing directly before it in lexicographical order. This information can be used, for instance, to avoid comparing common prefixes during binary searches.

More formally, let the position array for a text T if size n be p, and the longest common prefixes be stored in lcp. Then $p[j] = i$ if and only if $T[i...n]$ is the $(j+1)st$ suffix of T in sorted order. Also, the length of the longest common prefix between $p[i]$ and $p[i-1]$ is given by $lcp[i]$. An example of a suffix array is shown in Figure 1.

As mentioned in the introduction, one reason that suffix arrays were not chosen over suffix trees after they were proposed, despite their relatively low storage requirements, was the fact that they took longer to build. Suffix trees require $\Theta(n \log n)$ time for construction when the text it is built for has a general alphabet that only allows for symbol comparisons [3], but can be made in linear time for integer alphabets (where symbols are integers in the range

$[0, n^c]$ for a constant c) or alphabets of constant size [2, 3, 8, 12]. One of the first solutions for constructing a suffix array for an integer alphabet in linear time takes advantage of a divide-and-conquer approach to the problem, proposing a new merging algorithm [5].

Although this improvement in construction time makes the use of suffix arrays more practical, they still have a disadvantage over suffix trees that makes them still less attractive for some applications. The typical suffix array construction produces a structure which is not dynamic; that is, once built, the text T cannot change without having to rebuild p and lcp from scratch. A new strategy was recently published, however, that does allow for insertions and deletions of characters into the suffix array [10]. The trade-off is the need for additional supporting structures, though some of the structures used can potentially be compressed to save space.

These dynamic suffix arrays make use of an efficient four-stage algorithm for a dynamic Burrows-Wheeler transform [11]. By maintaining the Burrows-Wheeler transform of text T, the position array p and longest common prefixes in lcp can be kept up to date as well. Because of the similarities between the transform and the suffix array, each of the four stages can also be used to determine what changes, if any, are required for p and lcp. The structures needed to maintain all the needed information results in a space complexity of more than $12n$, but sampling techniques can be used to reduce this.

Experimental results [10] showed that this implementation of dynamic suffix arrays is often more efficient than rebuilding a static suffix array from scratch. The tests performed involved adding 500 characters in differing amounts to a dynamic suffix array already containing a certain amount of text, and comparing the time results to the time taken to rebuilding a suffix array on all of the text (both the text being added to and the characters being added). The only time that adding to the dynamic suffix array was slower was when 500 characters were added one at a time rather than in larger groups, and only up until the original text was 500 kilobytes or more for text containing DNA sequences, or 1400 kilobytes for random text on an alphabet of size 100. The most efficient scenario was adding all 500 characters at once.

While these results are useful, some more detailed experiments would shed more light on when exactly it is better to rebuild a static array instead of maintaining a dynamic one. Several such experiments are presented in the next section.

3 Experiments and Results

The goal of the following experiments is to help determine when it is preferable to choose a static suffix array and when to opt for the dynamic version. The experimental setup will be explained first, followed by results from several tests.

All experiments in this section were performed on a Ubuntu Linux distribution running in a virtual machine assigned two 2.4 GHz cores of an Intel Core 2 Quad CPU, and 2096 MB of dedicated memory. Running times were measured using the `gettimeofday` function in C. The implementation for constructing the static suffix array was written by Karkkainen

and Sanders[1] and the corresponding computation of the longest common prefix is an implementation of Kärkkäinen et al's permuted suffix array [4] written by Pat Morin[2]. The implementation of the dynamic suffix arrays was kindly provided by its authors, Salson et al. The texts considered here are two DNA sequences, one 975 KB and the other 50 MB.

The main premise of these experiments is to shift a sliding window over a large body of text. A window of size w and a slide amount $s \leq w$ are chosen. The window starts at the beginning of the text and slides a maximum of 100 times. If the current index of the beginning of the window is i, then the next starting index after a slide is $j = i + s$. A window may not be able to slide all 100 times, depending on the size of the text and the size of the window. Only the contents of the window will be in the suffix array at any given time, and as the window slides, the contents are updated either by rebuilding the suffix array (static case) or inserting and deleting the appropriate characters from it (dynamic case). This strategy fits most intuitively with an application that, say, needs to load only certain portions of a large disk-based body of text into main memory, but should also provide general insight to the main question at hand.

The experiments by Salson et al presented only one particular usage scenario for their dynamic suffix arrays: insertion of 500 characters into a large body of text. They varied the size of text being inserted into, affecting the reordering stage of their algorithm as well as the rebuilding of the static suffix array being compared to. Their results showed that insertions of one character at a time can be slower on smaller texts than rebuilding the suffix array, but that multiple insertions of ten or more characters at a time was always faster than rebuilding. These tests did not consider deletions, nor the time taken to build the intial dynamic suffix array before insertions. The following sliding window tests will help fill this gap.

The first results for the 975 KB text are shown in Figure 2. The time shown in milliseconds on the logarithmic y-axis denotes the average time taken per window slide. This includes either rebuilding the static array, or deleting and adding characters that are leaving or entering the window. When adding or removing characters, all are inserted or deleted together, since it was already shown that this is more efficient. The time taken for the initial build is spread across the number of slides when it is included at all.

Figure 2a shows the time taken per window slide for $s = 1$ to 1001 in increments of 100. Two relatively large window sizes are considered – $w = 600,000$ and $w = 700,000$ – since Salson et al's tests showed that insertion into dynamic suffix arrays is more efficient than rebuilding in all cases when the size of text being inserted into is large. In this case, the time taken to build the initial suffix array is included. The dynamic suffix array is built on the first window and so will be size w. When this building time is included in the average time per slide, the dynamic suffix array performs consistently worse than simply rebuilding static suffix arrays of size w each time.

The initial build time is not included in the results of Figure 2b. The dynamic suffix array performs much better than the static version when the value of s is low, indicating that the overhead for building the first dynamic suffix array is significant. As s reaches 1000, however,

[1] Available at http://www.mpi-inf.mpg.de/~sanders/programs/suffix/drittel.C

[2] http://cg.scs.carleton.ca/~morin/

adding and deleting catches up to simply rebuilding. Salson et al's tests considered insertions of up to 500 characters only, and the sliding window results for inserting that number of characters agrees with their results. Is appears, however, that insertion and deletion of too many characters for these window sizes incurs enough overhead to consider using a static suffix array instead.

Figure 3 shows results for the same 975 KB text for larger values of s but similar window sizes. Whether including the building time (Figure 3a) or not (Figure 3b), the dynamic suffix array is significantly slower than rebuilding the static suffix array. This agrees with the trend started in the previous results, which suggested that inserting and deleting more than 1000 characters at a time would be slower than rebuilding for these window sizes.

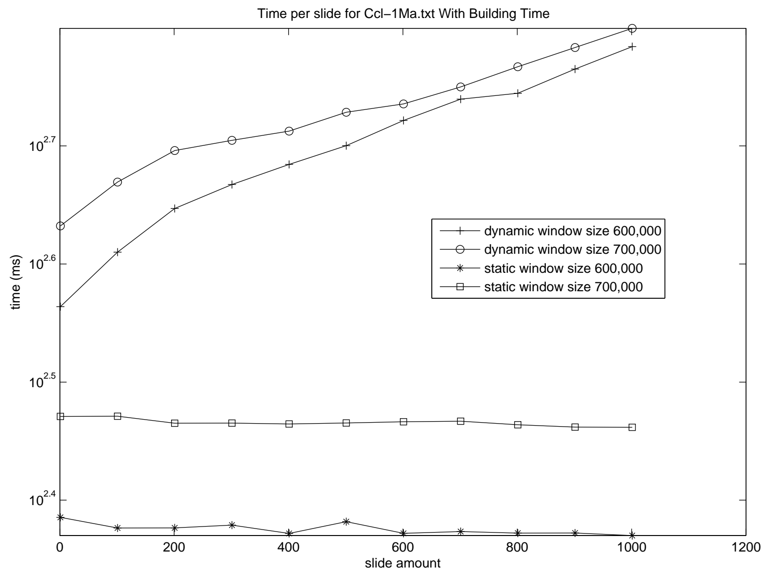
Results for the 50 MB DNA sequence for slide amounts $s = 1$ to 1001 are in Figure 4. These tests used a much larger window size of $w = 1000000$, made possible because of the larger text size. When w gets large, it is expected that rebuilding an entire suffix array that size will become very slow, making the dynamic suffix arrays a better choice. Based on these results and the previous ones, it turns out that when including the initial building time for a dynamic suffix array, rebuilding the static version is *still* a better choice. And, as before, when the slide amount s reaches about 1000, the dynamic suffix array starts to become slower even when not considering its initial build time. Figure 5 shows that for very large slide amounts the dynamic suffix array is much slower no matter whether build time is included.

4 Conclusion

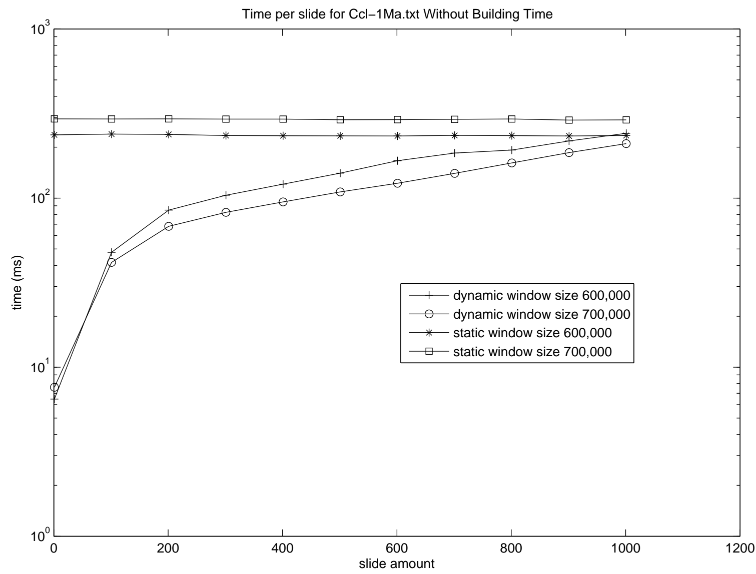
The authors of the dynamic suffix array showed that inserting 500 characters into a large body of text is more efficient with their structure than rebuilding a standard suffix array from scratch. They did not, however, include deletions in their tests, nor did they consider the time taken to build the dynamic suffix array before the insertions.

The experiments presented here used a sliding window paradigm to fill both these gaps. It was determined that if the building time is included (in this case, averaged over the number of window slides), the use of dynamic suffix arrays tends to be slower than rebuilding standard suffix arrays from scratch. However, dynamic suffix arrays are superior to the static version when inserting and deleting less than 1000 characters at a time. Based on this, dynamic suffix arrays are well suited to applications that make only small changes to the associated body of text, while static suffix arrays are a better choice when large changes are required often.

Several more experiments are required in the future to verify these results and provide more insight into the problem. In particular, texts other than DNA sequences should be examined since the results from Salson et al did show a difference between the two types. Even larger window sizes should be examined to see if the 1000 character limit remains, and further analysis on the initial building time of the dynamic suffix arrays would be beneficial.

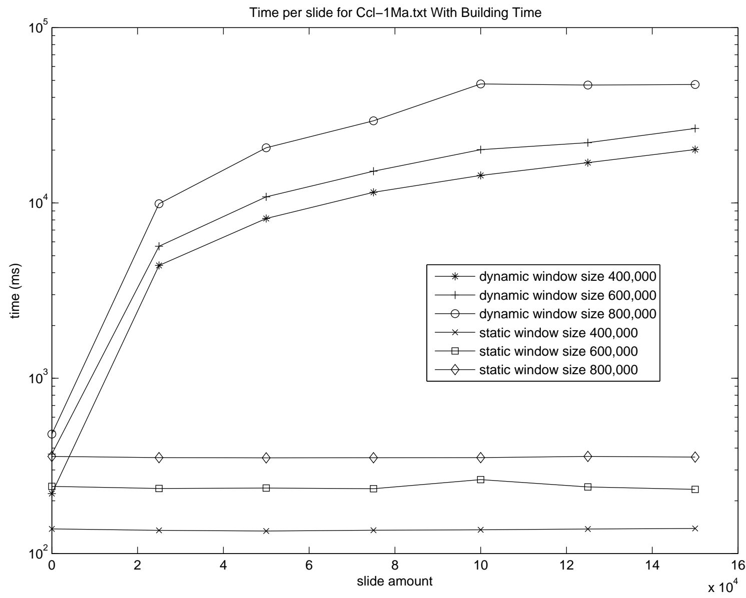


(a)

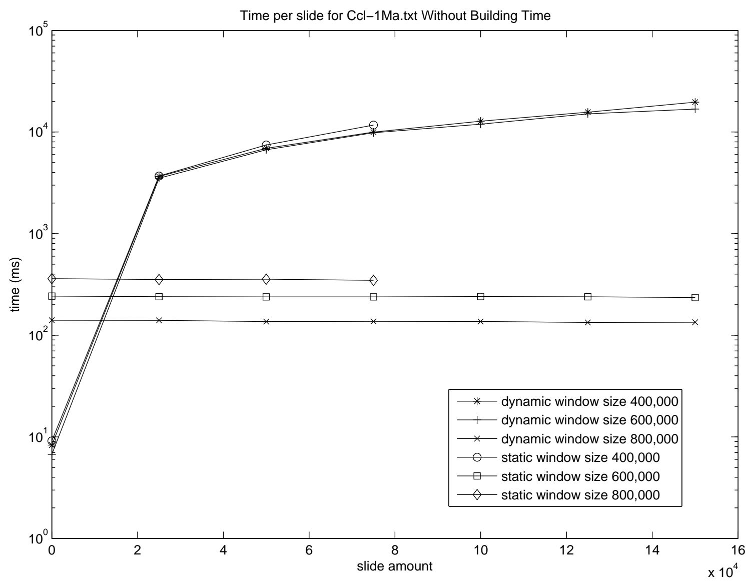


(b)

Figure 2: Average time spent on each slide for 975 KB DNA sequence for slide amounts 1-1001.



(a)



(b)

Figure 3: Average time spent on each slide for 975 KB DNA sequence for slide amounts 1-150001.

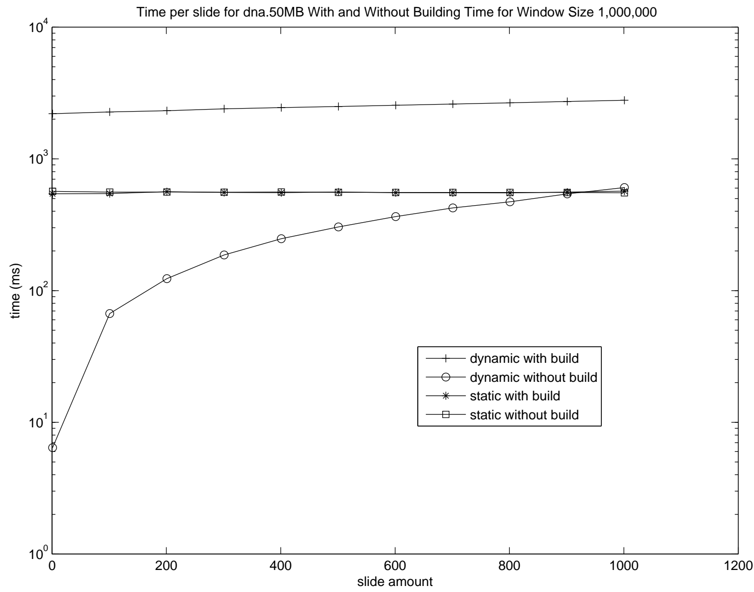


Figure 4: Average time spent on each slide for 50 MB DNA sequence for slide amounts 1-1001.

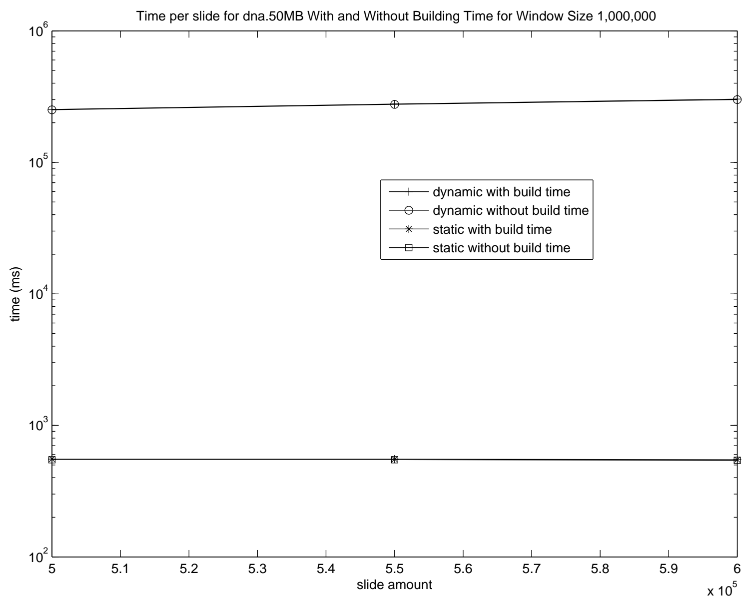


Figure 5: Average time spent on each slide for 50 MB DNA sequence for slide amounts 500000, 600000, and 700000.

References

[1] Burrows, M., and Wheeler, D. J. A block-sorting lossless data compression algorithm. Tech. rep., 1994.

[2] Farach, M. Optimal suffix tree construction with large alphabets. In *FOCS '97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1997), IEEE Computer Society, p. 137.

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

[4] Kärkkäinen, J., Sanders, P., and Burkhardt, S. Linear work suffix array construction. *J. ACM 53*, 6 (2006), 918–936.

[5] Kim, D. K., Sim, J. S., Park, H., and Park, K. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms 3*, 2-4 (June 2005), 126–142.

[6] Ko, P., and Aluru, S. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms 3*, 2-4 (June 2005), 143–156.

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

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

[9] Phoophakdee, B., and Zaki, M. J. Genome-scale disk-based suffix tree indexing. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2007), ACM Press, pp. 833–844.

[10] Salson, M., Lecroq, T., Léonard, M., and Mouchard, L. Dynamic extended suffix arrays. *Journal of Discrete Algorithms* (March 2009).

[11] Salson, M., Lecroq, T., Léonard, M., and Mouchard, L. A four-stage algorithm for updating a burrows–wheeler transform. *Theoretical Computer Science 410*, 43 (October 2009), 4350–4359.

[12] Ukkonen, E. On-line construction of suffix trees. *Algorithmica 14*, 3 (September 2005), 249–260.