1 Introduction

Sorting all the suffixes of a string $x = x[1..n]$ into lexicographical order is the most computationally expensive step in the Burrows-Wheeler Transformation for lossless compression [2]. One tool for achieving a suffix sort is the suffix array [12]. Suffix arrays are also the basis for a variety of compressed text indexing systems that allow arbitrary queries, for example the scheme of Grossi and Vitter [4].

Over the past decade, suffix array construction has been the focus of intensive research, and the many algorithms developed have been of two main types: direct comparison algorithms and doubling algorithms [16]. Direct comparison algorithms, in their simplest form, sort suffixes one character at a time as quicksort does. This technique was applied by Burrows and Wheeler [2], and has since been significantly improved by Seward [16] and Manzini and Ferragina [13]. Direct comparison algorithms are $O(n^2 \log n)$ in the worst case, but tend to be very fast for real inputs. Doubling algorithms on the other hand, with each sorting pass, double the depth to which the suffixes are sorted. In this way, the suffixes are sorted in a logarithmic number of passes, giving an overall worst-case time bound of $O(n \log n)$ assuming a linear sort, such as radix sort, can be used for each pass. Manber and Myers [12] were the first to apply this idea to suffix sorting and their approach was later significantly improved in Algorithm LS by Larsson and Sadakane [10, 15].

In 2003 the situation changed dramatically, with the discovery of three different algorithms requiring only $\Theta(n)$ time in the worst case to compute the suffix array of a given string $x = x[1..n]$ on an indexed (integer) alphabet [5, 6, 7]. Each of these algorithms was based on the approach used by Farach [3] to construct the suffix tree of a string on an indexed alphabet: namely separate the suffixes into two groups and use a recursive ordering

---

*Supported in part by grants from the Natural Sciences and Engineering Research Council of Canada.
of one group to determine a correct ordering of all suffixes. Of the three new algorithms, one was of theoretical rather than practical interest [6], but the other two, due to Ko-Aluru (Algorithm KA) and Kärkkäinen-Sanders (Algorithm KS), were implemented by Lee and Park [11] to reduce both time and space requirements.

However, to our knowledge, there has been no adequate assessment of the practical performance of the new algorithms. This paper addresses that need, by comparing running times and space usage of the linear approaches to that of other leading supralinear suffix sorting programs. Further, we describe and implement several optimizations to improve the performance of Algorithm KS. While we treat only KS, the same improvements could be readily applied to the other linear time algorithms.

The only two previous studies of which we are aware present ambiguous results. Lee and Park [11] compare implementations of KA and KS with the doubling algorithm of Manber and Myers (MM). According to their experiments on a wide range of strings and varying alphabet sizes, it was generally true that

$$t_{KA} < t_{KS} < t_{MM},$$

where $$t_X$$ denotes the time required to run Algorithm X.

On the other hand, Antonitio et al. [1] performed experiments with the original implementation of Algorithm KS as it appeared in [5] and found

$$t_{LS} < t_{KS},$$

leaving open the question of whether Algorithm KA would perform faster than Algorithm LS in practice. In this paper we present experimental results suggesting that Algorithm KA as implemented by Lee and Park, even though linear in the worst case, is in practice not as fast as several other supralinear suffix array construction algorithms.

Section 2 gives an overview of Algorithm KS, our improvements to which we describe in Section 3. Then in Section 4 we present our experimental results, comparing the new linear time approaches to other leading suffix sorting programs. Section 5 discusses the significance of these results. In Section 6 we offer conclusions and put forward ideas about how further progress with suffix array construction algorithms might be made.

## 2 Algorithm KS

In this section we describe the linear time suffix array construction algorithm of Kärkkäinen and Sanders using the string in Figure 1 as a working example.

---

**Figure 1:** An example string $$x$$ and the final suffix array of the string $$\sigma_x$$.
Algorithm KS uses the classic divide-and-conquer approach to suffix array construction. It splits the original string into two parts: a sorted list \( C \) of characters of length \( n/3 \) and a sorted list \( S \) of suffixes of length \( 2n/3 \). In particular, the characters are every third character in the string, and the suffixes are all the remaining positions. In our example string of 24 characters, the lists are

\[
C = \{\$, \text{a, i, i, l, l, s, s, t}\}
\]

\[
S = \{s_{13}, s_{4}, s_{16}, s_{22}, s_{5}, s_{11}, s_{2}, s_{10}, s_{1}, s_{7}, s_{8}, s_{23}, s_{14}, s_{19}, s_{20}, s_{17}\}
\]

where \( s_i \) is a suffix beginning at position \( i \).

These two lists can be merged in linear time to create a full ordering on all suffixes of the string—the suffix array \( \sigma_x \). The merge algorithm is a variant of the basic linear merge algorithm for two lists, which maintains a pointer \( j \) into the character list, and a pointer \( k \) into the suffix list. If \( C[j] < S[k] \) then \( s_j < s_k \) and \( j \) increases, and vice versa. If \( C[j] = S[k] \) then a comparison between \( S[j+1] \) and either \( S[k+1] \) or \( C[k+1] \), depending on which exists, can be used to break the tie. If \( C[j] = S[k] \) and \( S[j+1] = C[k+1] \), then the tie can definitely be broken by examining \( S[j+2] \) and \( S[k+2] \). At most three comparisons are required for each merge step.

The first list \( C \) can be generated in linear time on an indexed alphabet using a counting sort. The second list \( S \) is generated by recursively finding the suffix array of a modified string of juxtaposed, re-labelled triples of characters. The modified string is developed by first sorting all triples of characters beginning at positions \( i \neq 0 \pmod{3} \) and then relabelling them with an ordinal character. Figure 2 shows the triples in our example sorted lexicographically; \( t_i \) represents the first three characters of \( s_i \). The final row in the figure shows the triples relabelled with A through O.

The new string is formed by juxtaposing the new label of triples \( t_i, i = 1 \pmod{3} \) and then \( t_i, i = 2 \pmod{3} \). In the example, the juxtaposition yields

\[
H \ B \ I \ H \ A \ C \ M \ D \quad and \quad G \ E \ J \ F \ L \ O \ N \ K
\]
to form the new string \( x' = \text{HBIHACMDGEJFLONK}$$. Now we find the suffix array of
the new string of ordinal positions, which is shown in Figure 3. The suffix array for this
modified string not only gives us an ordering on the triples, as shown in the last row of
Figure 3, it also gives us an ordering on the suffixes beginning with those triples.

In summary, Algorithm KS proceeds as follows on a string \( x \).

1) Set \( C \leftarrow \{x[i], i = 0 \pmod{3}\} \)
2) Set \( T \leftarrow \{(x[i] \ldots x[i+2]), i \neq 0 \pmod{3}\} \)
3) Counting Sort \( C \) and Radix Sort \( T \)
4) Relabel each triple \( t_j \in T \) with an ordinal rank \( r_i \)
5) Form \( x' = \{r_i | i \mod 3 = 1\}\{r_i | i \mod 3 = 2\} \)
6) Derive \( S \) from the suffix array \( \sigma_{x'} \) for the string \( x' \)
7) Merge \( C \) and \( S \) to form the suffix array \( \sigma_x \)

All steps are linear in the length of the string, except for Step 6 which can be imple-
mented with a recursive call to the same algorithm on an input of size \( \lceil 2n/3 \rceil \), and so the
total running time is \( O(n) \).

3 Engineering Algorithm KS

Here we describe the techniques we employed to improve the practical performance of
Algorithm KS. Although we only treat KS here, the variations described could equally be
applied to Algorithm KA.

3.1 Shortening the recursion string

This optimization is based on the observation that the recursion string \( x' \) contains unique
characters and duplicate characters, being ranks of triples that were unique and ranks of
triples that were not unique respectively. Only the ordering of the duplicate characters are
of interest as they represent the suffixes that could not be sorted based on their leading three
caracters.

It is possible to shorten the recursion string while still maintaining the relative order
of the duplicate characters, by removing all but the first unique character in every run of
unique characters in \( x' \). More precisely, to form the shortened string \( \hat{x}' \): for each subword
\( u = x'[i..j] \) such that \( x'[i]..x'[j] \) are all unique characters but \( x'[i-1] \) and \( x'[j+1] \) are not, remove \( \hat{x}'[i+1..j] \).

In the example above, \( x' = \text{HBIHACMDGEJFLONK}$ would become \( \hat{x}' = \text{HBHAS}$
as H is the only duplicate character. Shortening the string in this way reduces the work
required at the next level of recursion, in this case dramatically.

It is easy to see that suffixes starting with duplicate characters in \( \hat{x}' \) have the same
relative order as they did in \( x' \). To distinguish two suffixes of \( x' \), say \( s'_i \) and \( s'_j \), that are equal
in the first \( k \) positions, it suffices to compare \( x'[i+k] \) and \( x'[j+k] \) to break the tie between
the two. If these two characters are unique, then it is guaranteed that \( x'[i+k] \neq x'[j+k] \),
and the tie can be broken. If there are more unique characters in positions following either
of these two characters, they play no role in breaking the tie of \( s_i' \) and \( s_j' \), nor any of the suffixes of those two suffixes.

To form the shortened string \( \hat{x}' \), we first form \( x' \). A pass over \( x' \) calculates the length of \( \hat{x}' \), and with this knowledge we make a judgement whether computing the shortened string will be profitable. Knowing the length of \( \hat{x}' \) also allows us to reserve exact memory for \( \hat{x}' \) and an array \( V \) of indices into \( x' \) which have been retained in forming \( S' \). Storing \( V \) allows us to break ties in the ranks of \( x' \) once the suffix order of \( \hat{x}' \) is known.

It will not always be worthwhile shortening \( x' \). If \( \hat{x}' \) is roughly the same size as \( x' \), then the extra memory required to hold \( x' \), \( \hat{x}' \) and \( V \) will outweigh the small time saving gained at the next level of recursion. We know the length of \( \hat{x}' \) before it is formed so if it is too long, say \(|\hat{x}'|/|x'| < c\) for some predefined constant \( c \), we can simply revert to Algorithm KS for \( x' \).

### 3.2 Computing more unique characters at each level

The more unique characters we have after ranking triples to form \( x' \), the more likely that savings will accrue from the shortening step described in the previous section. To obtain more unique characters from the sorting phase at each level, we can radix sort on more than the first three characters of the elements of \( S \). Obviously the extra time spent sorting must be counteracted by the time saved in shortening \( x' \).

The ideal number of characters on which to sort will be dependent on the input string. That is, past a certain depth the cost of sorting deeper will outweigh the reduction in processing at the next level (see Section 6).

The simplistic LSD radix passes used to order triples in Lee and Park’s implementation of KS become prohibitively slow when sorting to greater depth. We replace that sorting routine with a version of American Flag Sort (AFS) [14] adapted to sort suffixes to a specified depth \( k \). AFS is a very efficient implementation of MSD radix sort, intended to sort strings with a byte radix. The use of MSD radix sort maintains the linearity of KS.

### 3.3 Folding naming with sorting

With the elements of \( T \), the list of triples, ordered by their first \( k \) characters, they can be given ordinal ranks in a single pass that compares the first \( k \) characters of adjacent elements—a total of \( \lceil 2n/3 \rceil \times k \) comparisons. In the original algorithm with \( k = 3 \) this step did not dominate the running time, but in our version \( k \) is typically larger than 3 as noted in the previous section. With increased \( k \), the naming stage becomes time consuming.

To save time, we incorporate naming of the tied triples into the sorting routine. Naming tied groups integrates naturally to the depth-limited radix sort and adds little overhead to the code.

### 3.4 Making careful use of memory

Algorithm KS does not form any part of the final suffix array until the very end of the algorithm when the \( C \) and \( S \) arrays are merged. This means that the 4n bytes of memory which will eventually hold \( \sigma_x \) can be used as working space for the rest of the algorithm.
Table 1: Files from the Canterbury Corpus (CC) and the Manzini and Ferragina corpus (MF) used for testing. LCP refers to the Longest Common Prefix amongst all suffixes in the string. $|\Sigma|$ is the number of distinct characters in the string.

| String          | Source | Size (bytes) | $|\Sigma|$ | Mean LCP | Max LCP |
|-----------------|--------|--------------|-----------|----------|---------|
| world192.txt    | CC     | 2,473,400    | 94        | 23.0     | 559     |
| bible.txt       | CC     | 4,047,392    | 63        | 13.9     | 551     |
| E.coli          | CC     | 4,638,690    | 4         | 17.1     | 2,815   |
| chr22.dna       | MF     | 34,553,758   | 4         | 1,972.7  | 199,999 |
| howto           | MF     | 39,422,105   | 197       | 253.2    | 70,720  |
| etext99         | MF     | 105,277,340  | 146       | 1,073.1  | 286,352 |

Some other structures in the working of the algorithm are also mutually exclusive, and so can share space.

4 Experimental Results

Using implementations of KA and KS from Lee and Park [11] and our variant KS', performance in terms of run time and peak memory usage for typical inputs compared with other leading suffix sorting algorithms was analyzed. In addition to the linear implementations, we tested the $O(n \log n)$ algorithm of Larsson and Sadakane (Algorithm LS); the so-called deep-shallow suffix sorter of Manzini and Ferragina [13] (Algorithm MF) which is $O(n^2 \log n)$ in the worst case; and the difference-cover algorithm of Kärkkäinen and Burkhardt (Algorithm KB) [8] with $O(n \log n)$ asymptotic runtime. All algorithms have source code available online. For completeness we also tested Stefan Kurtz’s memory efficient suffix tree implementation (Algorithm K), which requires $O(|\Sigma|n)$ time, where $\Sigma$ is the alphabet of the string [9].

KS' sorts to a depth of $k = 54$ (as described in Section 3). This value was arrived at experimentally, and we are currently investigating having the algorithm choose $k$ adaptively.

Table 2: Files of synthetic data used for testing

| File     | Repeated string | Size (bytes) | $|\Sigma|$ | Mean LCP | Max LCP |
|----------|-----------------|--------------|-----------|----------|---------|
| alla.txt | 'a'             | $5 \times 10^7$ | 1         | 24,999,999,999 | 49,999,999 |
| ra1.txt  | Random 20 chars | $5 \times 10^7$ | 15        | 24,999,980 | 49,999,980 |
| ra2.txt  | Random 1,000 chars | $5 \times 10^7$ | 26        | 24,999,000 | 49,999,000 |
| ra3.txt  | Random 500,000 chars | $5 \times 10^7$ | 26        | 24,502,500 | 49,500,000 |

Files from the Canterbury corpus\(^1\) and from the corpus compiled by Manzini\(^2\) and

\(^1\)http://www.cosc.canterbury.ac.nz/corpus/
\(^2\)http://www.mfn.unipmn.it/~manzini/lightweight/corpus/
Ferragina [13] listed in Table 1 were used for testing. To simulate difficult cases, we generated several files with large LCP values following the approach of Burkhardt and Kärkkäinen [8]. These synthetic files ra1.txt, ra2.txt, and ra3.txt are composed of repetitions of strings of 20, 1000, and 500000 random characters respectively.

All tests were conducted on a 2.8 GHz Intel Pentium 4 processor with 2Gb main memory. The operating system was RedHat Linux Fedora Core 1 (Yarrow) running kernel 2.4.23. The compiler was g++ (gcc version 3.3.2) with -O3 option. Running times, shown in Table 3, are the average of four runs and do not include time spent reading input files. Times were recorded with the standard unix time function. Memory usage, shown in Table 4, was recorded with the memusage command available with most Linux distributions.

<table>
<thead>
<tr>
<th>File</th>
<th>MF</th>
<th>KB</th>
<th>LS</th>
<th>K</th>
<th>KA</th>
<th>KS</th>
<th>KS'</th>
</tr>
</thead>
<tbody>
<tr>
<td>world192.txt</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>bible.txt</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>E.coli</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>chr22.dna</td>
<td>16</td>
<td>42</td>
<td>35</td>
<td>51</td>
<td>47</td>
<td>102</td>
<td>52</td>
</tr>
<tr>
<td>howto</td>
<td>18</td>
<td>45</td>
<td>40</td>
<td>80</td>
<td>63</td>
<td>128</td>
<td>55</td>
</tr>
<tr>
<td>etext99</td>
<td>76</td>
<td>149</td>
<td>146</td>
<td>233</td>
<td>202</td>
<td>400</td>
<td>219</td>
</tr>
<tr>
<td>alla.txt</td>
<td>1</td>
<td>61</td>
<td>15</td>
<td>9</td>
<td>3</td>
<td>15</td>
<td>24</td>
</tr>
<tr>
<td>ra1.txt</td>
<td>–</td>
<td>122</td>
<td>85</td>
<td>21</td>
<td>18</td>
<td>52</td>
<td>102</td>
</tr>
<tr>
<td>ra2.txt</td>
<td>–</td>
<td>241</td>
<td>155</td>
<td>41</td>
<td>60</td>
<td>113</td>
<td>275</td>
</tr>
<tr>
<td>ra3.txt</td>
<td>–</td>
<td>136</td>
<td>250</td>
<td>177</td>
<td>93</td>
<td>148</td>
<td>186</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File</th>
<th>MF</th>
<th>KB</th>
<th>LS</th>
<th>K</th>
<th>KA</th>
<th>KS</th>
<th>KS'</th>
</tr>
</thead>
<tbody>
<tr>
<td>world192.txt</td>
<td>12</td>
<td>14</td>
<td>19</td>
<td>32</td>
<td>31</td>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>bible.txt</td>
<td>19</td>
<td>23</td>
<td>31</td>
<td>54</td>
<td>50</td>
<td>52</td>
<td>45</td>
</tr>
<tr>
<td>E.coli</td>
<td>22</td>
<td>26</td>
<td>35</td>
<td>74</td>
<td>58</td>
<td>59</td>
<td>52</td>
</tr>
<tr>
<td>chr22.dna</td>
<td>165</td>
<td>194</td>
<td>264</td>
<td>541</td>
<td>429</td>
<td>438</td>
<td>401</td>
</tr>
<tr>
<td>howto</td>
<td>188</td>
<td>221</td>
<td>301</td>
<td>526</td>
<td>526</td>
<td>499</td>
<td>489</td>
</tr>
<tr>
<td>etext99</td>
<td>503</td>
<td>590</td>
<td>803</td>
<td>1,405</td>
<td>1,406</td>
<td>1,338</td>
<td>1,071</td>
</tr>
<tr>
<td>alla.txt</td>
<td>238</td>
<td>280</td>
<td>381</td>
<td>972</td>
<td>429</td>
<td>620</td>
<td>664</td>
</tr>
<tr>
<td>ra1.txt</td>
<td>–</td>
<td>280</td>
<td>381</td>
<td>791</td>
<td>620</td>
<td>620</td>
<td>664</td>
</tr>
<tr>
<td>ra2.txt</td>
<td>–</td>
<td>280</td>
<td>381</td>
<td>782</td>
<td>620</td>
<td>620</td>
<td>664</td>
</tr>
<tr>
<td>ra3.txt</td>
<td>–</td>
<td>280</td>
<td>381</td>
<td>782</td>
<td>620</td>
<td>620</td>
<td>664</td>
</tr>
</tbody>
</table>

3Due to the page limit, only a subset of the two corpora are used here.
5 Discussion

Results, summarised in Figure 4, show the linear time algorithms (KA, KS, KS') to be inferior to the non-linear approaches (MF, KB, LS) in terms of both runtime and memory usage for all tested inputs. Algorithm MF was fastest for all inputs, outperforming KB and LS by roughly a factor of 2 and is on average about 5 times faster than KA and KS. Algorithm MF also uses the least memory, with the linear algorithms requiring a hefty 300% more space on average.

Performances of the non-linear algorithms confirm results suggested by Burkhardt and Kärkkäinen’s limited testing [8]: MF is clearly the most efficient suffix sorting algorithm in practice in terms of both time and space. The performance of Algorithm MF is particularly impressive when one considers that the minimum space required to hold the input string together with the suffix array is 5n bytes (n bytes for the input and 4n bytes for the suffix array). Not only is MF the fastest algorithm, it comes very close to this minimum space requirement during operation, requiring on average just 0.014 extra bytes per input symbol for working space.

Turning to the linear time algorithms, KS' is a significant improvement on KS for the real-world data, achieving 50-67% faster running times, with less space. The implementation of KA tested [11] uses the sign bit to indicate whether suffixes are type-S or type-L [7]. This means that for files etext and howto, the memory usage for KA is misleadingly large, because with $|\Sigma| > 127$ each input symbol requires a 16-bit word (rather than 8-bit, even after recoding the alphabet), even though a significant portion of this larger word will never used. A more flexible implementation would use a separate bit vector, at the cost of a slight increase in memory. The larger memory usage of the KA and KS algorithms relative to the others reflects the difference in sorting schemes between the linear and supralinear
approaches. Radix sort (relied on by KS and KA to achieve linearity) requires the use of two extra arrays of combined size at least 5n bytes. Algorithms MF, KB and LS on the other hand all use multikey quicksort, meaning sorting is done inplace with a little extra for a small recursive stack in nearly all cases. But the difference in memory requirements is not solely due to the sorting requirements. The linear algorithms must construct a new string after sorting, requiring space for it and subsequently its suffix array. Compacting the recursion string as KS′ does, can reduce memory, but KS′ still requires more than twice the memory of MF.

Another way to ensure the recursion string is smaller would be to choose a smaller sample of suffixes. The linear merge of Algorithm KS between the list of characters $C$ and the sample of suffixes $S$ will work for any $(j+1)n/(2j+1)$ subset of suffixes (integer $j \geq 1$). Choosing, say, $4n/7$ suffixes rather than $2n/3$ suffixes will shorten the recursion string, though perhaps increasing the cost of the merge.

The good performance of the linear algorithms on the synthetic data is perhaps not surprising; these pathological inputs are particularly catastrophic for MF [8]. Running times for KS′ now fall behind KS, the price paid for the effort expended sorting to depth $k = 54$ at the top level, before falling back to normal KS with no reduction of the recursion string. A smaller value of $k$ here would certainly have led to a faster runtime, and gives reason to find a way to choose $k$ adaptively: by keeping track of the unique ranks gained at each level of sorting, the algorithm could decide to stop sorting early.

All of the methods use less memory than the suffix tree-based sort (Algorithm K). The linear-time algorithms are not significantly faster than the tree-based approach, but the supralinear algorithms are faster on the real data, though slower on the pathological data.

6 Conclusions and Future Work

In this paper we have illustrated that the superior asymptotic complexity of linear time suffix sorting algorithms does not readily translate into faster suffix sorting, compared to implementations of supralinear algorithms. We have also resolved the ambiguity surrounding the practicality of Algorithm KA: it is slower than supralinear approaches on real data.

We described several optimizations to the $O(n)$ KS algorithm that significantly improve performance for real world inputs, but still fall short of some supralinear approaches. It is worth noting that most of the optimizations we describe could also be applied to Algorithm KB, which may then outperform the well tuned suffix sorter of Manzini and Ferragina [13]. Source code for the improved KS variant tested in this paper, and the synthetic data used for testing, is available from the authors.

Manzini and Ferragina introduce the idea of lightweight suffix array construction algorithms as being those that use small space, less than 6n bytes say [13]. The question remains: is there a $\Theta(n)$ time suffix array construction algorithm which is also lightweight?

In [16], Seward suggests that all robust implementations of the BWT using direct comparison approach to suffix sorting (such as algorithm MF) would need to utilise Algorithm LS as a fall back routine when worst cases occur. Linear time suffix sorting routines may yet find a use in such a role, where their linear performance would come to the fore.
References


