LU-CS-TR:98-204 [LUNFD6/(NFCS-3134)/1–6/(1998)], Lund University, Sweden

NOTES ON SUFFIX SORTING

N. JESPER LARSSON

Abstract. We study the problem of lexicographically sorting the suffixes of a string of symbols. In particular, we analyze the time complexity of Sadakane's suffix sorting algorithm [8], showing that this is $O(n \log n)$ in the worst case. We also give a small improvement in the space requirements of this algorithm. We conclude that Sadakane's algorithm, which has previously been shown to outperform the closely related algorithm of Manber and Myers [6] in practice, also endures a theoretical comparison.

1. Introduction

Suffix sorting, the problem of sorting the complete list of indexes of a string of symbols according to the lexicographical order of the corresponding suffixes of the string, is currently studied primarily for two applications: The sorted list, referred to as a *suffix array* [6] or PAT *array* [5], may be used for binary search yielding a less space-consuming—while asymptotically also less time-efficient—alternative to a *suffix tree*. Lately, suffix sorting has also become important as a component of the *Burrows-Wheeler transform* [3], used for data compression.

Sorting suffixes differs from ordinary string sorting in that the elements to sort are overlapping strings of length linear in the input size n. This implies that a comparison-based algorithm which uses $\Omega(n \log n)$ comparisons may require $\Omega(n^2 \log n)$ time, and a normal radix sorting algorithm may require $\Omega(n^2)$ time.

Linear time for sorting can be achieved by building a suffix tree and obtaining the sorted order from the leaves. If a suffix-link based suffix-tree construction algorithm [7, 10] is used, linear construction time is obtained for non-constant alphabets by hashing, and the edges of the tree subsequently bucket sorted. Farach's recursive construction algorithm [4] can be used if hashing is undesirable.

However, a suffix tree involves considerable overhead, particularly in space requirements, which commonly makes it too expensive to use for suffix sorting alone.

Manber and Myers [6] presented a radix-sorting based algorithm which takes $O(n \log n)$ time. This is recapitulated in Section 3.

Sadakane [8] showed improved practical performance in relation to several natural sorting methods, including the Manber-Myers algorithm and suffix tree construction. (Note however, that Sadakane's implementation of Ukkonen's suffix tree construction algorithm uses linked list storage of edges its construction time is thus superlinear unless the alphabet is regarded as constant.) This algorithm is recapitulated in Section 4.

The time complexity of Sadakane's algorithm has not previously been analyzed. In Section 5 we show that Sadakane's algorithm can be implemented to run in $O(n \log n)$ worst case time—the same as the Manber-Myers algorithm. Furthermore, in Section 6, we show that the extra auxiliary space introduced by Sadakane in addition to that of Manber and Myers is unnecessary. Hence, Manber-Myers has no theoretical advantage over Sadakane in the worst case. We argue that Sadakane is likely to perform better for natural data and should be superior to Manber-Myers for practical purposes.

2. Definitions

We define input as a string $X = x_0 x_1 \dots x_n$ of $n+1$ symbols, where $x_n = \$$ is a unique symbol whose value we choose to define as the lowest of all symbols.

The output of suffix sorting is defined as an array I holding the integers $0, \dots, n-1$ such that for all $i \in [1, n-1]$ the suffix beginning in position $I[\,i-1\,]$ of X lexicographically precedes that beginning in position $I[\,i\,]$.

For auxiliary space we use an array V of integers in the range $[\,1, n-1\,]$ which is used for pointers into I.

3. Manber and Myers

Manber and Myers [6] suggested an algorithm that is in principal an MSD radixsort, but where the number of passes is reduced to at most $\log n$ by taking advantage of the fact that each suffix is a prefix of another one: the order of the suffixes in the previous sorting pass is used as the keys for preceding suffixes in the next pass, each time doubling the number of considered symbols per suffix. This yields an algorithm which takes $O(n \log n)$ time in the worst case. A brief description of the algorithm is as follows:

After filling I with the numbers $0, \dots, n-1$ do the following:

1. Bucket sort I using x_i as the key for i. Regard I as partitioned into as many buckets as there are distinct symbols in X. Set k to 1.
2. I is now sorted on the k first symbols of each suffix. If all suffixes are in separate buckets, then stop.
3. For $i = n-k, \dots, n-1$, move suffix $I[\,i\,]+k$ to the beginning of its bucket and mark the position for bucket splitting.
4. For $i = 0, \dots, n-k-1$, move suffix $I[\,i\,]+k$ to the beginning of its bucket. When i crosses a bucket limit, mark accessed buckets for splitting.
5. Split the buckets thus marked.
6. Double k and go to 2.

Manber and Myers give an implementation that requires, in addition to storage space for X and I, the integer array V which is used as the inverse of I, and two boolean arrays of size n for marking buckets (which may be incorporated as the sign bits of I and V). They also show how the algorithm can be made to perform in expected linear time under certain assumptions on the probability distribution of X. However, these assumptions are hardly realistic when considering natural data, which frequently comprises highly regular patterns.

4. Sadakane

With real-life data, most of the elements of I are often sorted in one of the first few passes of the Manber-Myers algorithm, leaving only a few groups to be sorted by subsequent passes. Still, all elements of I are traversed during each pass.

Sadakane [8] improved the practical behavior of this algorithm by replacing the radix sorting mechanism with a comparison-based algorithm. Thereby, the time for each pass becomes dependent only on the number of remaining unsorted elements. The algorithm is as follows:

1. Sort I using x_i as the key for i. Regard I as partitioned into as many groups as there are distinct symbols in X. Set k to 1.
2. Sort each group of size larger than one with a comparison-based algorithm, using the the first position of the group containing x_{i+k} as the key for i when $i + k < n$, and -1 otherwise.
3. Split groups between non-equal keys.
4. Combine sequences of unit-size groups so that these can be skipped over in subsequent passes.
5. Double k, and if there are any groups larger than one left, go to 2. Otherwise stop.

Sadakane's implementation requires, in addition to storage space for X and I, the integer array V for holding group positions ($\sim$ inverse of I), one *byte* array of size n which holds sizes of groups and single-size groups as well as flags to determine, for each size cell, whether it applies to a group or a group sequence. In Section 6 we show that only X, I, and V are necessary, which eliminates the space overhead of this algorithm compared to Manber and Myers.

5. Time Complexity

The trivial upper bound on the time complexity of the Sadakane algorithm is $O(n\,(\log n)^2)$. A more detailed complexity analysis has not previously been presented. We now show that a worst case performance of $O(n \log n)$ is possible.

For the comparison-based subroutine of Sadakane's algorithm, we use Quicksort with a tripartite partition, such as the split-end partition of Bentley and McIlroy [1] (as is also suggested by Sadakane). Tripartite denotes that the partitioning routine splits the subarray assigned to it into *three* parts: one with elements smaller than the pivot element, one with elements equal to the pivot, and one with larger elements. The algorithm recursively sorts the *smaller* and *larger* parts but leaves the *equal* part, since this is already in its correct place. For guaranteed worst-case performance, we use the true median for pivot element. Locating the median, which is done in guaranteed linear time, for example using the algorithm of Schönhage, Paterson, and Pippenger [9], must thus be included in the partitioning routine. (This is hardly desirable in practice—there exists a range of pivot-choice methods which balances guaranteed worst-case versus expected performance [1, 2].)

While Sadakane proposes that the initial sorting in step 1 is done by bucket sorting, we analyze, for simplicity, a variant where the same Quicksort algorithm is used all through.

We view the sorting process as a construction of an implicit ternary tree, analogous to the search tree discussed by Bentley and Sedgewick [2]: Each call to the partitioning routine corresponds to a node in the tree, the root being the the first partitioning of the whole array in step 1. Each node has three subtrees: a middle subtree which holds the elements equal to the pivot, and left and right subtrees which hold smaller and larger elements respectively. All internal nodes have nonempty middle subtrees, while their left or right subtrees are empty for subarrays with less than three distinct keys. The tree has n leaves, corresponding to the elements of I (the suffixes), in sorted order.

Lemma 1. *The path length from the root to any leaf is at most* $2\log n+3$.

Proof. Consider first the number of middle-subtree roots on the traversal from the root to a specific leaf. At the first such node encountered, only the first symbol of each suffix is considered by the sorting. Then, at each subsequent middle-subtree root encountered, the number of symbols considered by the sorting is twice as large as at the previous one. Consequently, the full length of the suffix is considered after encountering at most $\log n+1$ middle-subtree roots, at which time sorting is done.

Now consider the left- and right-subtree roots on a similar path. For each such node encountered, the number of leaves in its subtree is at most half that number in the previous one. Thus, we are down to a single leaf after encountering at most $\log n+1$ left- or right-subtree roots.

Summing the root and the maximum number of middle-, left-, and right-subtree roots on a path, we have a path length of at most $2\log n+3$. □

Lemma 2. *Partitioning operations corresponding to all the nodes of any given depth of the tree takes at most* $O(n)$ *time.*

Proof. Partitioning a subarray takes time linear in its size. The initial array, whose partitioning corresponds to the root, has size n, and since no overlapping subarrays are ever assigned to different subtrees of any node, the total size for of all subarrays at any given depth is at most n. The total time for partitioning at this depth is thus $O(n)$. □

Theorem 1. *Suffix sorting by Sadakane's algorithm with a tripartite Quicksort as a subroutine can be done in* $O(n\log n)$ *worst-case time.*

Proof. Partitioning asymptotically dominates sorting time; splitting and combining groups is done in linear time on subarrays which are already sorted.

From Lemma 2, the total partitioning cost is at most $O(n)$ times the height of the tree. Lemma 1 implies that the height of the tree is $O(\log n)$ and consequently the total partitioning time is $O(n\log n)$. □

6. Getting Rid of the Size Array

In Sadakane's implementation $V[\,i\,]$ points to the leftmost position of the group containing suffix i in I. To find the end of each group or sequence of single size groups, an additional array is used to hold sizes.

To reduce auxiliary space to the V array only, we incorporate the group sizes into V by letting $V[\,i\,]$ point to the *last* element of the group containing suffix i. Sequences of single-size groups are handled specially: if i is the first, and j the last suffix in a sequence of single-size groups in I, we exchange $V[\,i\,]$ and $V[\,j\,]$.

Now, each sorting pass proceeds as follows: We scan I left to right. Let i denote the next unexplored position in this scan, and set j to $V[I[\,i\,]]$. If $V[I[\,j\,]] = j$ we call the sorting subroutine for the subarray $I[\,i \dots j\,]$. (Otherwise we have encountered a sequence of single-sized groups and do nothing.) We continue by setting i to $j + 1$, stopping when $i = n$. Group splitting and combining is similarly straightforward.

Wit this scheme, the keys for the sorting routine have to be picked somewhat more carefully: The elements of V are used as keys, preserving order (it does not matter whether the first or last position of a group is used as its key), except for the first and last suffix of a sequence of single-size groups, whose cells in V are exchanged. Therefore, when we would use $V[\,i\,]$ as a key, we instead use $V[I[V[\,i\,]]]$, which reverses this exchange when needed and has no effect otherwise.

7. Conclusion and Future Research

When performing suffix sorting on real-life data, particularly when this is a component of data compression—which is an important application—, a large proportion of the suffixes become sorted by the first few passes of suffix sorting algorithms such as the Manber-Myers or Sadakane algorithms, while, due to repetitions, a few suffixes require several passes for their order to be resolved. Therefore, it is desirable to limit the work in subsequent passes by treating only the few unsorted suffixes in each pass. This is clearly demonstrated by Sadakane's experiments [8].

Our analysis shows that with a carefully chosen comparison-based subroutine, Sadakane's algorithm is not, as it might seem at first glance, asymptotically inferior to that of Manber and Myers. Furthermore, we have shown that it does not require any additional space. This, in combination with its demonstrated superior practical performance, leads us to conclude that Sadakane's algorithm is generally the better choice for practical applications.

We note however, that resorting to a comparison-based algorithm is not the only possible way to exploit the fact that the number of suffixes decreases rapidly in the first passes of the sorting. In future research, we will explore the possibility of radix sorting on fewer bits at a time (i.e. less than the $\lceil \log n \rceil$ bits implicit in the Manber-Myers algorithm) to obtain sublinear time for each sorting pass while increasing the number of passes.

References

1. Jon L. Bentley and M. Douglas McIlroy, *Engineering a sort function*, Software—Practice and Experience **23** (1993), no. 11, 1249–1265.
2. Jon L. Bentley and Robert Sedgewick, *Fast algorithms for sorting and searching strings*, Proceedings of the eighth Annual ACM–SIAM Symposium on Discrete Algorithms, January 1997, pp. 360–369.
3. Michael Burrows and David J. Wheeler, *A block-sorting lossless data compression algorithm*, Research Report. 124, Digital Systems Research Center, Palo Alto, California, May 1994.

4. Martin Farach, *Optimal suffix tree construction with large alphabets*, Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, October 1997, pp. 137–143.
5. Gaston H. Gonnet and Ricardo A. Baeza-Yates, *Handbook of algorithms and data structures*, Addison-Wesley, 1991, ISBN 0-201-41607-7.
6. Udi Manber and Gene Myers, *Suffix arrays: A new method for on-line string searches*, SIAM Journal on Computing **22** (1993), no. 5, 935–948.
7. Edward M. McCreight, *A space-economical suffix tree construction algorithm*, Journal of the ACM **23** (1976), no. 2, 262–272.
8. Kunihiko Sadakane, *A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation*, Proceedings of the IEEE Data Compression Conference, March–April 1998, pp. 129–138.
9. A. Schönhage, M. Paterson, and N. Pippenger, *Finding the median*, Journal of Computer and System Sciences **13** (1976), no. 2, 184–199.
10. Esko Ukkonen, *On-line construction of suffix trees*, Algorithmica **14** (1995), no. 3, 249–260.