

CSE 584A Class 7

Jeremy Buhler

February 7, 2018

1 A Basic Algorithm for Suffix Array Construction

Let's consider a “stupid” algorithm to compute a sorted suffix array A from a string S of length n .

- *Preliminary:* I will assume that every string ends with a “special” marker symbol $\$$, which is $\notin \Sigma$.
- The $\$$ marker sorts *before* all chars in Σ .
- Why? To model lexicographic order on suffixes using only character comparisons, not length comparisons.
- For example, consider string $acac$. Suffix “ac” should sort before suffix $acac$.
- If we add $\$$ to end, we get $acac\$$, and indeed, “ac $\$$ ” sorts before “acac $\$$ ”.
- (To generalize, we may pretend that string ends with arbitrarily many copies of $\$$ as needed.)

Now, on to the algorithm.

- First, prepare an unsorted suffix array A from S .
- Now *progressively* sort A as follows.
- First, sort A based only on *first* character of each suffix.
- For all groups of ≥ 2 suffixes with same first character, sort them on first *two* characters.
- For all groups of ≥ 2 suffixes with same first two chars, sort them on first *four* characters.
- Proceed in this fashion, doubling the comparison length each time, until no group has ≥ 2 suffixes.
- (Suffixes all have different lengths, so this happens eventually.)

- When we're done, all suffixes are correctly sorted.

How fast does this algorithm run?

- Sorting length doubles each time but must be $\leq n$.
- hence, at most $\log n$ sorting passes required to sort all suffixes.
- How expensive is each sorting pass?
- Each pass could operate on up to n suffixes (though usually they will be divided into several smaller groups).
- A pass that orders by first h characters could be implemented in time $\Theta(hn)$ using a radix sort.
- (Note that this pass need sort a group only on its last $h/2$ chars, which haven't been inspected yet.)
- Since $h = 2^i$ in i th pass, worst-case total cost is

$$\sum_{i=0}^{\log n} 2^i \cdot n$$

which is $\Theta(n^2)$.

- In practice, we hope that the string does not contain many long repeats, so the order of most suffixes will be resolved before h gets too long.
- For i.i.d. random DNA sequence, we should be mostly done sorting when h reaches $\log_4 n$.
- Unfortunately, real sequences may contain lots of repeats.

2 Avoiding Redundant Comparisons

Manber and Myers (1993) gave one of the first fast (worst-case subquadratic) algorithms for suffix array construction. They showed how to make progressive sorting faster!

- **Key Idea:** use results of prior comparisons, rather than repeating them.
- We are *not* sorting arbitrary strings – we're sorting suffixes!
- **Example:** suppose we have sorted all suffixes of string “cgcgca\$” up to first two characters.

- We now need to sort the remaining groups on their first four characters.
- Naively, we must compare characters 3 and 4 of each suffix in a group.
- But consider group with suffixes 2 and 4.
- Comparing chars 3, 4 of these suffixes is equivalent to comparing characters 1, 2 of suffixes 4 and 6!
- What do we know about suffixes 4 and 6?
- Sorting them by first 2 chars placed 6 before 4, in different groups (check A)!
- Hence, suffix 4 is lexicographically $<$ suffix 2.
- Similarly, consider suffixes 1, 3, and 5.
- To sort them by second 2 chars, check order of suffixes 3, 5, and 7.
- 7 belongs before 3 and 5, but 3 and 5 are as yet unordered (in same group by first 2 chars).
- Hence, sorting on 1st four chars divides 1, 3, 5 into two groups: $\{5\}\{1, 3\}$.

The point: each pass of progressive sort can re-use information from previous passes to avoid explicit character comparisons. This requires that we can efficiently determine grouping and relative order of suffixes in the current, partially sorted array.

3 The M&M Algorithm

Let's write some code to implement Manber and Myers' improvement to progressive sort.

- We will use two arrays: A , the suffix array, and G , the *group array*.
- Each sorting pass will both update A and create G' , a revised group array that will be used for the next pass.

```

MMBUILD( $S$ ) ▷  $|S| = n$ 
  create initial, unsorted suffix array  $A$  from  $S$ 
  sort suffixes in  $A$  by first character only
  for  $i$  in  $1..n$  do
     $G[A[i]] \leftarrow$  index of leftmost entry in  $A$  with same 1st char as  $A[i]$ 

   $h \leftarrow 1$ 
  while any group has size  $> 1$  do
    for each group  $A[j..k]$  do
      sort suffixes of  $A[j..k]$  using  $\text{key}(A[i]) = G[A[i] + h]$ 
      for  $i$  in  $j..k$  do
         $G'[A[i]] \leftarrow$  index of leftmost entry in  $A[j..k]$  with same key as  $A[i]$ 
     $h \leftarrow 2 \times h$ 
   $G \leftarrow G'$ 

```

return A

- Let's look at correctness (inductively).
- After setup code, suffixes are sorted in A , and groups are assigned in G based on 1st char of each suffix.
- Now suppose in general that A has been sorted into groups by its first h chars, and G assigns groups based on these chars.
- Within one group, all strings share their first h chars.
- We sort these strings by their 2nd h chars using the group numbers of the suffixes h chars past the ones being sorted.
- Since G groups all suffixes by their first h chars, this sort will give correct ordering for the group.
- Moreover, once the group is sorted, we set entries in G' for the group based on their subgrouping, i.e., their 2nd h chars. (Recall that they all share the same first h chars.)
- Hence, after we set $G \leftarrow G'$, G contains correct groups for all suffixes by their first h chars.
- Conclude that each pass correctly updates A and G to be sorted by twice as many chars as before. QED

Our example above:

How efficient is this algorithm?

- Still only $\log n$ sorting passes.
- Each pass uses a key of *constant* size (one character initially one integer $\leq n$ in general).

- It takes linear time in the size of each group, hence $\Theta(n)$ overall, to identify group boundaries in the partially sorted array.
- Once a group is re-sorted, it takes linear time in its size, hence $\Theta(n)$ overall, to update the G' array with the new subgroup numbering.
- If we use a general comparison sort on keys to sort the suffixes in each group, each pass costs $\Theta(n \log n)$, for total cost of $\Theta(n \log^2 n)$.
- If we use a radix sort on keys, each pass costs $\Theta(n)$, for total cost of $\Theta(n \log n)$.

Note: we can show that the final group array G is inverse permutation of the suffix array; that is, $G[A[i]] = i$. Put another way, $G[i]$ is the *rank* (i.e. the offset) of suffix $S[i..n]$ in the sorted suffix array. (*Exercise*)

4 Enhancements to Basic M&M

Here are some hints to make Manber-Myers go faster in practice.

- Do we really need separate arrays G and G' ?
- If a group with number g contains k suffixes, these suffixes are stored in array cells $A[g..g + k - 1]$.
- After subsorting, group indices for these suffixes remain in range $g..g + k - 1$.
- Moreover, no other suffixes of A have group numbers in this range.
- Suppose we simply overwrite this group's entries in G , rather than creating G' .
- Comparisons of suffixes from group g with those *not* from group g yield the same result – the “out-group” suffix has group # either $< g$ or $> g + k - 1$.
- Comparisons of two suffixes within group g are no less informative than before (and may be *more* informative due to sub-sorting of g).
- Hence, correctness is maintained if we just overwrite each group's entries in G as we go.
- (This optimization saves n words of memory and *might* sort faster.)

What else can we do?

- In practice, Manber-Myers spends a lot of time finding groups to sort.
- It must read over all “singleton” suffixes (which are already correctly sorted) to locate non-singleton groups, and it must find the length of each non-singleton group before sorting it.
- Hence, we must make $\log n$ passes over entire array.
- This is extremely wasteful, especially in later passes when almost all suffixes are sorted and therefore singletons.

- Instead, keep an auxiliary list that enumerates each non-singleton group's start and length in A . Use this list in each pass to avoid reading singletons and recomputing group lengths. Create an updated list during each pass to be used in next pass.
- Doesn't help asymptotic complexity but can greatly improve constant factor and takes only $\Theta(n)$ more space (and often much less than linear).
- Larsson and Sadakane (1999, 2007) show how to avoid rereading singleton groups without using storage beyond A and G , and add some other sneaky tricks such as using QuickSort instead of radix sort while maintaining $O(n \log n)$ complexity. For details, see <https://www.sciencedirect.com/science/article/pii/S0304397507005257>
- A comparison of algorithms in 2014 (see link below) suggests that LS is within an order of magnitude of the practically fastest algorithms available to directly construct suffix arrays from a variety of different kinds of strings.

For more pointers on practical construction of suffix arrays, including an interesting newer algorithm and a comparison of many practical algorithms, have a look at the following recent-ish paper:

<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4101535/>