String Data Structures: Suffix Arrays

DANIEL ANDERSON [1]

We just saw the suffix tree data structure, a compact structure for processing and answering questions about strings. In addition to the suffix tree, there are many other data structures that utilise the suffixes of a string to perform efficient string related queries. We will study one more such structure as a more space efficient alternative to the suffix tree, the suffix array.

Summary: Suffix Arrays

In this lecture, we cover:

- Suffix arrays
- How to build a suffix array
- Applications of suffix arrays

Recommended Resources: Suffix Arrays

- Weiss, Data Structures and Algorithm Analysis, Section 12.4.1
- `https://youtu.be/NinWEPPrkDQ` - MIT 6.851 lecture on string data structures (advanced)
- `https://visualgo.net/suffixarray` - Visualisation of suffix arrays
- `http://www.allisons.org/ll/AlgDS/Strings/Suffix/`

What are Suffix Arrays?

Suffix arrays are a compact data structure that index all of the suffixes of a particular string in sorted order. While this may not sound immediately useful at first sight, the suffix array has a mountain of applications in string processing which carry over to applications in fields ranging from the study of natural languages to bioinformatics.

Consider as a first example, the string `banana`. Recall that when working with suffixes, we often mark the end of the string with a special character, denoted by \$. The suffixes of `banana$` are then given by

```
banana$
anana$
nana$
ana$
na$
a$
$
```

In general, we can see that a string of length N has $N+1$ suffixes (including the empty suffix containing only \$.) The suffix array of a string is a sorted array of its suffixes, so the suffix array of banana looks like

```
$
a$
ana$
anana$
banana$
na$
nana$
```

Note that the special character (\$) is considered to be lexicographically less than all other suffixes, so it appears at the beginning of the suffix array. This is useful since it conceptually signifies the *empty string* (the suffix of length zero of the original string.)

[1] FACULTY OF INFORMATION TECHNOLOGY, MONASH UNIVERSITY: `daniel.anderson@monash.edu`. These notes are based on lecture slides by Arun Konagurthu, the textbook by CLRS, some video lectures from MIT OpenCourseware, and many discussions with students.

Since actually writing down all of the suffixes of a string would take $O(N^2)$ space, suffix arrays are typically represented in a more compact form, where each element simply refers to an index $1 \leq i \leq N$ into `S` at which the corresponding suffix begins. For example, the suffix array for `banana` would be stored as:

```
SA("banana$") = [7,6,4,2,1,5,3]
```

where the i^{th} index corresponds to the position in `banana` at which the i^{th} sorted suffix starts as shown in the table bellow.

Index	Suffix
7	`$`
6	`a$`
4	`ana$`
2	`anana$`
1	`banana$`
5	`na$`
3	`nana$`

In practice, the first index is often omitted since it is guaranteed that the empty suffix is always the first one.

Building a Suffix Array

The naive approach

The simplest and most obvious way to build a suffix array is to simply build a list containing the indices `1..N` and to sort them by comparing suffixes.

Algorithm: Naive Suffix Array Construction

```
1: function SUFFIX_ARRAY(S[1..N])
2:     Set SA[1..N] = [1..N]
3:     sort(SA[1..N], suffix_compare(S, ...))
4:     return SA
5: end function
6:
7: function SUFFIX_COMPARE(S[1..N], i, j)
8:     return (S[i..N] < S[j..N])
9: end function
```

Although short and simple, this implementation is useless in practice since its time complexity is $O(N^2 \log(N))$, owing to the fact that sorting performs $O(N \log(N))$ comparisons and each pair of suffixes takes $O(N)$ time to compare, which means that this algorithm can not be used on large strings.

A more clever approach: prefix doubling

The naive algorithm for constructing suffix arrays is simply too slow to work on large strings so a better approach is needed in practice. A much faster approach which is still conceptually simple is the *prefix doubling* algorithm.

Key Ideas: Prefix Doubling Algorithm

To construct the suffix array of the string `S[1..N]`

1. Sort the length 1 (single character) substrings
2. Sort the length 2 substrings by using the known sorted order of the length 1 substrings
3. Sort the length 4 substrings by using the known sorted order of the length 2 substrings
4. Sort the length 8 substrings by using the known sorted order of the length 4 substrings
5. ...
6. Sort the length 2^{k+1} substrings by using the known sorted order of the length 2^k substrings
7. Continue until all of the suffixes are sorted

Note: If a suffix beginning at a particular position has length less than 2^k at a particular iteration, it is still considered, remembering that the empty string compares less than all other strings, eg. "cat" < "cathode."

Since we are sorting fixed length substrings at each iteration, you can imagine that what we are really doing is sorting consecutively larger prefixes of the suffixes, hence the name prefix doubling. To elaborate, how do we sort the length 2^{k+1} substrings by using the known sorted order of the length 2^k substrings? The key observation is that any length 2^{k+1} substring is simply the concatenation of two length 2^k substrings.

For example let S_1`[1...`2^{k+1}`]` be the concatenation of the two strings `A[1..`2^k`]` and `B[1..`2^k`]` and let S_2`[1...`2^{k+1}`]` be the concatenation of the two strings `C[1..`2^k`]` and `D[1..`2^k`]`. If we know that $A < C$, then we immediately know that $S_1 < S_2$ as well. If instead $A > C$, then we know that $S_1 > S_2$. Otherwise, if $A = C$, then $S_1 < S_2$ if and only if $B < D$. The most important fact that we notice here is that the comparisons take just $O(1)$ operations, rather than the $O(2^{k+1})$ operations that would be required to compare the strings character by character.

Algorithm: Prefix Doubling Suffix Array Construction

```
1:  function SUFFIX_ARRAY(S[1..N])
2:      Set SA[1..N] = [1..N]
3:      Set rank[1..N] = [ord(S[1..N])]
4:      for k = 1 to N, stepping k *= 2 do
5:          sort(SA[1..N], suffix_compare(rank, k, ...))
6:          Set temp[1..N] = [0,0,0,...]
7:          for i = 1 to N-1 do
8:              temp[SA[i+1]] = temp[SA[i]] + suffix_compare(rank[1..N], k, SA[i], SA[i+1])
9:          end for
10:         swap(temp, rank)
11:     end for
12:     return SA
13: end function
14:
15: function SUFFIX_COMPARE(rank[1..N], k, i, j)
16:     if rank[i] ≠ rank[j] then
17:         return rank[i] < rank[j]
18:     else if i+k ≤ N and j+k ≤ N then
19:         return rank[i+k] < rank[j+k]
20:     else
21:         return j < i
22:     end if
23: end function
```

In the algorithm above, the array `rank` is used to keep track of the relative order of each sorted prefix in the partially sorted suffix array. The value `ord(c)` simply refers to the order or rank of the character `c` in the alphabet. The `suffix_compare` function compares two prefixes in $O(1)$ by looking at the ranks of the two halves of the prefix. Finally, we simply loop `k = 1, 2, 4, 8, ...`, consecutively sorting the prefixes of length k and recomputing the new ranks until all of the suffixes are sorted.

Sorting at each iteration of the algorithm takes $O(N \log(N))$ time since the comparisons are now performed in $O(1)$,

and we have to perform $\log(N)$ iterations to fully sort the suffixes, so the total time complexity of the prefix doubling algorithm is $O(N \log^2(N))$.

Speeding up prefix doubling

The prefix doubling algorithm is already significantly faster than the naive suffix array construction method, but there is still lots of room for improvement. Since our alphabet size can assumed to be no larger than the length of our string `S`, we can speed up prefix doubling by replacing the $O(N \log(N))$ `sort` with an $O(N)$ radix sort instead.

We only need to perform two passes of radix sort on each iteration, one to sort the prefixes at positions `1+k,2+k,3+k, ...` followed by a sort on the prefixes at positions `1,2,3,...` to bring all of the prefixes of size $2k$ into order. Note that we have to make sure that each pass of radix sort is stable in order to obtain the correct final order.

Using this method, the complexity of the prefix doubling algorithm drops to $O(N \log(N))$.

Can we do even better? (NOT EXAMINABLE)

Prefix doubling with radix sort achieves reasonably good performance on very large strings in practice, but we can still do even better! Several algorithms exist for constructing suffix arrays in just $O(N)$ time for a fixed size alphabet. Probably the most well known of these algorithms is the DC3 algorithm[2], which works by sorting a set of "sample suffixes" recursively, which are the suffixes at positions in the string that are not divisible by three. Using the sample suffixes, one can then sort the remaining 1/3 of the suffixes quickly using radix sort in a fashion similar to prefix doubling. By performing the bulk of the work recursively on only 2/3 of the string at each level of recursion, the algorithm runs in $O(N)$ time for any fixed size alphabet.

Applications of Suffix Arrays

As fun as suffix arrays are to think about, they also admit a wide range of useful applications to string processing problems. Let's look at just a few of them.

Pattern matching with suffix arrays

Recall that the pattern matching problem is the problem of finding all occurrences of a given pattern string `P[1..M]` inside a bigger text string `T[1..N]`. Many well known algorithms exist that will solve this problem in $O(N+M)$ per query, but what if we wish to keep the same text string `T[1..N]` and search for lots of small patterns `P[1..M]`? Doing the $O(N)$ work per query might turn out to be extremely wasteful if the text string has length $N = 1,000,000$ and the patterns only have length $M = 20$.

Using the suffix array of the text string `T[1..N]`, we can perform pattern searches fast, saving time if a large number of searches are desired. The idea is very simple: since the suffixes are sorted, all occurrences of the pattern that we are searching for will exist in some contiguous range of the suffix array, which we can find with binary search.

[2] See *Linear Work Suffix Array Construction* , Juha KÂĺarkkÂĺainen, Peter Sanders, Stefan Burkhardt

Algorithm: Pattern Matching using the Suffix Array

```
1:  function FIND_PATTERN(SA[1..N], T[1..N], P[1..M])
2:      Set lo = 0, hi = N
3:      while lo ≠ hi - 1 do
4:          Set mid = floor((lo + hi)/2)
5:          if T[SA[mid]..N] < P[1..M] then
6:              lo = mid
7:          else
8:              hi = mid
9:          end if
10:     end while
11:     Set begin = hi
12:     lo = 1, hi = N + 1
13:     while lo ≠ hi - 1 do
14:         Set mid = floor((lo + hi)/2)
15:         if T[SA[mid]..N] ≤ P[1..M] then
16:             lo = mid
17:         else
18:             hi = mid
19:         end if
20:     end while
21:     Set end = lo
22:     return begin, end
23: end function
```

All of the occurrences of the pattern are then found at the positions given by `SA[begin..end]` if this range is non-empty, otherwise there were no matches.

The algorithm simply performs two binary searches, one to locate the beginning of the range where the pattern appears in the suffix array, and a second to locate the end of the range where the pattern appears in the suffix array. Since each string comparison takes $O(M)$ time and the binary searches must perform $O(\log(N))$ iterations, the total time complexity of pattern matching using a suffix array is $O(M\log(N))$ which is a significant improvement over $O(N+M)$ when $N \gg M$. It is possible to speed this up to just $O(M)$, which is optimal, using the LCP array (see the next section, not examinable).

The longest common prefix array (NOT EXAMINABLE)

The longest common prefix (LCP) array is a companion to the suffix array that stores the length of the longest common prefix shared by each pair of adjacent suffixes in the suffix array. Taking the example of `banana`, whose sorted suffixes are

`a, ana, anana, banana, na, nana,`

The lengths of the longest common prefixes between each pair of suffixes is given by

`0, 1, 3, 0, 0, 2.`

(`a` and `ana` share the prefix `a`, while `ana` and `anana` share the prefix `ana`, etc.) Conventially, the LCP array begins with a zero since the longest common prefix of any string with the empty string is empty. There are many algorithms for computing the LCP array in linear time, see *Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications,* Kasai et al.

Equipped with the LCP array, a number of problems become easy to solve with suffix arrays. For example, consider the longest repeated substring problem which is the problem of finding the longest substring in a given string that occurs multiple times. Equipped with the LCP and suffix arrays, the longest repeated substring just occurs at the position corresponding to the maximum value in the LCP array.