Simple Linear Work Suffix Array Construction

Authors: : Juha Kärkkäinen and Peter Sanders

6.827 Paper Presentation

Presenter: Edmund Williams

May 2022

Table of Contents

Suffix Array Problem

- Given a length n string return an array (of length n) where the value at the ith index is the rank of the $n - i$ long suffix of the given string.
- Suffix arrays are more memory efficient alternative to suffix trees, both are used to solve a number of common string operations e.g. locating substrings, longest common substring, repetition of substrings etc.
- Naive Solution1: Merge sort an array of all the suffices. Worst case is $O(n^2 log(n)$, it is usually assumed that comparisons are $O(1)$ but in this problem comparing to string is proportional to the length of the string in the worst case. This adds multiplicative factor of $O(n)$
- Naive Solution2: Radix sort an array of all the suffices. Worst case though is $O(n^2)$ due to the suffices being $O(n)$ giving a depth of $O(n)$.

Suffix Array Problem

- Given a length n string return an array (of length n) where the value at the ith index is the rank of the $n - i$ long suffix of the given string.
- Suffix arrays are more memory efficient alternative to suffix trees, both are used to solve a number of common string operations e.g. locating substrings, longest common substring, repetition of substrings etc.
- Naive Solution1: Merge sort an array of all the suffices. Worst case is $O(n^2 log(n)$, it is usually assumed that comparisons are $O(1)$ but in this problem comparing to string is proportional to the length of the string in the worst case. This adds multiplicative factor of $O(n)$
- Naive Solution2: Radix sort an array of all the suffices. Worst case though is $O(n^2)$ due to the suffices being $O(n)$ giving a depth of $O(n)$.

Suffix Array Problem

- Given a length n string return an array (of length n) where the value at the ith index is the rank of the $n - i$ long suffix of the given string.
- Suffix arrays are more memory efficient alternative to suffix trees, both are used to solve a number of common string operations e.g. locating substrings, longest common substring, repetition of substrings etc.
- Naive Solution1: Merge sort an array of all the suffices. Worst case is $O(n^2 log(n)$, it is usually assumed that comparisons are $O(1)$ but in this problem comparing to string is proportional to the length of the string in the worst case. This adds multiplicative factor of $O(n)$
- Naive Solution2: Radix sort an array of all the suffices. Worst case though is $O(n^2)$ due to the suffices being $O(n)$ giving a depth of $O(n)$.

Suffix Array Problem

- Given a length n string return an array (of length n) where the value at the ith index is the rank of the $n - i$ long suffix of the given string.
- Suffix arrays are more memory efficient alternative to suffix trees, both are used to solve a number of common string operations e.g. locating substrings, longest common substring, repetition of substrings etc.
- Naive Solution1: Merge sort an array of all the suffices. Worst case is $O(n^2 log(n)$, it is usually assumed that comparisons are $O(1)$ but in this problem comparing to string is proportional to the length of the string in the worst case. This adds multiplicative factor of $O(n)$
- Naive Solution2: Radix sort an array of all the suffices. Worst case though is $O(n^2)$ due to the suffices being $O(n)$ giving a depth of $O(n)$.

Example Input/Output

Here is an example using the word = *mississippi*. The third row is the desired output, the values indicated represent the starting index of a given index, and the order of these values gives the sorted order of all the suffices

$$P = \left[\begin{array}{c|c|c|c|c|c|c|c|c|c|c} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ m & i & s & s & i & s & s & i & p & p & i \\ \hline 10 & 7 & 4 & 1 & 0 & 9 & 8 & 6 & 3 & 5 & 2 \\ \hline i & i & i & i & m & p & p & s & s & s & s \\ & p & s & s & i & i & p & i & i & s & s \\ & p & s & s & s & & i & p & s & i & i \\ & i & i & s & s & & & p & s & p & s \\ & & p & s & i & & & i & i & p & s \\ & & p & s & s & & & & p & i & i \\ & & i & i & s & & & & p & & p \\ & & & p & i & & & & i & & p \\ & & & p & p & & & & & & i \\ & & & i & p & & & & & & \\ & & & & i & & & & & & \end{array}\right]$$

Previous Work

- M. Farach[1] proposed a divide conquer approach to the problem that solves suffix arrays in $O(n)$

High Level Steps

Step 1: Recursively solve the ordering of suffices with only a subset of them
Step 2: Leveraging the results of step 1 to order the remaining set of suffices (this step is not to be done recursively)
Step 3: Merge both sets of suffices in $O(n)$

- In Farach's work the merge step is complicated and is difficult to implement (this paper didn't report whether or not Farach's work is practically performant)

[1] M. Farach. Optimal suffix tree construction with large alphabets. In Proc. 38th Annual Symposium on Foundations of Computer Science, pages 137–143. IEEE, 1997

Table of Contents

New Algo

- The proposed algorithm in this paper has the same high level steps, but makes modifications to what subsets are recursed on and the the merge step was significantly simplified.

High Level Steps

Step 1: Recursively order suffices whose length is not divisible by 3 (any suffix, S, where $len(S) mod 3! = 0$

Step 2: Order all suffices whose length is divisible by 3 using the results of step 1:

Step 3: Merge all the suffices through a variation of a stable radix sort.

Splitting up Suffices

- Let $S_i = s[i : n]$, the suffix from the ith letter to the end of the given string s
- Core Idea: S_i could be rewritten as the pair $(s[i], S_{i+1})$, enabling the use of radix sort on these length 2 pairs
- For step 2, all S_i that has a length of multiple of 3, can be written as a pair $(s[i], S_{i+1})$, and all S_{i+1} will be order already from step 1, so all S_{i+1} can compared in constant time since the suffix array already exists, this avoids the potential $O(n)$ comparison time for length $O(n)$ suffices. Using this pair idea a radix sort can be applied.
- For step 3, is similar to step 2, with the exception of S_j where $len(S_j) mod 3 == 2$, we rewrite as a triple $(s[j], s[j+1], S_{j+2})$ where $len(S_{j+2}) mod 3 == 1$. The triple is important so that S_j can be compared with something that comes from the suffix array generated from step 1

Splitting up Suffices

- Let $S_i = s[i : n]$, the suffix from the ith letter to the end of the given string s
- Core Idea: S_i could be rewritten as the pair $(s[i], S_{i+1})$, enabling the use of radix sort on these length 2 pairs
- For step 2, all S_i that has a length of multiple of 3, can be written as a pair $(s[i], S_{i+1})$, and all S_{i+1} will be order already from step 1, so all S_{i+1} can compared in constant time since the suffix array already exists, this avoids the potential $O(n)$ comparison time for length $O(n)$ suffices. Using this pair idea a radix sort can be applied.
- For step 3, is similar to step 2, with the exception of S_j where $len(S_j) mod 3 == 2$, we rewrite as a triple $(s[j], s[j+1], S_{j+2})$ where $len(S_{j+2}) mod 3 == 1$. The triple is important so that S_j can be compared with something that comes from the suffix array generated from step 1

Splitting up Suffices

- Let $S_i = s[i : n]$, the suffix from the ith letter to the end of the given string s
- Core Idea: S_i could be rewritten as the pair $(s[i], S_{i+1})$, enabling the use of radix sort on these length 2 pairs
- For step 2, all S_i that has a length of multiple of 3, can be written as a pair $(s[i], S_{i+1})$, and all S_{i+1} will be order already from step 1, so all S_{i+1} can compared in constant time since the suffix array already exists, this avoids the potential $O(n)$ comparison time for length $O(n)$ suffices. Using this pair idea a radix sort can be applied.
- For step 3, is similar to step 2, with the exception of S_j where $len(S_j) mod 3 == 2$, we rewrite as a triple $(s[j], s[j+1], S_{j+2})$ where $len(S_{j+2}) mod 3 == 1$. The triple is important so that S_j can be compared with something that comes from the suffix array generated from step 1

Splitting up Suffices

- Let $S_i = s[i : n]$, the suffix from the ith letter to the end of the given string s
- Core Idea: S_i could be rewritten as the pair $(s[i], S_{i+1})$, enabling the use of radix sort on these length 2 pairs
- For step 2, all S_i that has a length of multiple of 3, can be written as a pair $(s[i], S_{i+1})$, and all S_{i+1} will be order already from step 1, so all S_{i+1} can compared in constant time since the suffix array already exists, this avoids the potential $O(n)$ comparison time for length $O(n)$ suffices. Using this pair idea a radix sort can be applied.
- For step 3, is similar to step 2, with the exception of S_j where $len(S_j) mod 3 == 2$, we rewrite as a triple $(s[j], s[j+1], S_{j+2})$ where $len(S_{j+2}) mod 3 == 1$. The triple is important so that S_j can be compared with something that comes from the suffix array generated from step 1

Step 1, Labeling + Recursion

- Goal: Generate a suffix array for all suffices whose length isn't a multiple of 3
- For all $imod3! = 0$, let T_i be the triple $s[i : i + 2]$
- Perform radix sort on the set of triples and let r_i be the rank of triple i in the sorted list (note that there can be duplicate triples of the same rank), and this gives us the free property that if $r_i < r_j$ then $T_i < T_j$ (lexicographically)
- Let s^{12} be the concatenation of the array of all r_i when $imod3 = 1$ and the array of all r_i when $imod3 = 2$ (image next slide). Recursing on s^{12} yields the desired suffix array. Why?
- Note that $[T_i, T_{i+3}, T_{i+6}...T_n]$ has a 1:1 correspondence with the suffix S_i and if we have a suffix array for T_i where $imod3! = 0$, then we have a suffix array for S_i where $imod3! = 0$
- Note: the purpose of the recursive step is to break ties in between triples and generating an ordering of suffices that start with the same triple

Step 1, Labeling + Recursion

- Goal: Generate a suffix array for all suffices whose length isn't a multiple of 3
- For all $imod3! = 0$, let T_i be the triple $s[i : i + 2]$
- Perform radix sort on the set of triples and let r_i be the rank of triple i in the sorted list (note that there can be duplicate triples of the same rank), and this gives us the free property that if $r_i < r_j$ then $T_i < T_j$ (lexicographically)
- Let s^{12} be the concatenation of the array of all r_i when $imod3 = 1$ and the array of all r_i when $imod3 = 2$ (image next slide). Recursing on s^{12} yields the desired suffix array. Why?
- Note that $[T_i, T_{i+3}, T_{i+6}...T_n]$ has a 1:1 correspondence with the suffix S_i and if we have a suffix array for T_i where $imod3! = 0$, then we have a suffix array for S_i where $imod3! = 0$
- Note: the purpose of the recursive step is to break ties in between triples and generating an ordering of suffices that start with the same triple

Step 1, Labeling + Recursion

- Goal: Generate a suffix array for all suffices whose length isn't a multiple of 3
- For all $imod3! = 0$, let T_i be the triple $s[i : i + 2]$
- Perform radix sort on the set of triples and let r_i be the rank of triple i in the sorted list (note that there can be duplicate triples of the same rank), and this gives us the free property that if $r_i < r_j$ then $T_i < T_j$ (lexicographically)
- Let s^{12} be the concatenation of the array of all r_i when $imod3 = 1$ and the array of all r_i when $imod3 = 2$ (image next slide). Recursing on s^{12} yields the desired suffix array. Why?
- Note that $[T_i, T_{i+3}, T_{i+6}...T_n]$ has a 1:1 correspondence with the suffix S_i and if we have a suffix array for T_i where $imod3! = 0$, then we have a suffix array for S_i where $imod3! = 0$
- Note: the purpose of the recursive step is to break ties in between triples and generating an ordering of suffices that start with the same triple

Step 1, Labeling + Recursion

- Goal: Generate a suffix array for all suffices whose length isn't a multiple of 3
- For all $imod3! = 0$, let T_i be the triple $s[i : i + 2]$
- Perform radix sort on the set of triples and let r_i be the rank of triple i in the sorted list (note that there can be duplicate triples of the same rank), and this gives us the free property that if $r_i < r_j$ then $T_i < T_j$ (lexicographically)
- Let s^{12} be the concatenation of the array of all r_i when $imod3 = 1$ and the array of all r_i when $imod3 = 2$ (image next slide). Recursing on s^{12} yields the desired suffix array. Why?
- Note that $[T_i, T_{i+3}, T_{i+6}...T_n]$ has a 1:1 correspondence with the suffix S_i and if we have a suffix array for T_i where $imod3! = 0$, then we have a suffix array for S_i where $imod3! = 0$
- Note: the purpose of the recursive step is to break ties in between triples and generating an ordering of suffices that start with the same triple

Walk Through

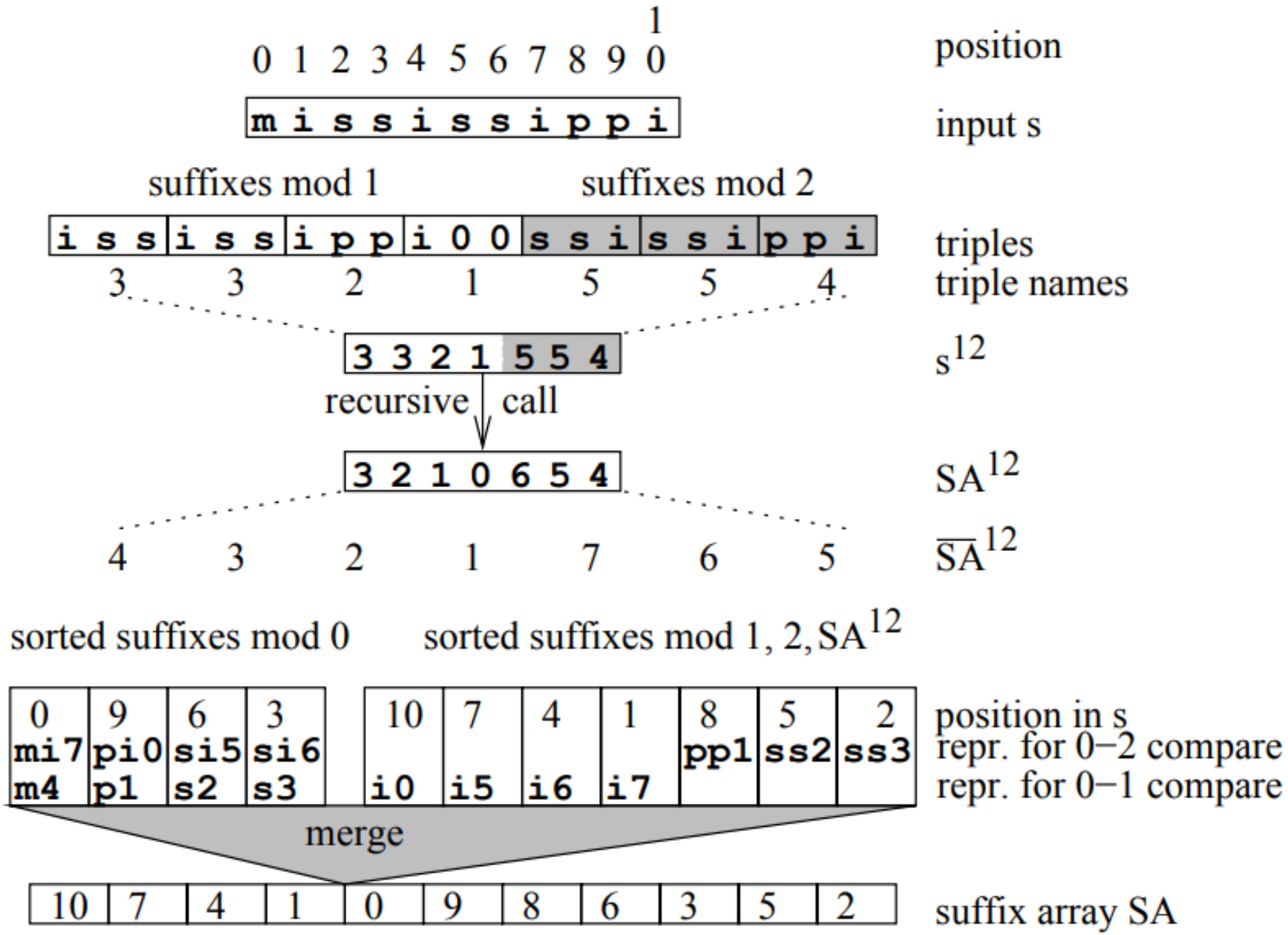

Table of Contents

Contributions

- Improvements over Farach's work in regards to simplicity of the algorithm.
- Asymptotic improvements in the BSP and EREW-PRAM models: " first linear work BSP algorithm"
- Generalization to how large the recursive subproblem should be for this technique to work. Solving that question is a reduction to difference cover problem.