Succinct Data Structures, Suffix Tree and Nearly Optimal BST

Data Structures - Assignment 4

Winter 2022

1. Question

[15 marks] Draw the suffix tree for the string mississippi. Append a $ (the end of file symbol) to the end of the string when drawing the suffix tree.

Solution:

Input: `mississippi$`; The below figure is a suffix tree for the string `mississippi$`

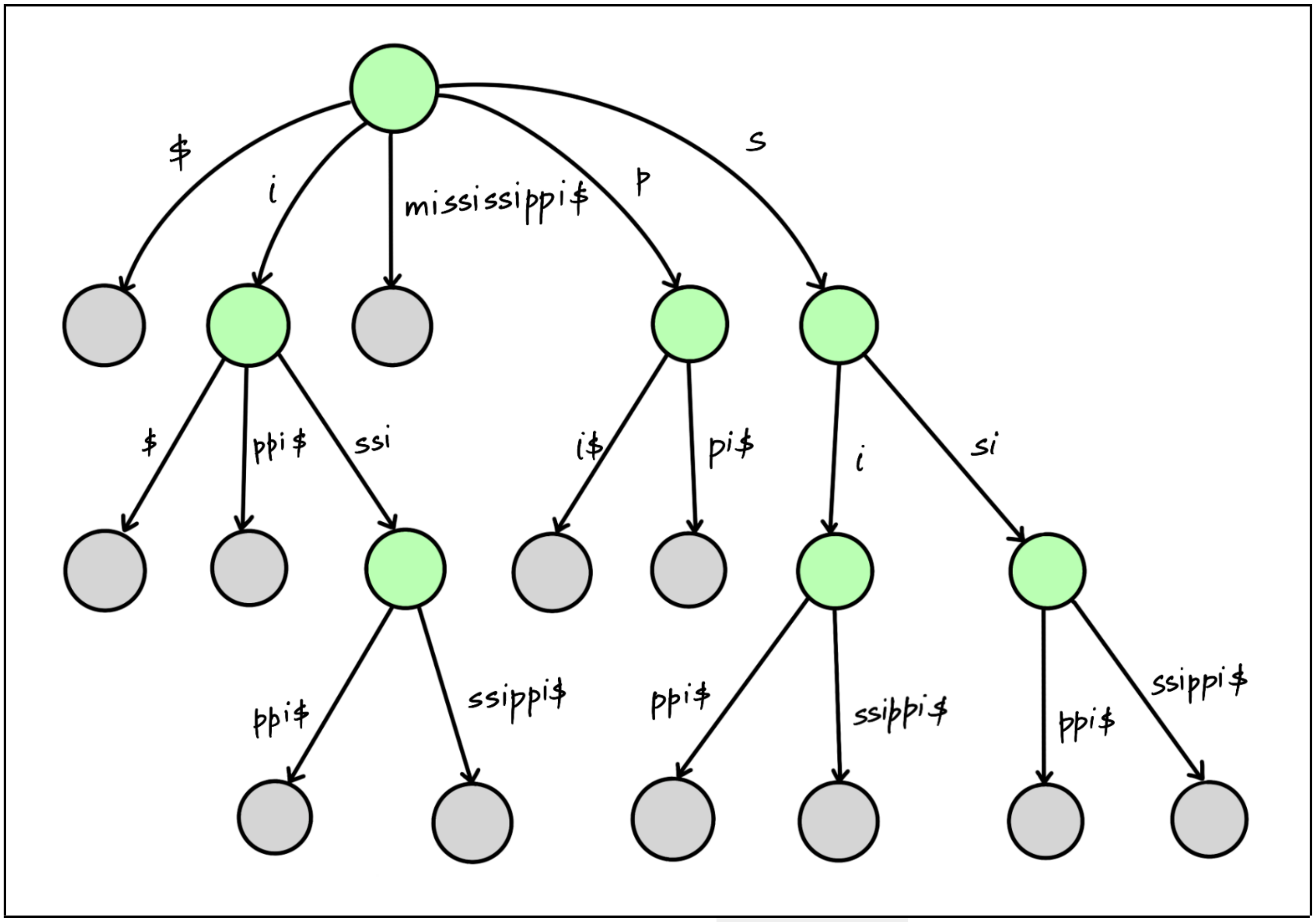

Figure 1: Suffix Tree for `mississippi$`

2. Question

[10 marks] In class, we learned how to construct nearly optimal binary search trees in linear time. The running time is given by the following recursion

`T(n) = O(lg min{i, n−i+1}) + T(i−1) + T(n−i)`

Prove by induction that `T(n) = O(n)`.

Hint: Pages 85-86 of the CLRS book might help.

Solution:

Searching for `i`$_o$ with exponentially increasing (a faster greedy approach) steps from both ends. i.e., `1, n, 1+1, n-1, 1+2, n-2, 1+4, n-4, 1+8, n-8` and so on; this determines an interval `[1+(2^c) , 1+2^(c+1)] ([(n-2)^(c+1), (n-2)^c])` for `i`$_o$ in `O(c)` steps, a binary search determines `i`$_o$ in `O(min(log i`$_o$`,log(n-i`$_o$`)))` time.

Approach 1:

The recurrence relation is given by:
`T(n) = O(lg min{i, n−i+1}) + T(i−1) + T(n−i)`, to prove that it is a Linear solution, i.e., `T(n) = O(n)`.

Observation: Starting from both ends, the worst case is choosing `(n/2)th` position as the root.

We start with the guess that the solution and try induction: `T(n) ≤ cn – dlg(n)`
For some constant c and d (for all `1 <= i <= n`)
As mentioned, in our inductive hypothesis, we assume `T(n) ≤ cn – dlg(n)` for all positive numbers less than `n`

Therefore, `T(i-1)` would be `c(i-1) - dlg(i-1)` and `T(n-i)` would be `c(n-i) - dlg(n-i)`
So, substitute `T(n) = c(lg min{i, n−i+1}) + ci - c - dlg(i-1) + cn - ci - dlg(n-i)`
`= ci - c - dlg(i-1) + cn - ci - dlg(n-i) + c(lg i)` for `1 ≤ i ≤ n/2`

We require, `dlg(i-1) + dlg(n-i) + c > lg(n)`

Which is fine when i is large, however, `c` and `d` have to be chosen appropriately for smaller values of i, such that the inequality is respected

Approach 2:

This approach is very similar to the prior one but with slight modifications to better simplification and proof.

Induction on n: `T(n) ≤ (2d + c)n - d lg(n + 1)`
It is certainly true for the base case for n = 0 and for n > 0, we have:

`T(n) ≤ T(i-1) + T(n-i) + O(lg min(i, n-i+1))` for `1 ≤ i ≤ n`

```
T(n) ≤ T(i-1) + T(n-i) + d(lg min(i, n-i+1)) + d + c
```
for `1 ≤ i ≤ (n+1)/2`
```
    = T(i-1) + T(n-i) + d(log i) + d + c
```

by the symmetry of the above expression in `(i-1)` and `(n-i)`. Applying the induction hypothesis, we get:

```
≤ (2d+c)(i-1+n-i) - d(lg i + lg (n-i-1)) + d log i + d + c
 = (2d+c)n + [-d(1 + log(n-i+1))] ------------(1)
```

The equation in the square brackets is always negative and is max for `i = (n+1)/2`
Therefore,

```
T(n) ≤ (2d+c)n - d(1 +lg(n+1)/2)
     = (2d+c)n - dlog(n+1)
```

Note: the addition of d and c is for easier simplification of the expression, it still leads to a similar equation and the proof still holds; the proof would be:

```
= T(i-1) + T(n-i) + d(log i)
= (2d+c)(i-1) - d(lg i) + (2d+c)(n-i) - d(lg n - i + 1) + d(log i)
= (2d+c)i - (2d+c) + (2d+c)n - (2d+c)i - d(lg n - i + 1)
= (2d+c)n - (2d+c) - d(lg n - i + 1)
```

The equation (1) would look like:
```
= (2d+c)n + [- ((2d+c) + d(lg n - i + 1))]
```

Irrespective, the upper bound is going to be `O(n)`

3. Question

[10 marks] Let A be a string of length m over a constant-size alphabet, and B be a string of length n over the same alphabet. We wish to find the longest common (contiguous) substring of A and B, i.e., the longest string that appears as a (contiguous) substring of both A and B. Design an algorithm to solve this problem in `O(m+n)` time. Show your analysis of the running time. You are not required to give pseudocode, but feel free to give pseudocode if it helps you explain your algorithm.

Hint: It may be helpful to construct a suffix tree. However, it is unlikely that a suffix tree built over A$ or B$ will help you achieve the desired running time. Think about what else you could do.

Solution:

Given: m and n be the lengths of two strings A and B, respectively.
Assuming the size of the alphabet is constant, to prove that the longest common substring of two strings can be found in `O(m+n)` time.

Overview: The longest common substrings for the two strings are found by building a generalized suffix tree for the given two strings and finding the deepest internal nodes with leaf nodes from all the strings in the subtree below it.

Taking an example, let A = `xabxa` and B = `babxba`; instead of building the suffix tree for A and B individually, we combine the two strings with unique terminal symbols, we have a new string `A#B$`, i.e., `xabxa#babxba$`

The below figure is a suffix tree for the string `xabxa#babxba$`

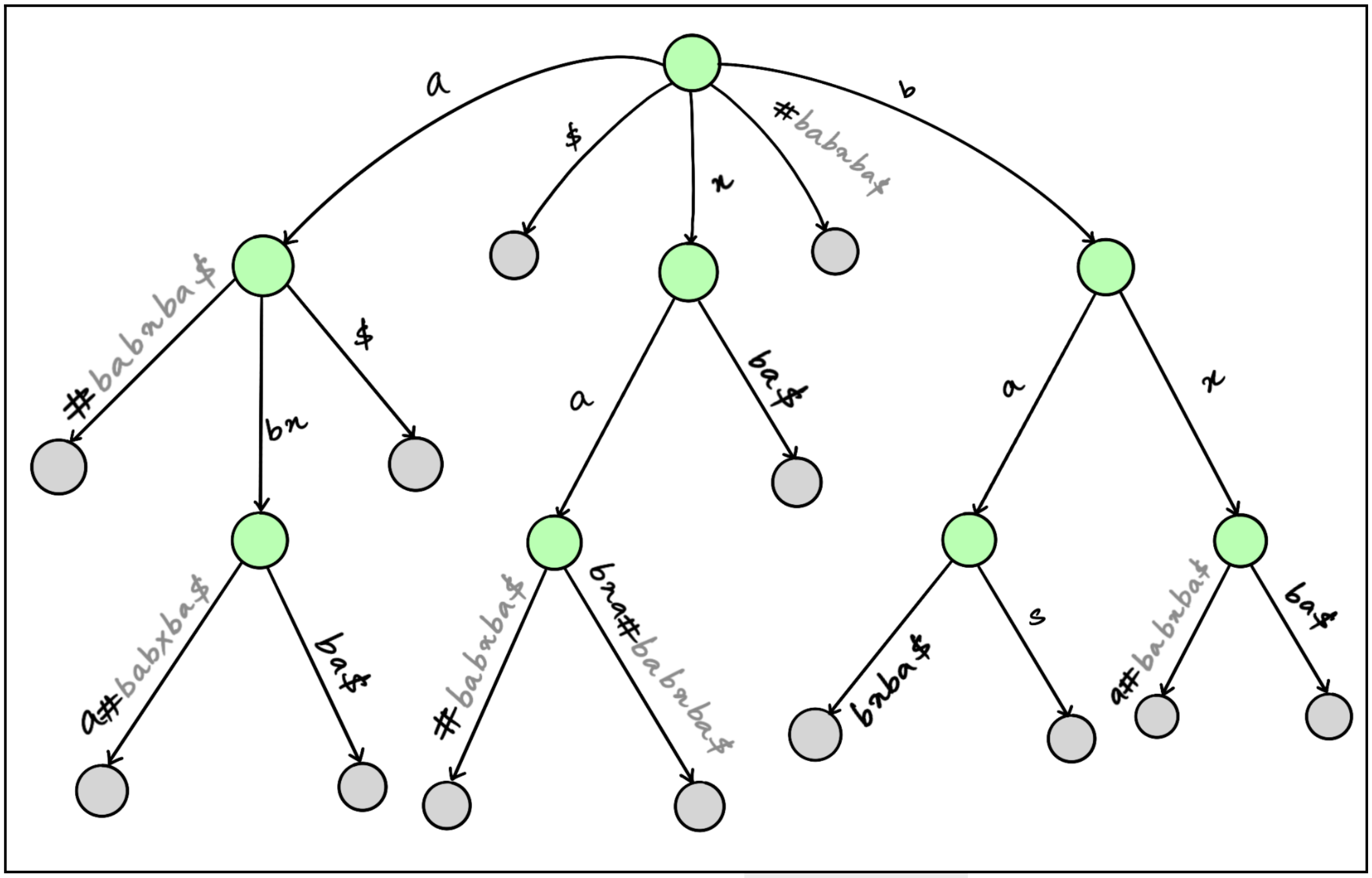

Figure 2: Suffix Tree for `xabxa#babxba$`

The suffix tree can be further refined by only having the strings belonging to either A or B; the new suffix tree would be the same as *Figure 2*, without the grayed text at each node.

Analysis of running time: We know that the above Suffix Tree construction takes `O(m+n)` time, and finding Longest Common Substring is nothing but Depth First Search on the tree, which is again `O(m+n)` time. So overall, the time complexity is `O(m+n)`.

4. Question

[15 marks] In class, we learned the succinct data structures that can support the rank queries over a bit vector of length n. Among the set of structures constructed, one of them is a look-up table F . This table stores, for each possible bit vector of length `(lg n)/2` and each position in it, the answer to rank.
Now, we construct a different table E. Table E has one entry for each possible bit vector of length `(lg n)/2` , and it stores the number of 1s in this bit vector.
For simplicity, we assume that `(lg n)/2` is an integer.

1. [7 marks] Show that E occupies `O(√n lg lg n)` bits.
2. [8 marks] In the set of data structures constructed to support rank, we replace table F by table E. Show how to use the resulting set of data structures to support rank in constant time. You are allowed to use bit operations.
   Show your analysis of the running time. You are not required to give pseudocode, but feel free to give pseudocode if it helps you explain your algorithm.

Solution Part-1:

Analysis:

1. `Rank(i)` is the number of 1s until the position i (Including position `i`).
2. Old lookup table F: For a bit vector of length `n`, the lookup table F has `2^(0.5 lg n)` entries, and each bit in an entry is associated with rank;
3. Space taken by table F: `2^(0.5 lg n)` entries, where each row takes a space of `(0.5 lg n) * (lg (0.5 lg n)`; hence the total space is: `2^(0.5 lg n) * (0.5 lg n) * (lg (0.5 lg n))`.
4. New lookup table E: `2^(0.5 lg n)` entries, but each entry is of `(lg (0.5 lg n))`; this is because table E has one entry per row, which stores the total number of 1s in the entry.

Putting it together for proof:

The total space for Table F: `2^(0.5 lg n) * (0.5 lg n) * (lg (0.5 lg n))`
Instead of storing the rank of space `(lg n)/2` for each bit in the entry, we now store the total number of 1s for each entry; therefore, the space for Table E: `2^(0.5 lg n) * (lg (0.5 lg n))`

$$\text{Table F} \longrightarrow 2^{\frac{\lg n}{2}} \cdot \frac{\lg n}{2} \cdot \lg\left(\frac{\lg n}{2}\right)$$

$$= \sqrt[2]{2^{\lg n}} \cdot \frac{\lg n}{2} \cdot \lg \lg n - \lg 2$$

$$= \sqrt{n} \cdot \frac{\lg n}{2} \cdot \lg \lg n$$

$$\text{which is } o\left(\sqrt{n} \lg n \cdot \lg \lg n\right) \longrightarrow (1)$$

1) $2^{\lg n} = n$

2) $\lg 2 = 1$

3) $a^{m/n} = \sqrt[n]{a^m}$

4) $\lg\left(\frac{a}{b}\right) = \lg a - \lg b$

$$\text{Table E} \longrightarrow 2^{\frac{\lg n}{2}} \cdot \lg\left(\frac{\lg n}{2}\right)$$

$$= \sqrt[2]{2^{\lg n}} \cdot \lg \lg n - \lg 2$$

$$= \sqrt{n} \cdot \lg \lg n$$

$$\text{which is } o\left(\sqrt{n} \cdot \lg \lg n\right) \longrightarrow (2)$$

Equation (1) shows that E occupies `O(√n lg lg n)` bits.

Solution Part-2:

Assumption: `(lg n)/2` is an integer.

Below is the explanation in the form of steps:

1. Consider a bit vector `B[0..n)`.
2. Consider a super block of size $b_1 = (\log n)^2$
3. Let's say the ranks of the super block are stored in array R_1, which takes a space of `O(n/lg n) = o(n)` bits.
4. And each super block is divided into sub-blocks of size $b_2 = 0.5 \lg n$
5. And another array R_2 stores the relative ranks to the nearest preceding super block, hence: $R_2[i] = Rank_B[ib_2] - R1[\lfloor ib_2/b_1 \rfloor]$
    a. This uses $O(\lg b_1) = O(\lg \lg n)$ bits per entry, R_2 needs `O((n lg lg n)/lg n) = o(n)` bits

6. Therefore, $\mathrm{Rank}_B[i] = R_1[k_1] + R_2[k_2] + \mathrm{bitcount\text{-}1}(B[k_2 b_2 ... i))$ where, $k_1 = \lfloor i/b_1 \rfloor$ and $k_2 = \lfloor i/b_2 \rfloor$

7. **Note:** The `bitcount` here is the bit shifting (logical right shifts) by (length of bit vector of size - i), and then use the lookup table to find the number of 1s, and this operation takes constant time.
   Example: For `1011`, to find the number of 1s until `101`, we can do bit shifts `(4-3)` times `1011 >>> 1` → `0101` and then use the lookup table.

Conclusion:

Hence, the bits can be counted in constant time, with a lookup table of $O(n^{1/2} \lg \lg n) = o(n)$ bits. A bit vector `B[0..n)`, when augmented with the data structure of $O((n \log \log n)/ \log n) = o(n)$ bits, rank queries take constant time.

References

[1] "Generalized suffix tree," *Wikipedia*, Mar. 11, 2022. https://en.wikipedia.org/wiki/Generalized_suffix_tree (accessed Mar. 17, 2022).

[2] K. Mehlhorn, "Nearly optimal binary search trees," *Acta Informatica*, vol. 5, no. 4, 1975, doi: 10.1007/bf00264563.

[3] K. Mehlhorn, "Data Structures and Algorithms 1: Sorting and Searching" EATCS Monographs on Theoretical Computer Science. Springer-Verlag.

[4] A. Fariña, S. Ladra, O. Pedreira, and Á. S. Places, "Rank and Select for Succinct Data Structures," *Electronic Notes in Theoretical Computer Science*, vol. 236, pp. 131–145, Apr. 2009, doi: 10.1016/j.entcs.2009.03.019.