Suffix Trees and Arrays

Stephen Huan

February 7, 2020

1 Suffix Trees

1.1 Trie

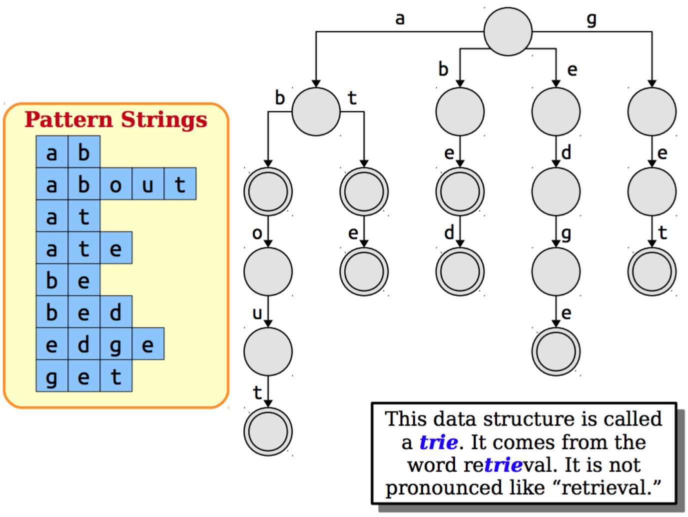


Figure 1: The trie data structure

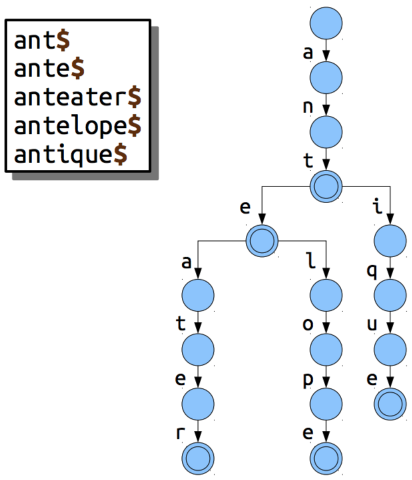
A ***trie*** is a tree based data structure which stores a list of strings. Once a trie has been created, certain operations are faster.

For example, consider the "autocomplete" problem, or the problem of finding whether a given target string is a prefix of any of the pattern strings in the list. Let T be the target string of length m and the multiple pattern strings be $P_1, P_2, \ldots, P_k$. We then define n to be $\Sigma_{i=1}^{k} |P_i|$, or the total length of all the pattern strings. Propagate a pointer down the trie, following edge lables which correspond to characters in the target string. If you "fall off" the trie, that is, there aren't edges anymore, then it isn't a prefix. The notation I will use for running time is $\langle f, g \rangle$, where f is the preprocessing time and g is the query time. For example, the naïve algorithm of scanning every pattern string is $\langle O(1), O(n)\rangle$, as it has no preprocessing time but could scan every character in the patterns. However, our trie algorithm yields an $\langle O(n), O(m)\rangle$ algorithm, which is asymptomically optimal as it's necessary to read in the input string in linear time.

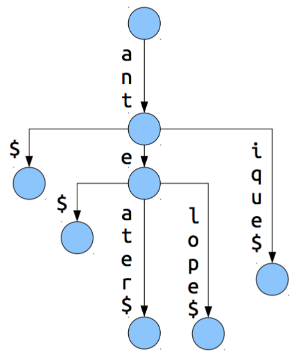
Why is it important to find prefixes? Consider that a substring is a prefix of a suffix. First, we define a suffix by its starting index i, such that the suffix i is equal to all characters including and after i, i.e. $s[i:]$. All substrings start at a given index and have a particular length. The start of a suffix is represented by the suffix at that index, and its length is a prefix of a suffix. Therefore, all substrings are prefixes of a suffix.

If we want to find whether or not a string T is a substring of another string P, we can compute it efficiently by instead finding whether T is a prefix of any suffix of P. This leads to the next data strucuture, a trie of all the suffixes of a particular string.

1.2 Suffix Trie



(a) A trie



(b) The resulting Patricia trie

Figure 2: A trie and its corresponding Patricia trie

First, we'll add an extra delimiter character to the end of the string, "\$" by convention (because it's lexicographically less than most characters and stands for sentinel), so that no suffix is a prefix of any other suffix in order to keep the suffixes distinct. Then, if we use our typical $O(n)$ time algorithm to generate a trie from all the suffixes of a string, there are $1+2+3+\cdots+m = \frac{m(m+1)}{2}$ total characters, which is a $O(m^2)$ time algorithm. However, note that some nodes are useless. If a node only has one child, then there's only one path that it could go down. Therefore, we could "compress" these so-called silly nodes into an edge, which yields a tree called a ***Patricia trie*** (or Radix trie). This new structure has at most one node for each string added to the trie, thus it can theoretically be constructed in $O(m)$ because there are m suffixes.

A Patricia trie of suffixes is known as a ***suffix tree***.

1.3 Suffix Tree

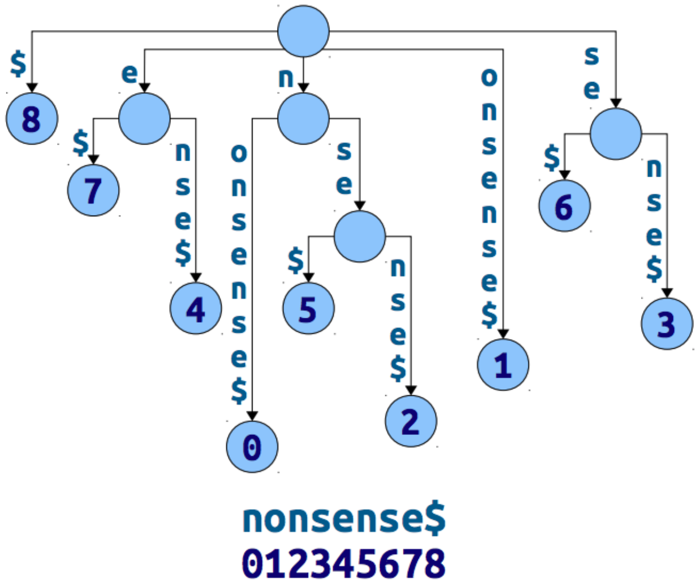


Figure 3: A suffix tree

We arrive at our final data structure. The obvious use is substring matching, but there are a variety of uses dicussed in the sample problems below.

In to actually implement suffix trees, certain factors need to be considered. First, edges in the tree are substrings of the original string. Thus, we'll represent an edge by its start and end indexes in the string. Second, for a particular node the edges emanating from it have unique first characters (if they didn't, then there'd be a new node). Thus, we'll actually represent children via a dictionary mapping the first character of an edge to a node which contains the actual start and end indexes representing the edge. This allows an O(1) lookup to determine whether or not to follow an edge in the tree. Third, the leaves of the suffix tree correspond to suffixes. We'll label leaves by the starting index of the suffix, as usual. Lastly, we'll copy the original string and store the copy at the root of the tree, in order to actually read what the edges say.

Certain properties of the suffix tree are important. First, we say a node in the suffix tree "corresponds" to the string created by adding all of the edges to get to that node from the root. The root corresponds to the empty string, commonly denoted ϵ. Then, internal nodes in the suffix tree correspond to ***branching words***, or substrings which appear in the string at least twice and at least two of their occurrences have distinct characters after them. That is, a string ω is branching if there exists at least two strings ωa and ωb which appear in the original string. In order to prove this, we'll prove each direction. If a word is not branching, then it can't be an internal node. A word isn't branching if it isn't in the string multiple times or there's the same character after each occurence. For the first case, it can't be an internal node since it doesn't have children.

For the second case, it's not an internal node because we can extend its edge by at least one character. Now, we prove that if a node is an internal node it corresponds to a branching word. Since the node is internal, it must have more than one children (since we merge nodes with only one children together) and thus it repeats more than once. Also, each of its children must have a distinct starting character and thus a node is internal in a suffix tree if and only if it corresponds to a branching word. The second is that since the leaves are suffixes, a sorted depth first traversal (following lexicographically minimal edges first) will yield the suffixes in sorted order.

This brings us to our next data structure, an array of all the suffixes in sorted order called a ***suffix array***.

2 Suffix Arrays

Suffix trees take too much memory (3-6 machine words per character in the string). Although this is a linear memory cost, consider human DNA which is three billion characters long. Normally, each character is a byte but since DNA only has four different characters (A, C, G, T) it's possible to have 1 nucleotide per 2 bits and thus 4 characters per byte. Thus, DNA is 800 MB and assuming a 64 bit machine translates into almost 96 GB of memory (way more than a typical computer can store).

$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ABANANABANDANA$

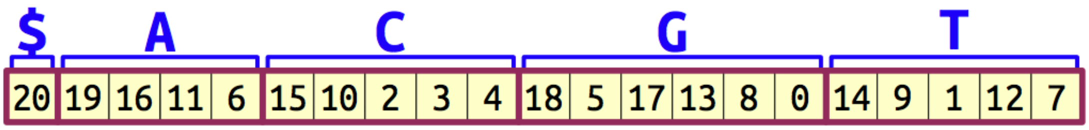
Figure 4: A suffix array

Instead of storing a tree, we'll store an array of the suffixes in sorted order, which only requires one word per character. Although the above photo has strings, we're actually storing integer starting positions as usual.

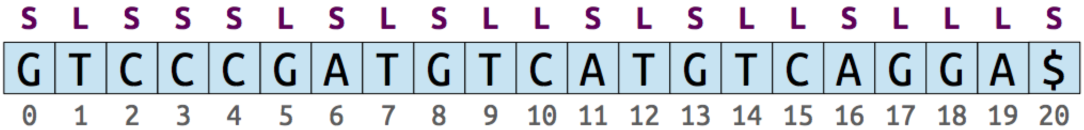
2.1 Construction

I mentioned before that a DFS on a suffix tree yields a suffix array. However, we'll actually construct the suffix array first and use it to construct a suffix tree. The naïve algorithm is to use a standard $O(n \log n)$ comparision based sort on each suffix, however each comparision takes $O(n)$ time and thus the entire algorithm is $O(n^2 \log n)$.

Instead, we'll use suffix arrays by induced sorting (***SA-IS***).



(a) Each suffix bucketed by first character



(b) Each suffix labeled as S-type or L-type

Figure 5: Bucketing and labeling applied to a string

First, we'll bucket each suffix by its first character. This gives a pretty good preliminary sort but we don't know the position of each suffix in its bucket yet. Second, we'll label each index as ***S-type*** or ***L-type***. An index is S-type if the corresponding character in the string is lexicographically less than the index after it, or if it's the sentinel. A index is L-type if it's lexicographically greater, and if the two are equal then the index has the same type as the index after it.

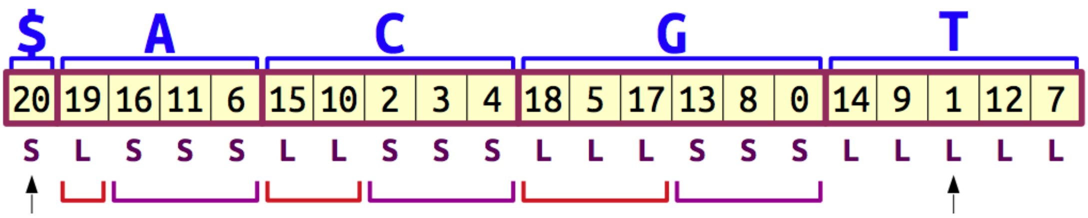
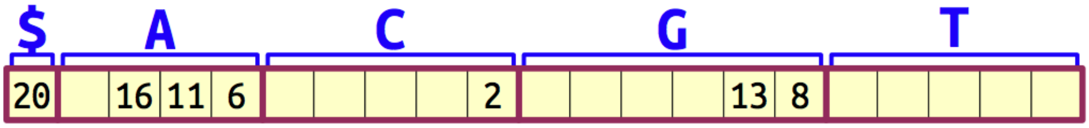


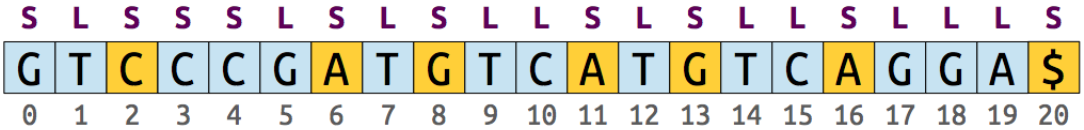
Figure 6: Patterns under labeling and bucketing

We notice that in each bucket, the L-types come before the S-types. This is because if an index is L-type, the character *after* it is going to be smaller than it is. Correspondingly, if an index is S-type, then the character after it is larger, therefore L-type suffixes will be lexicographically smaller than S-type suffixes in the same bucket.

Since we can bucket and label in $O(n)$, we just need to sort the suffixes within each group, i.e. suffixes with the same starting character and type. We then define ***LMS suffixes***, or ***L***eft***M***ost ***S***-type suffixes. A suffix is a LMS suffix if it's S-type and the suffix before it is L-type. The key observation is that if we can sort the LMS suffixes, we can infer the sorted order of all other suffixes via "induced sorting".



(a) Buckets with boundaries



(b) LMS suffixes

Figure 7: Buckets with boundaries containing the LMS suffixes in sorted order

To see how that is possible, first determine the boundaries of each bucket by counting how many times each character occurs in the string. Then, put the LMS suffixes in sorted order, and from the right of the each bucket boundary. Then, for each L-type suffix in order, put it at the front of its corresponding bucket. Reset the rightmost bucket boundaries and then do a reverse scan of the S-type suffixes, and put them at the back of each bucket, overwriting the existing LMS suffixes there.

The intuition behind this is that it simulates a multiway merge, but the details are a bit messy and to be honest I don't understand it that well myself.

However, we still need to sort the LMS suffixes. First, we define LMS blocks to be the substring formed by starting from a particular LMS suffix to the next LMS suffix.

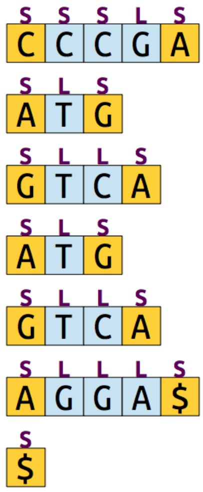


Figure 8: LMS blocks

Then, the order of LMS suffixes depends on the order of their LMS blocks. In order to sort LMS blocks, we can just put them into the buckets in an arbitrary manner (in order of appearance is easiest) then run the aforementioned induced sort.

In order to actually sort the LMS suffixes from the sorted LMS blocks, the problem is that LMS blocks end "early", that is, before their corresponding suffix ends. However, note that a LMS suffix can be made up of multiple LMS blocks joined end to end. Thus, sorting LMS suffixes is equivalent to rewriting the string as a sequence of LMS blocks.

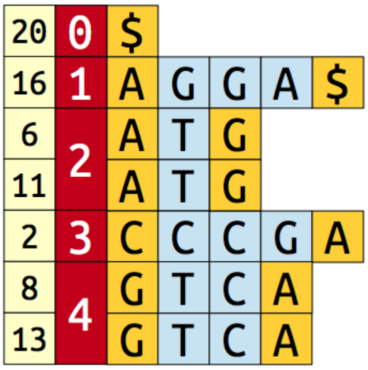


Figure 9: LMS block numbers

Number each LMS block, giving the same block the same number (check whether a block is equal to the one directly after it - when we sort the blocks, equal blocks have to be adjacent to each other). Then, we now have a new "reduced string" formed by adding block numbers in the order of when the LMS block appears in the original string. We number the LMS blocks because a smaller number will natually be lexicographically smaller than a larger number, and the suffix array of this new reduced string will sort the LMS suffixes (since a LMS suffix is equivalent to its block plus all the blocks after it, which is represented in the reduced string).

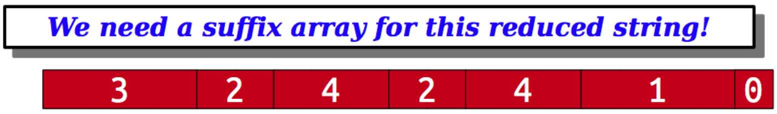


Figure 10: The corresponding reduced string

We now have a problem though. Our algorithm for finding a suffix array requires finding a suffix array. However, we can resolve this seemingly paradoxial situation with recursion. What's the base case?

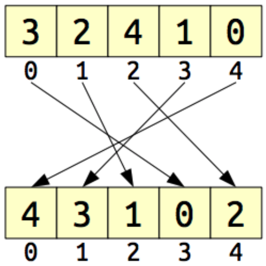


Figure 11: The base case

If all the blocks are distinct, then the suffix array is just the inverse of the reduced string, i.e. if the index i corresponds to the number k in the reduced string, then the index k is i in the suffix array. Otherwise, we recur.

SA-IS(*T*):

Scan *T* from right-to-left to mark each character as *S*-type or *L*-type.
Identify all the LMS suffixes of *T*.

Run induced sorting using the LMS suffixes in the order they appear in in *T*.

Scan the result, gathering LMS suffixes in the order they ended up in.
Number the LMS blocks, assigning duplicate blocks the same number.
Form the reduced string *T'* from the block numbers.

If all blocks are unique, get a suffix array for *T'* by directly inverting *T'*.
Otherwise, get a suffix array for *T'* by calling SA-IS(*T'*).

Use the suffix array for *T'* to sort the LMS suffixes of *T*.
Do a second induced sorting pass of *T* using the LMS suffixes in sorted order.

Figure 12: Summary of SA-IS

The running time of this algorithm is analyzed as follows. First, the reduced string has to be at most half of the length of the original string, since each LMS block is at least two characters (a S-type and a L-type) and each block contributes one character to the reduced string. Second, at each level of the recursion the algorithm runs in $O(n)$. Then, the overall running time is $O(n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 1)$. Since the geometric series $1 + \frac{1}{2} + \frac{1}{4} + \ldots$ converges to 2, then the algorithm runs in $O(2n) = O(n)$.

One nontrivial implementation detail is to convert the string from a list of characters to a list of integer ASCII values. This prevents character overflow (possible in C++) and is more convenient overall as the reduced string and the original string are both lists of integers. If you keep it as strings, then if there are more than ten blocks the number will have two digits and thus will not be represented in one character. However, with integers this won't happen.

A sample implementation in Python is given here along with all other algorithms discussed in this lecture. An alternative C++ implementation is given here.

3 LCP Array

Our linear time suffix array is all well and good, except just as is a suffix array can't replace a suffix tree. As aforementioned, the leaves of a suffix tree correspond to suffixes and the internal nodes correspond to branching words. The suffix array gives us the leaves, but what about internal nodes?

We define the ***L**ongest **C**ommon **P**refix* (LCP) to be the length of the longest common prefix. For example, the LCP between "abcd" and "abef" is 2.

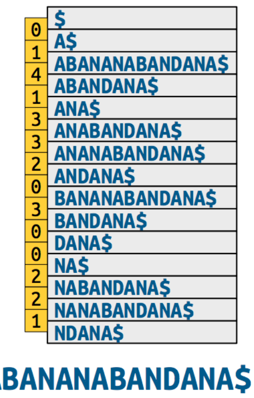


Figure 13: LCP array

The ***LCP array*** is then the LCP value between adjacent suffixes in the suffix array. Note that the LCP value corresponds to the ***string depth*** (the length of a node's corresponding string) of the lowest common ancestor (LCA) between the two leaves in the suffix tree since in a trie, parents are the longest possible prefixes of their children.

3.1 Construction

Luckily, the algorithm to construct a LCP array is relatively simple compared to SA-IS. First, consider the LCP value h between a particular suffix i and the suffix k right before it in the suffix array. If you think of the suffix after i, $i+1$ then that suffix must have a LCP of at least $h-1$ with its adjacent suffix because i and $i+1$ have the same prefix with one character removed for $i+1$, and thus the suffix $k+1$ matches with $i+1$ at least $h-1$ characters.

In order to find the position of a suffix in the suffix array, we first compute the inverse of the suffix array, or the rank array. Then, we start with the longest suffix or the suffix of index 0. Compute h between the suffix 0 and its adjacent suffix by the naïve algorithm, i.e. starting at $h=0$ and checking whether $\mathrm{s[i+h] == s[k+h]}$, incrementing h if they are. Store this value at the index of 0. From our previous observation, the next suffix of 1 has to match with its adjacent suffix at least $h-1$ characters, and thus we can decrement the current value of h by 1 and start checking from there.

3.2 Non-pairwise LCP

However, our LCP array only stores the LCP value for adjacent suffixes. How do we generalize to the LCP between any two suffixes?

The observation is that the LCP value between two nonadjacent suffixes is the minimum LCP value between them in the LCP array. To see why this is true, notice that LCP decreases monotonically moving away from a suffix. Since the suffix array is sorted lexicographically, suffixes with a larger LCP value must be closer than suffixes with a smaller LCP value to a particular suffix. Call the LCP value h and the minimum LCP value between the two suffixes m. $h \leq m$ because of monotonicity. Also, $h \geq m$ because a range minimum of m implies that there exists a prefix of length m common to all the suffixes between two suffixes, thus the LCP is at least m. If $m \leq h \leq m$, then $h = m$.

Thus, we can construct a $\langle O(n), O(1) \rangle$ range minimum query (RMQ) structure on the LCP array (most likely Fischer-Heun). Then, the LCP between two suffixes i and j is lcp[rmq(lcp, i, j - 1)] assuming $i \leq j$ which is an $O(1)$ query.

4 Constructing Suffix Trees

We've seen how both suffix arrays and LCP arrays can be constructed in $O(n)$.
How do we construct suffix trees in $O(n)$?

There exists direct algorithms like Ukkonen's but we'll use a more indirect method. First, recall that a inorder DFS of the suffix tree results in the suffix array. Second, recall that the LCP value corresponds to the string depth of a node in the suffix tree. These two observations inspire Cartesian trees. A ***Cartesian tree*** is defined as a binary tree whose inorder traversal yields the original array and is a min-heap.

Notice how the two properties of a Cartesian tree align with the two properties of the suffix tree. If we build a Cartesian tree on the LCP array, then the inorder traversal of the Cartesian tree will yield the LCP array which corresponds to suffixes. Also, within the same subtree a greater string depth implies a greater height, thus the min-heap property guarantees that in the same subtree greater LCP values are lower in the tree.

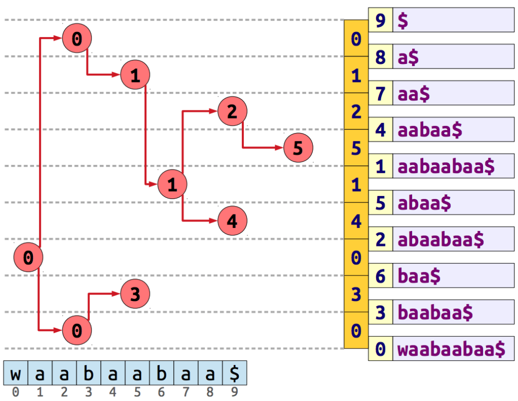


Figure 14: Cartesian tree built on the LCP array

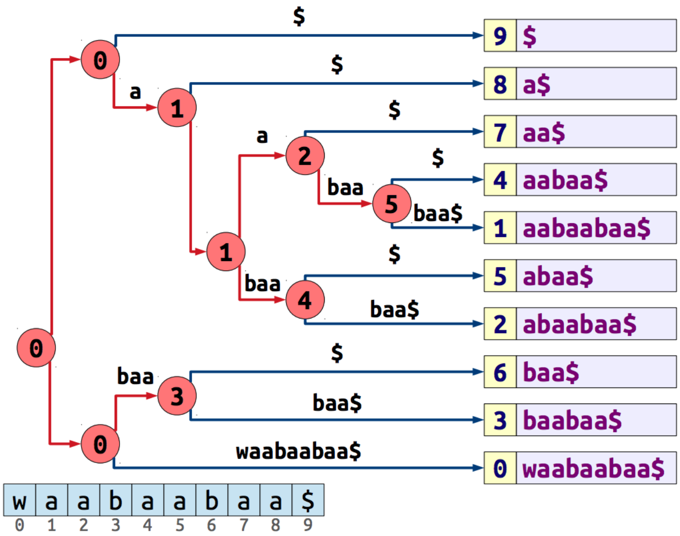


Figure 15: Cartesian tree with labeled edges

DFS on the Cartesian tree, adding suffixes as children to nodes missing children. Then, label edges based off the LCP values. Recall that edges are represented by a start and end index, and that suffixes start at their index and end at the end of the string. A node will end at the minimum start of its children minus 1, and will begin at its end minus the difference in LCP between it and its parent (the length of the edge).

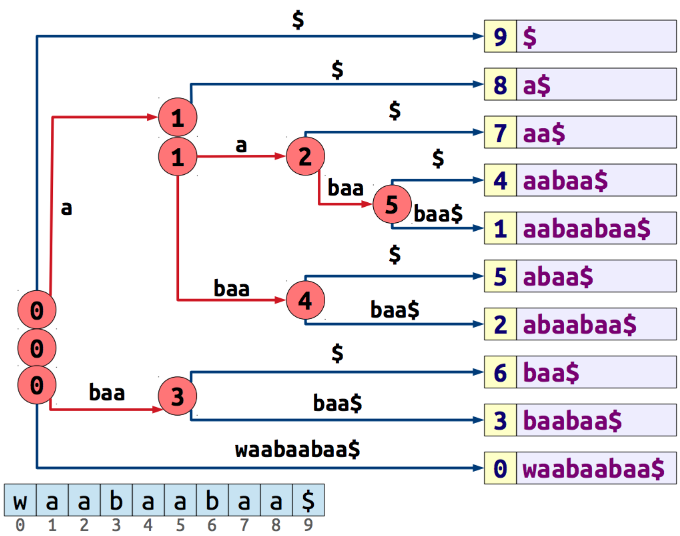


Figure 16: The final suffix tree formed by merging nodes

Also, merge nodes with the same LCP values into their parents before labeling edges.

5 Generalizations

5.1 Generalized Suffix Arrays

The suffix structures we've discussed up to this point only apply to a single string. What about multiple strings?

Luckily, suffix arrays generalize extremely easily. Simply concatenate all the strings together, with a unique delimiter in between them.

```
def generalized_suffix_array(words: list) -> tuple:
    n = [v for i, word in enumerate(words) \
        for v in (list(map(ord, word)) + [i - len(words)])]
    return n, suffix_array(n)
```

5.2 Generalized Suffix Trees

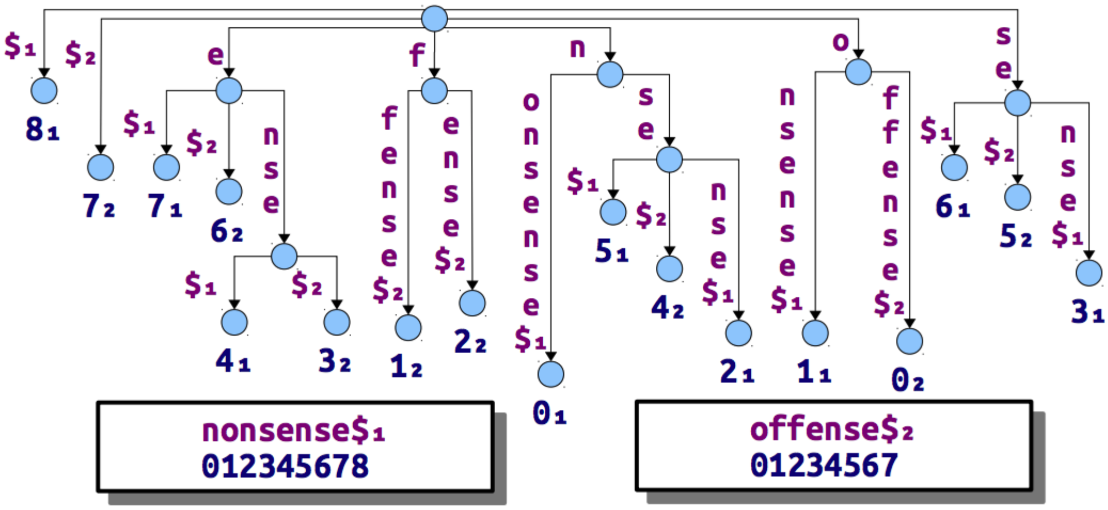


Figure 17: A generalized suffix tree

Creating a generalized suffix tree takes more work. Keep track of a mapping from suffix to the string it belongs to, and a string to its starting position in the concatenated string. Then, we have to prune out suffixes by ending them at their own string's delimiter and not the end of the concatenated string.

The resulting algorithm is to, as before, concatenate all the strings together, with a unique delimiter in between them. Create a suffix tree, and while creating the suffix tree instead of suffixes ending at the end of the concatenated string they end at the start of the next word from their corresponding word.

6 Sample Problems

1. SPOJ SARRAY: Construction of suffix array.

2. SPOJ DISUBSTR: Given a string s with length m, find the number of distinct substrings. For example "aab" has 5 ("a", "b", "aa", "ab" and "aab").

   Solution: In order to count the number of distinct substrings, we will first find the total number of substrings then subtract out duplicates. The total number will be m substrings of length 1, $m-1$ substrings of length 2, ... until 1 substring of length m. $1+2+3\cdots+m$ equals $\frac{m(m+1)}{2}$ substrings.

   Recall that a substring is a prefix of a suffix. Build a suffix array and a LCP array and think of each suffix as representing a starting point for possible substrings. Then, the number of duplicates for a particular suffix is the greatest longest common prefix (the LCP!) for any other suffix. This would seem like a n^2 loop: for each suffix, for each other suffix, find the LCP value and return the largest one. However, recall that the LCP between two suffixes is the minimum of adjacent LCP values. Therefore, LCP values must monotonically decrease and thus the largest LCP value for a particular suffix is in the precomputed LCP array. Therefore, the number of duplicates will be just the sum of the LCP array.

3. SPOJ MINMOVE: Given a string s, find its lexicographically minimal cyclic rotation, breaking ties by the starting index of the rotation.

   Solution: Think of a cyclic rotation as starting at some index i. A rotation is then formed by adding the suffix of s starting at i to the prefix of s ending at i. We can sort rotations by their suffix half with suffix arrays and then pick the first suffix disregarding the special terminator "\$" (suffix arrays are lexicographically sorted by definition, so the minimum suffix appears first in the suffix array), but that doesn't sort rotations by prefix. For example, take the string "dbczbcdbc". The algorithm might compare the suffix at index 7, "bc" with the suffix at 4, "bcdbc" and conclude that "bc" < "bcdbc" without checking the entire rotation, which after looking at "bcdbc***z***" and "bcdbc***d***" the suffix at 4 actually wins. To account for this, add the string to itself. Since the latest possible rotation occurs at the length of the string, all rotations finish within two times the length of the string. Therefore, the string doubled will allow each rotation to be completely compared. Going back to the previous example, the resulting string is "dbczbcdbcdbczbcdbc" and the suffixes are correctly compared.

   However, doubling adds two complications. First, only suffixes that occur in the first half of the string actually correspond to valid rotations. Second, for two suffixes that are identical (take the example of "aaa") the algorithm will take the shorter suffix, which corresponds to a greater index. The solution is to add a character lexicographically greater than any other character to the end of the string, forcing greater index suffixes right in the suffix array.

The final algorithm is to take s, add it to itself, then add a character (I pick "{" since it has an ASCII value 1 greater than "z"'s) and add the standard "\$" to terminate the string. Build a suffix array on this new string, and then scan from left to right, returning the first suffix which is less than the length of the string.

4. SPOJ BEADS: Identical to the previous problem, with the annoyance of 1-indexing. The "doubling" solution actually times out for me, motivating another solution.

   Prepare the LCP array for $O(1)$ RMQ as discussed above. We again scan the suffix array from left to right, although we will compare suffixes without doubling. We only need to compare suffixes if one is a prefix of another - if one suffix isn't a prefix of the other, then there is a character on which they differ, causing one to be unambiguously greater than the other. Otherwise, one suffix beats the other on length, which may not be correct for rotations. Find h, where h is the LCP value between the current best suffix b and a new suffix i. Check if h is equal to the length of one of the suffixes. If it is, then we need to compare the prefixes of the two suffixes. Since a prefix is lexicographically less than the original string, b is guaranteed to be shorter than i and thus "wraps around" s first. In order to compare b and i, find the latest point either of them shares characters and compare the character after that point. i shares h characters with b, so we start i at $i+h$ and compare it to the start of s, which is the suffix starting at 0. Compute p, the LCP value between suffixes $i+h$ and 0. i's point is then $(i+h+p) \mod (|s|-1)$ (modulo because rotations wrap around s, minus 1 because s ends with "\$"), and b's point is p. Therefore, if $s[(i+h+p) \mod (|s|-1)] <= s[p]$, update b to i.

5. Kattis burrowswheeler: Compute the Burrows-Wheeler transform of a given string. The Burrows-Wheeler transform (BWT) of a string is defined by generating all cyclic rotations then, going through the rotations in lexicographic order, constructing a new string from the last character in each rotation.

   Solution: We already know how to sort cyclic rotations of a string from the aforementioned "doubling" algorithm. There are two complications:

   (a) We need the last character of each rotation. If a rotation starts at index i, then its last character is $i-1 \mod n$.

   (b) Most online sources claim sorting cyclic rotations can be done with just T[SA[i] - 1], but this is assuming the string ends with a unique delimiter.

6. SPOJ BWHEELER: Given the BWT, recover the original string.

   A simple algorithm is given here (no relation to suffix structures).

7. SPOJ LPS: Given a string s of length n, find the length of the longest substring that is palindromic, i.e. equal to its reverse.

   Solution: The reverse of s, denoted s' might be useful because a palindrome of s will also appear in s'. However, you can't just find the longest common substring between s and s' - take the example of "abdeba" and the reverse "abedba". The longest common substring between the two is either "ab" or "ba", but the longest palindrome is one character long. Therefore, you'll need to decompose palindromes in another way.

Think of a palindrome as having a "center" index and a "radius". The palindrome starts at some center, and extends in either direction the radius number of characters. For example, the palindrome in "dfe***abba***g" has a center of 5 and a radius of 2. Thus, to find the longest palindrome, go through each center and find the largest radius at that center. The largest radius is equal to the length of the longest string that occurs before and after the center, and since we have the reverse string s' it's equal to the longest string after the center and after the position of the center in s' (a character on the left side of s appears on the right side of s'). In order to calculate that, create a generalized suffix array on s and s' and compute the LCP array. Prepare the LCP array for constant time RMQ, and now the longest prefix of two suffixes is just the LCP value between them.

Odd length palindromes (like "aca") are computed similarily, just offset the "reverse position" by 1.

8. Leetcode LPS: Equivalent to the previous problem, but return the palindrome.

   Keep track of the starting index i along with the size l of the longest palindrome. Return the substring $s[i - \lfloor \frac{l}{2} \rfloor : i + \lceil \frac{l}{2} \rceil]$ (the radius is symmetric to the center).

9. Leetcode palindromic-substrings: Count the number of palindromic substrings.

   Since palindromicity is monotonic, finding that the longest palindrome has a radius of R implies that the center also has valid palindromes of radius $r = 1, 2, 3, \ldots, R$. Do the aforementioned algorithm but add the radii up instead of finding the max.

10. SPOJ STAMMER/USACO Gold December 2006 Milk Patterns (POJ link): Direct application of longest k-th repeated substring.

    Solution: Build a suffix tree on the string. Then, a substring will repeat k times if it has at least k leaves in its subtree. Run a DFS on the suffix tree, counting the number of leaves under each node and maintaining the string length up to that node. Pick the node with the largest string length that has at least k occurrences.

11. SPOJ LCS/SPOJ ADAPHOTO: Direct application of longest common substring.

    Solution (using suffix arrays): Construct a generalized suffix array on the two strings. Then, substrings which appear in both strings must be adjacent in the suffix array and has length equal to the LCP of the two suffixes. Scan through the suffix array, and if the pair is valid (each suffix starts in a different string) then consider whether the LCP value between them is larger than the current best.

    This can apparently be generalized to multiple strings in $O(n)$ if one is somehow able to determine whether a range contains suffixes from all of the strings in $O(1)$ and keeps track of the minimum LCP value in a range using a sliding window technique with a structure like a monotonic queue.

12. SPOJ LONGCS/SPOJ LCS2 Direct application of LCS for more than two strings.

    Solution (using suffix trees): Construct a generalized suffix tree on all the strings. Then, valid substrings must have leaves from all the strings and has a length equal to the string depth up to that node. Run a DFS on the suffix tree, numbering each leaf based off which string it came from and counting the number of distinct IDs under each node while maintaining the string length up to that node. Pick the node with the largest string length that contains all strings.

    How do you count the number of distinct IDs under a node, given each leaf has an ID? The count for a particular node will be the sum of the count of each of its children, minus duplicates.

    To keep track of the number of duplicates, consider two leaf nodes. Any node lower than the LCA of these two leaves will not have one of them in its subtree and thus will not have duplicates resulting from these leaves, and any node higher than the LCA is inductively corrected if the LCA is correct. Therefore, you only need to subtract one from the count of the LCA.

    The final algorithm is while running the DFS, keep track of the last leaf node seen for each ID. On processing a leaf node, subtract one from the LCA of the current node and the last node seen. For a non-leaf node, add up the counts of its children.

13. USACO Platinum December 2017 Standing Out from the Herd

7 Past Lectures

1. "Suffix Arrays and Longest Common Prefix" (Daniel Wisdom, 2019)
2. "Suffix Arrays and SA-IS" (Charlie Gunn, 2018)
3. "Suffix Arrays" (Justin Zhang, 2017)

8 Works Cited

1. Stanford lectures
    (a) Suffix Trees
    (b) Constructing Suffix Arrays and Trees
    (c) Constructing LCP array
    (d) Applications of Suffix Trees
    (e) Suffix Trees in Competitive Programming
    (f) Burrows-Wheeler
2. Original paper on SA-IS
3. Original paper on Kansai's algorithm for LCP