02-613 Week 9

Algorithms and Advanced Data Structures

Aidan Jan

October 29, 2025

Suffix Tries

Problem: Given a large known, fixed **text**, e.g. a dictionary, and many unknowns of changing search queries, how do we design a data structure that can quickly search up terms in the structure, but also substrings?

- For example, if "uninterested" was in the dictionary, and we wanted to search for "interested" or "interest", they should both show up.
- Using any of our structures so far, there is not an efficient way to do this.
- What we *can* do is make a tree, where each node represents a letter, where every suffix is a path from the root to some leaf node.
- We will use the string $S = ABAABA\$$. ($ is a terminus character to mark the end of the string.)

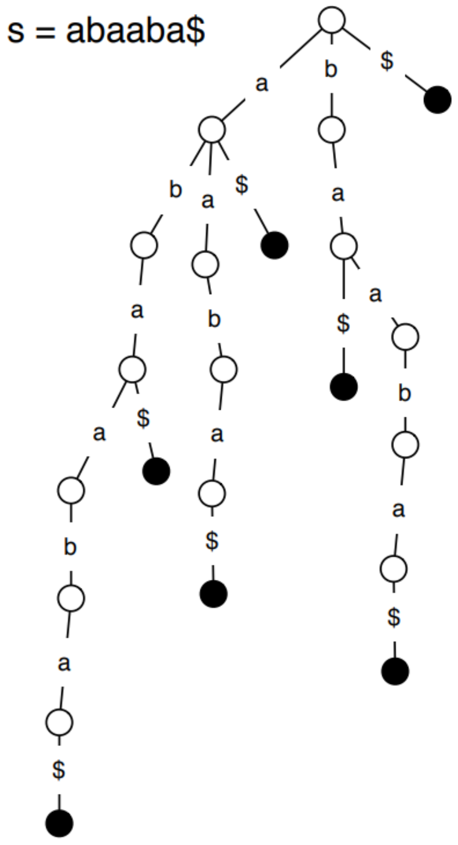

Image by Dr. Carl Kingsford @CMU

This structure allows for searching for any word in our dictionary, but also any suffix.

- This is known as a **trie**. (Not a tree, since trees don't allow internal nodes with only one child.)

Runtime of Tries

The runtime of a trie is $\Theta(\Sigma)$, where Σ is the size of our alphabet. For the English alphabet, $\Sigma = 27$ (26 letters, plus an end-of-word marker, \$.)

- Similarly, a DNA sequence would have $\Sigma = 5$ since it includes $A, C, T, G, \$$.
- We add a \$ to be able to distinguish when a valid string ends. If we want to find if a query is a suffix of our text, then the terminators become very important.

Searching in Tries

To search for a term q in our text T, we simply follow the path down from the root, since every substring would have a unique path from the root.

- If we want to know if q is a suffix of T, we can do that too since q would be a substring of T.
- If we want to know the number of times q is in T, then we follow the path down for q, and at the last character, count the number of leaf nodes in the subtree.
- To find the longest repeated substring, we return the deepest branching node.
- To find the lexicographically smallest suffix, from the root we follow the tree down always following the branch with the smallest character, assuming \$ is less than any other character in our alphabet.

Suffix Trees

Consider the string s $= a^n b^n$, where $n \in \mathbb{Z}$. Basically, a library of string containing a series of a followed by a suffix if b. The trie would look like:

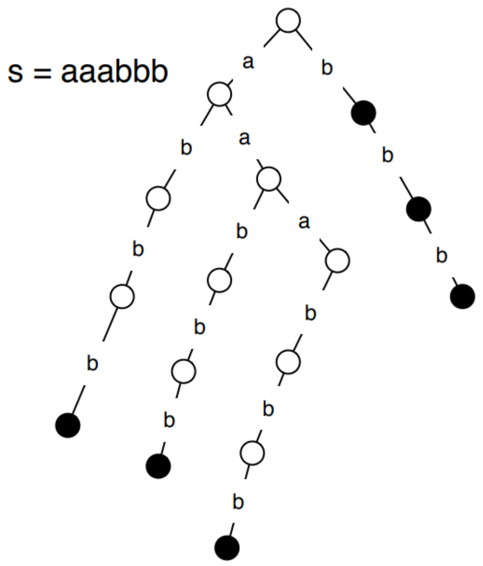

Image by Dr. Carl Kingsford @CMU

In this case, once we get into the string of b's, there is only one long branch. However, if we wanted to encode something like the human genome (with 3B base pairs), we can't store $O(n^2)$ nodes like we do here. To make this more space efficient, we can compress this.

Key Points of Construction

- The number of leaves we have after compression is O(n) with the above example.
- All internal nodes are branching.
- If instead we store string indices rather than the actual string on every edge, we don't have to build every substring in our tree either, which brings storage down to O(n).
- Tries are useless in practice, we usually build the suffix tree directly.

Building a Suffix Tree

- Start from the empty string (root)
- Add one character at the beginning of the string, and add the suffix link too
- Add the next character to the end of every suffix following the suffix link back to the root
  - This is equivalent to adding the next character to the end of every existing suffix.
- Repeat until entire string is represented in the tree.

```
current suffix = longest suffix
do:
    add child labeled S[i] to current suffix
    follow suffix link to set current suffix to next
    add suffix link to new node
while not every letter of string is in the tree
```

For this class, construction is not discussed much. Use Ukkonen's Algorithm for construction in O(n) time.

Finding Substrings

Suppose we want to find the longest common substring between S and a query.

- First, let's find the longest common prefix. To do this, we simply start with q, and walk down the suffix tree as far as we can.
- Now, extending this to all substrings, we can change the "starting point" by following the suffix links.
  - If we get to a point where we cannot extend our prefix, we follow the last suffix link (which effectively moves the starting point forward by a few positions), and continue iterating!
  - This way, the algorithm will take at longest O(m), where m is the length of the query, since we never backtrack in the query string. This beats the naive method which is O(m^2).

For example, suppose we want to find the longest common substring between $q = BBAA$ and $S = ABAABA$

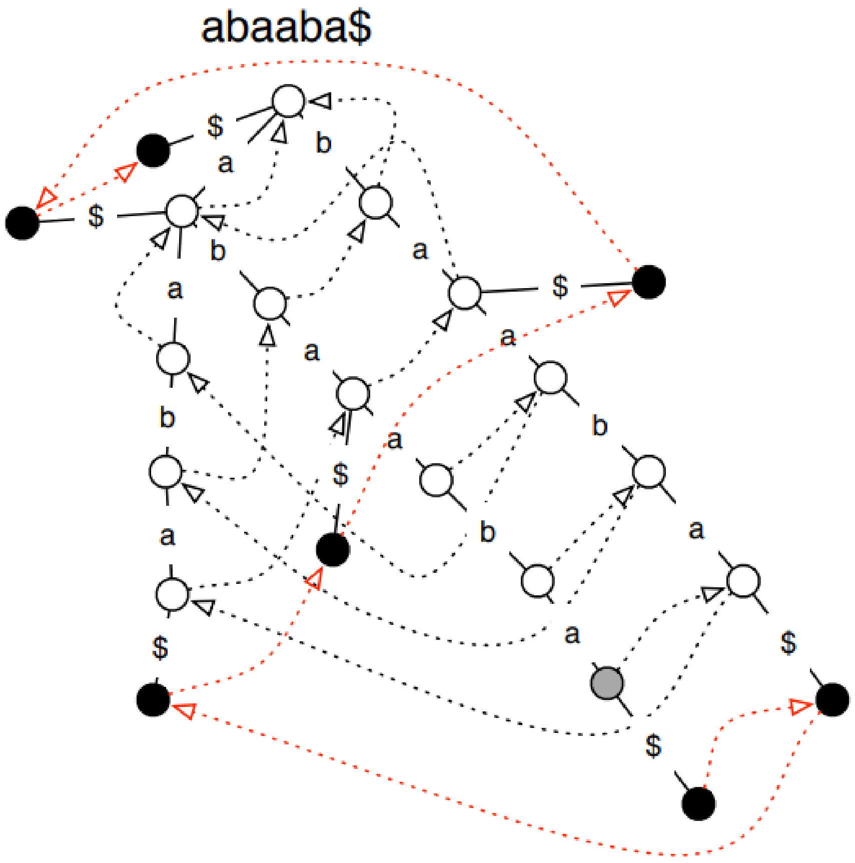

Image by Dr. Carl Kingsford @CMU

Suffix Tree of Multiple Strings

Given a set $S = \{s_1, s_2, \ldots, s_n\}$ sequences from an alphabet, represent as a suffix tree. How do we construct this?

- There is an easy way: concatenate all the strings.
- Consider $S = \{aat, tag, gat\}$. We can build an $S' = aat\$_1tag\$_2gat\$_3$.
- Now, build a suffix tree using Ukkonen's on S', iterate through each node, and delete everything past the first terminal character.
- This runs in O(n), where n is the total length of all the sequences.

For the above example, we will get the result:

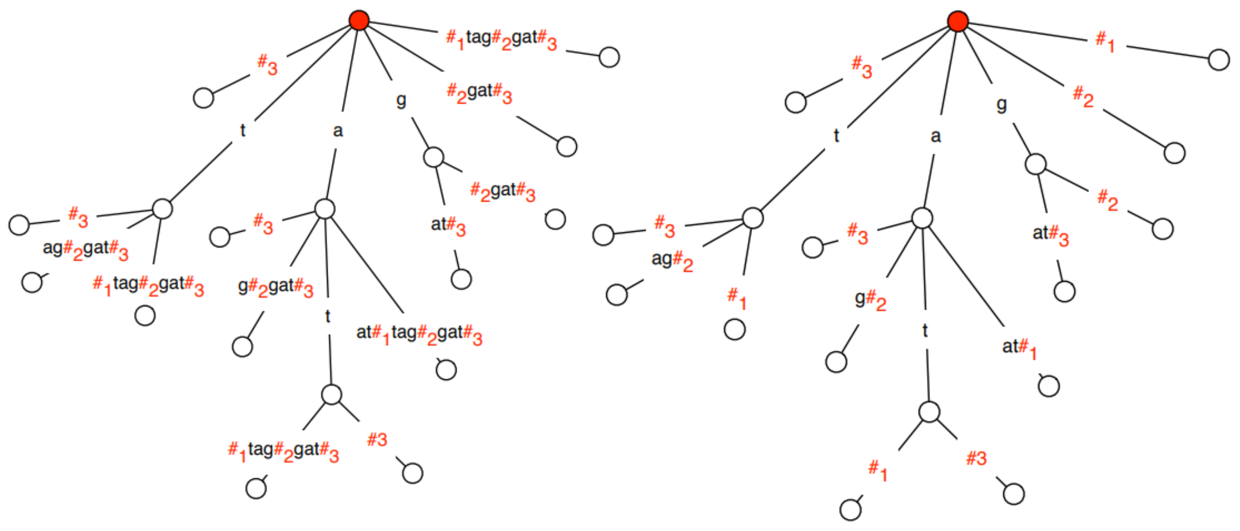

Image by Dr. Carl Kingsford @CMU

Longest Common Substring

Building upon the suffix tree of multiple strings, how do we find the longest common substring between two sequences?

- We can do this also in O(n) time, where n is the length of the two strings combined.
- Build a suffix tree containing both strings of interest, and run a DFS.
- Propagate the value of each leaf node (always a terminus, $\$_i$) backwards, and the deepest node that can reach both terminus would be the longest common substring.

We can also use this strategy to find all the strings in a database $\{S_1, S_2, \ldots, S_n\}$ that contain any query, q.

Longest Common Extension

Suppose we are given strings S and T. We will be provided with many pairs (i, j) as queries, and we want to be able to quickly find the longest substring of S starting at i that matches a substring of T starting at j.

- Build a suffix tree containing both S and T.
- Preprocess the tree so that the lowest common ancestors (LCA) can be found in constant time. (We won't prove this, but it can be done in O(N), where N is the total length of the two strings.)
- Create an array mapping suffix numbers to leaf nodes.
- Given query (i, j),
  - Find the leaf nodes for i and j
  - Return string of the LCA for i and j
- Query can be found in O(1).

LCE for Palindromes

A palindrome has the property where the left half is equal to the reverse of the right half. To find **maximal** palindromes in S (e.g., given a center point, find the longest palindrome that contains it), we construct the reverse string, S^r.

- Notice that if given a center point i, the right half would be a substring beginning at i on the forward string, and the left half would be a string starting at $(n - i)$ on the reverse string.
- With this, we can create a suffix tree containing S and S^r, find the LCE given point i, and the string going from the root to that intersection would be a palindrome.
- The total runtime is O(n).

What if we allow mismatches in our palindrome?

k-mismatch Palindromes

This is the same problem, except this time, we allow for k-mismatches. For example, $TACOBAT$ has 1 mismatch, since when comparing the left half to the reverse of the right half, there is one mismatch. To do this,

- Find the maximal palindrome from the center i. Suppose it has length $2j$, so the right of the string is at position $i + j$ on S, and the right begins at $n - i - j$ on S^r.
- For each k mismatches, we can add one "error" letter, and repeat the process with another LCE, with our query being $i + j + 1$ on S and $n - i - j - 1$ on S^r.
- Therefore, for k mismatches, it will take O(kn).