Lecture 26

String matching, Aho-Corasick, and suffix trees.

Mon Dec 5, 2022
Reading: Weiss 12.4
Recording
Scribbles

A pdf version of these notes is available.
This lecture contains material from these lecture notes on tries and suffix trees.

String matching

In the multi-string searching problem we're given string $T$ along with $k$ nonempty strings $P_1, \ldots, P_k$ called patterns. We want to find occurrences of the patterns in $T$. The problem is important for building antivirus databases or finding occurrences of certain keywords on a webpage. To help us discuss this problem, we'll use $m = |T|$ and $n = |P_1| + \cdots + |P_k|$, and we'll assume $|\Sigma| = O(1)$.

Once again, there's an "obvious" but suboptimal solution: Run KMP once per pattern $P_i$. We'll need to look at $T$ $k$ times, so the total running time will be $O(km + n)$. We'll look at two ways to reduce the running time to $O(m + n)$. Both involve some preprocessing on either the pattern strings $P_i$ or the string $T$. I'm mostly going to focus on the data structures involved and how they're used, but not so much on constructing the data
structures, because constructing them quickly is actually really really tricky. We'll also just focus on figuring out whether or not there exists a match instead of trying to list them all.

**Aho-Corasick**

We'll start with an algorithm that preprocesses the set of patterns $P_1, \ldots, P_k$. We only have to do this preprocessing once; afterword, we spend $O(m)$ time searching the string $T$. In fact, we can reuse our work to perform new searches on a new string $T'$ in only $O(|T'|)$ additional time.

Let's start with what we know: If we'd like to figure out whether a prefix of $T$ matches some pattern, then we can create a trie for each of the $k$ patterns and then search for $T$ in the trie. Now, suppose we're doing this search, we reach the node for some string $w$, but now there is no link to follow for the next character of $T$. After this point, none of the patterns are going to match starting at the first character of $T$. We could start a new search from the second character of $T$, but that would be kind of wasteful; after all, we know exactly which characters we're going to be looking at again, and we may not even make as much progress through the trie as we did when we found the path for $w$. What we'd like to do is 1) figure out how many characters of $T$ we need to skip to make meaningful progress in the trie, and 2) quickly figure out where we'll end up in the trie after we do so.

A **suffix link** is a link from a node for some string $w$ to the node corresponding to the longest proper suffix of $w$ in the trie. All nodes except for the root can have a suffix link (the empty word has no proper suffixes). It turns out we can build the trie and compute all the suffix links in only $O(n)$ time. We won't discuss how, though, in the interest of time.

Now suppose we want to figure out if one of our patterns appears in $T$. We act as if we're searching for $T$ in our trie, but whenever we cannot move forward, we follow a suffix link before processing the next character. At any point during this search, we've considered a prefix of $T$, and the current node represents the largest suffix of that prefix that appears in the trie. We can add a boolean to each node to say whether such a suffix contains one of our patterns, and if we ever hit a node with a true boolean, we can report success.

For running time, we can only follow as many suffix links as characters we process, so the whole search takes $O(m)$ time. Hurray! This algorithm is known as the **Aho-Corasick** string matching algorithm.

**Suffix trees**

The Aho-Corasick algorithm takes the set of patterns $P_1, \ldots, P_k$ and constructs a data structure from them. Given any string $T$, the data structure can answer if any of the patterns appear in $T$ in only $O(|T|)$ time (and with a bit more work, we can return the positions for the matches in time proportional to the number of positions). Computational genomics often involves the opposite problem, though. There is a huge static database with very large strings and we want to work with many patterns that are yet unknown. Can we build the data structure for $T$ instead that takes patterns as its input?
The solutions derives from the following observation: if $x$ is a substring of $w$, then $x$ is a prefix of a suffix of $w$ (in particular, the suffix that starts where $x$ begins). We already know a good data structure for prefix searching, the trie!

A suffix trie of $T$ is a trie over all the suffices of $T$. Given one, we can check whether the patterns exist in $T$ by just searching for each of them in the trie. We need not worry about the "real string" booleans, because we just want to know if the patterns are prefixes of some suffixes. The time for the searches is merely $O(n)$. To make things a bit cleaner, its common to append a new character $\$\$ to $T$ and build the suffix trie over $T\$$. Now all suffixes end precisely at the leaves of the trie.

The only real issue with our approach is the size of the suffix trie. Consider the string $a^m b^m \$$. Its suffix trie contains a path of $m$ as from each of which dangle a distinct path of $m$ bs. That's $\Theta(m^2)$ nodes total! We'd need the space for all those nodes and at least that much time to construct them.

### Suffix trees

But we've seen a solution to our problem already. Use a Patricia trie! A suffix tree for string $T$ is a Patricia trie of $T\$ where each leaf is labeled with the index where the corresponding suffix starts in $T\$$. The suffix tree has exactly $m + 1$ leaf nodes, and unless $T$ is empty, every internal node has at least two children. Therefore, the total number of nodes is $\Theta(m)$.

Now, if we explicitly write out the substrings represented by each edge, then we haven't actually saved space (asymptotically). Therefore, each edge is labeled with a consecutive range of characters from $w$. So we can just label each edge with the start and end positions of its substring. The total size ends up being $\Theta(m)$. The suffix tree can also be constructed in $\Theta(m)$ time, but wow that algorithm is complicated.

And now back to our problem of string matching. Given the patterns and $T$, we build the suffix tree for $T$ in $O(m)$ time. Then, we check all the patterns using the suffix tree in $O(n)$ time. That's only $O(m + n)$ time total. And we still have our suffix tree lying around in case we want to check any other patterns in the future.