Lecture 25

Strongly connected components, tries, and string matching.

Wed Nov 30, 2022
Recording
Scribbles

A pdf version of these notes is available.

This lecture contains material from Chapter 6 of Algorithms by Jeff Erickson along with these lecture notes.

Strong connectivity

So we've said some things about connected components in undirected graphs. But what about directed ones? Recall, a directed graph is weakly connected if the underlying undirected graph that ignores edge directions is itself connected. The directed graph is only considered strongly connected if there is a path from every vertex to every other vertex.

A strongly connected component or strong component is a maximal subset of vertices where the induced subgraph, the subgraph including exactly those edges between members of the component, is strongly connected. Finally, the strong component graph $scc(G)$ is another directed graph made from directed graph $G$ where we contract each
strong component to a single vertex and collapse parallel edges. This graph is always a DAG; a cycle would imply we could do more contracting.

We can find the strong component of a single vertex \( v \) pretty easily. First, we do a search from \( v \) to find those vertices reachable from \( v \). Then, we do a search from \( v \) in the reversal of \( G \), the graph made by reversing all the edges. Vertices reachable in the reversal are exactly those that can reach \( v \). The vertices that are reachable from \( v \) and can also reach \( v \) are exactly those in the strong component of \( v \). Doing these two searches takes \( O(|V| + |E|) \) time. But what if we want to learn all the strong components that quickly? Doing so is kind of tricky.

It turns out we need to really take advantage of depth-first searching. First off, we might notice the following: Consider any strong component \( C \). If we do a \texttt{dfsAll} the vertices of \( C \) all appear as a subtree of the DFS forest (here, subtree just means a subset of edges in the forest that are weakly connected). We'll call the root of this subtree the root of \( C \). A sink of a DAG is a vertex of out-degree 0. All DAGs have at least one sink. A sink component of \( G \) contracts to a sink in \( \text{sc}(G) \). A sink component has no edges leaving for other strong components, so its subtree in the DFS forest consists of its root and all descendent vertices. Further, the vertices reachable from any member of a sink component are exactly the members of that sink component.

Now, consider the following high level algorithm: We find a vertex \( v \) in a sink component (somehow), label everything reachable from \( v \), remove the component from \( G \), and then recursively label the remaining components. But how will we quickly find each candidate vertex \( v \)?

The solution lies in an algorithm by Kosaraju and Sharir. While finding a single vertex of a sink component is somewhat tricky, it turns out it's easy to find a vertex in a source component. Just use the last vertex in a postordering of \( G \). The reasoning here is essentially the same as how we can topologically sort a DAG by doing a reverse postordering.

OK, we're almost done. It's not too hard to see \( \text{rev}(\text{succ}(G)) = \text{sc}(\text{rev}(G)) \). So, the last member of a postordering of \( \text{rev}(G) \) is in a sink component of \( G \) itself. In addition, we can just keep reusing a single postordering of \( \text{sc}(G) \) by removing vertices from it as they're found in each successive sink component.

So here's the final algorithm. We compute a postordering of \( \text{rev}(G) \), adding each vertex to a stack so we can easily look at them in reverse order. Then, we do a \texttt{dfsAll}, with the outer loop popping vertices from the stack. The total running time will be that of two calls to \texttt{dfsAll}, so \( O(|V| + |E|) \).

**Tries**

For the rest of the day, we're going to go back to data structures to discuss some useful structures involving strings. To start with, consider the following problem: We have a collection of \( n \) strings/words over some alphabet \( \Sigma \) that we want to store in some kind of dictionary that allows you to do some of the standard operations like insert, remove, contains, and predecessor.

The obvious solution to our problem is to store the strings in a balanced binary search tree.
where the strings themselves are the node keys helping us during a search. Any of those operations can be performed using only $O(\log n)$ comparisons. Unfortunately, comparing two strings of lengths $r$ and $s$ takes $O(\min\{r, s\})$ time. If $M$ is the maximum string length, the total time spent doing comparisons in a binary search tree ends up being $O(M \log n)$. Can we do better? Yes!

We're still going to use a tree, but now we'll have individual letter comparisons guide our path through it. A trie (short for re-trie-val, but pronounced "try" for some reason) is a tree where each node stores a boolean along with an array of links, one for each character in alphabet $\Sigma$. Each node $x$ corresponds to a string given by the links used going from the root to that node. The boolean is set to true if that node's string is actually a member of the dictionary. As to not waste space, we'll guarantee leaves always have their boolean set to true.

Let's assume $|\Sigma| = O(1)$. Given string $w$, we can perform contains in only $O(|w|)$ time by just following the appropriate pointers until we either find $w$ or hit a leaf.

predecessor isn't much harder.

insert is simple: just act as if you're doing contains, adding nodes as necessary whenever you reach a leaf. The the boolean to true when you reach the node for the input word.

remove is essentially the opposite: first find the node for the word you want to remove and set its boolean to false. Then while you're at a false node with no children, remove the node and continue with its parent.

### Patricia tries

A trie with $N \leq Mn$ nodes requires $\Theta(N \cdot |\Sigma|) = O(Mn \cdot |\Sigma|)$ space. This is a bit wasteful if the strings we're storing have long spans along which they can't be distinguished. The most extreme example would be a trie over a single word (say, "antidisestablishmentarianism") that is very long. We end up with $M$ nodes, even though $n = 1$.

A Patricia trie (also known as a radix trie) attempts to fix this issue by merging all nodes with a single child into their parent. Links are labeled with the entire string that would be traversed if the nodes were not merged. We still end up representing each string, but now each node has at least two children, meaning there are at most $2n - 1$ nodes total.

### String matching

Tries are a data structure for storing lots of strings. Starting now and on Monday, we're going to see how they can be used to find hidden patterns within a string.

In the basic string matching problem, we're given some text string $T$ and a nonempty string $P$. Our goal is to find all occurrences of $P$ within $T$. It's what happens when you type ctrl-f in your word processor.
One solution is to go through $T$, and for each character $c$ spend $O(|P|)$ time checking if $P$ starts at $c$. The issue here is that there are $|T|$ characters to start from, so this algorithm takes $O(|P| \cdot |T|)$ time.

One faster solution is the Knuth-Morris-Pratt algorithm. We do some preprocessing over $P$ to build a data structure of size $O(|P|)$ in $O(|P|)$ time. We can then scan $T$ using this data structure to finish the search in $O(|T|)$ additional time. So $O(|P| + |T|)$ time total. But it turns out KMP is really a special case of a more powerful algorithm for a more general problem.

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.