Lecture 24

Connected components, Borůvka's algorithm for minimum spanning trees, and strongly connected components.

Mon Nov 28, 2022

A pdf version of these notes is available.
This lecture contains material from Chapters 6 and 7 of Algorithms by Jeff Erickson.

Connected components

Today, we're going to discuss connected components. Recall, in an undirected graph, a connected component is a maximal subset of vertices such that there is a path between every pair of vertices in the subset. A basic question we can ask is, given an undirected graph $G = (V, E)$, what are its connected components?

Earlier, we saw the algorithms breadth-first search (BFS) and depth-first search (DFS). Those algorithms, along with others like them, take a single vertex $s$ and mark everything reachable from $s$. If the graph is undirected, then they'll have to mark everything in $s$'s connected component. We also saw, with DFS in particular, that it is sometimes useful to loop over all the vertices, performing a search every time you encounter one that has yet to be marked. Since we're marking every vertex in each such vertex's connected component, each search we initiate finds exactly those vertices in a connected component
we have yet to explore. By passing around a counter variable, we can even label the vertices during a search so that we can make sure two vertices have the same label if and only if they belong to the same component. The counter can also be used to, ahem, count how many components there are.

```java
int countAndLabel(Graph g) {
    int count = 0;
    for each Vertex v in g {
        if (!v.visited) {
            labelComponents(v, count++);
        }
    }
    return count;
}

boolean labelComponent(Vertex v, int label) {
    v.visited = true;
    v.label = label;
    for each Vertex w adjacent to v {
        if (!w.visited) {
            labelComponent(w, label);
        }
    }
}
```

This counting and labeling algorithm is another slightly dressed up DFS, so it again runs in $O(|V| + |E|)$ time.

**Borůvka's algorithm for MST**

The components of a graph, and their size, shape, etc. can potentially tell you a lot of information about whatever you're modeling with the graph. As we saw before break, they're also a necessary consideration when designing algorithms for computing minimum spanning trees. In fact, the `countAndLabel` method above is a useful subroutine for a third MST algorithm we didn't discuss last time.

For simplicity, let's assume we're given an undirected connected graph $G = (V, E)$ with distinct edge weights; i.e., no two weights are equal. Recall, the high level idea behind computing a minimum spanning tree is that we add edges to a spanning forest $F$ over several rounds. Any edge we add during a single round has to be a lightest edge leaving one of the components of $F$. If all edge weights are distinct, then there is exactly one lightest edge leaving each component.

In Prim's algorithm, we took the lightest edge leaving a single component that grew over the course of the algorithm. In Kruskal's algorithm, we looked at the edges in increasing order by weight, taking each one that left its endpoints' components. The idea behind Borůvka's algorithm is that instead of taking a single edge each round, we can just take all the lightest edges leaving all the components at once. We won't form a cycle doing so, because then the heaviest edge we added in the cycle wouldn't actually be the lightest edge leaving either of its endpoints' components.

The easiest and fastest way to implement this algorithm is to run the `countAndLabel` procedure once per round on $F$ itself, and then loop through all the edges, creating a list of those to be added using an array indexed by component labels.
ArrayList<Edge> boruvka() {
    List<Edge> mst = new ArrayList<Edge>();

    Graph f = graph from edges in mst;
    int count = countAndLabel(f);

    while (count > 1) {
        List<Edge>[] lightest = new List<Edge>[count];
        for each edge e { // e = uv
            Vertex u = e.getu();
            Vertex v = e.getv();
            if (u.label != v.label) {
                if (lightest[u.label] == null ||
                    e.weight < lightest[u.label].weight) {
                    lightest[u.label] = e;
                }
                if (lightest[v.label] == null ||
                    e.weight < lightest[v.label].weight) {
                    lightest[v.label] = e;
                }
            }
        }
        for (int i = 0; i < count; i++) {
            mst.add(lightest[i]);
        }
        f = graph from edges in mst;
        count = countAndLabel(f);
    }

    return mst;
}

Each iteration of the while loop spends constant time looking at each edge in addition to a single countAndLabel over the edges in the spanning forest. Therefore, each iteration takes $O(|V| + |E|) = O(|E|)$ time, because the graph is connected. But how many iterations can there be? Each iteration ends up reducing count by a factor of at least 2. In the worst case, the components combine in pairs, but there is a chance that we could end up creating the entire MST in a single iteration. We start with $|V|$ components, so the worst-case running time is $O(|E| \log |V|)$.

OK, but we've seen that running time twice already. What's the big deal? First off, that worst-case running time really is a worst-case. If you're lucky enough to finish in just a couple iterations, the running time could be as low as $\Theta(|E|)$. In fact, there's a slight variation of this algorithm that is guaranteed to run in $\Theta(|E|)$ for certain classes of graphs, including those that can be drawn in the plane without edges crossing. This algorithm is much easier to parallelize than the others if you have a large graph in a multicores or distributed system. Finally, theoretically faster algorithms than the ones we've seen in this class tend to be based on Borůvka's algorithm instead of the other ones.

In short, you really want to implement Borůvka's algorithm if you need to roll your own MST algorithm for some reason. The other two are mostly interesting for theoretical reasons and just commonly taught.

**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 $\mathcal{O}(|V| + |E|)$ 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 *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 $scc(G)$. 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 $rev(succ(G)) = scc(rev(G))$. So, the last member of a postordering of $rev(G)$ is in a sink component of $G$ itself. In addition, we can just keep reusing a single postordering of $scc(G)$ 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 $rev(G)$, adding each vertex to a *stack* so we can easily look at them in reverse order. Then, we do a *dfsAll*, with the outer loop popping vertices from the stack. The total running time will be that of two calls to *dfsAll*, so $\mathcal{O}(|V| + |E|)$. 