Homework 8

Sam Daitzman // DSA Spring 2020

Exercise 1: Single Minimum Spanning Tree

> Prove that if a graph $G = (V, E)$ has unique edge weights (i.e. $we1 \neq we2$ for any two edges $e1, e2 \in E$), then there is a single minimum spanning tree. In other words, there is only one optimal solution. **Hint**: Use a proof by contradiction and suppose that there are two optimal spanning trees T_1* and T_2*.

Want to show: if we have some graph G that has unique edge weights, there is exactly one spanning tree.

Suppose there are two different (non-identical) minimum spanning trees T_1* and T_2* across G. We can imagine that these two spanning trees are different by only a single minimum-weighted edge element. Within a minimum spanning tree, adding any single incorrect edge must result in a cycle within the tree because it will connect two paths along the tree, necessarily creating a closed loop and violating tree structure. This means that T_1* and T_2* contain some otherwise matching elements e_1 and e_2 that are not equal to result in this structure. Suppose $e_1 \geq e_2$, which can be arbitrarily swapped. Since swapping just this edge results in an unequal total weight $T_1* \neq T_2*$, one of our minimum spanning trees is not a minimum spanning tree, because it is not the minimum. This is a contradiction!

Exercise 2: Breadth-First Search

> Suppose that you are given an unweighted graph $G = (V, E)$ and a node $i \in V$. One way to find the shortest path from i to all other nodes $j \in V$ is to run breadth-first search from i. In particular, for every node v added to the queue when processing u in BFS, we mark u as v's parent. This induces a tree of shortest paths from i, as shown below in red for $i = 1$. In this case, the shortest path from 1 to 5 is 1-3-5.

Exercise 2a: BFS Proof by Contradiction

> Use a proof by contradiction to show that BFS keeps track of the shortest path from i to all other nodes. **Hint**: Consider the closest node v to the source i whose BFS path is not a shortest path.

Want to show: BFS can track the shortest path from any node i to each other node.

Consider the closest node v to the source i whose path is not the shortest path. Breadth-first search (BFS) descends all paths one step at a time by enqueueing new encountered nodes. Supposing that the node v is closest to the source i, it must be connected by a series of nodes. Traversing back up the chain of jumps, if v has a BFS path that is not the shortest path, this means that the distance to its parent u or some iteration along the chain of jumps must be greater than the minimum possible. But the previous node must have found the next minimum node because of how BFS enqueues nodes, which is a contradiction!

Exercise 2b: Weighted to Unweighted

> Given a weighted graph $G = (V, E)$ in which each edge weight $w_e \geq 0$ is an integer, explain how to convert G into an unweighted graph $G' = (V', E')$ such that finding the shortest path from i to j in G' gives you the shortest path in G. How many vertices and edges does this new graph have?

If I understand correctly, this is an attempt to use an unweighted graph G to calculate the shortest path between some start arbitrary nodes $i, j \in V$. Of course, an unweighted graph is simply a graph with $1 = e_n \in E$. To effectively replicate this in reverse, we can create an unweighted graph with $w_{u \to v} - 1$ additional `WeightNodes` between each normal node, where $w_{u \to v}$ is the weight of the edge between the nodes u and v. As long as we ensure that `WeightNodes` are not considered identical to regular nodes $\in V$, we can run an existing BFS-based shortest-path algorithm to find the shortest connection between i, j.

The total number of vertices will be tend to be bounded by the sum of all weights $\sum e_1, e_2 ... e_n = N$. The total number of edges will be $N - 1$, so in O-notation it will still be bounded by N.

Exercise 2c: BFS Shortest Path

> Using your construction above, explain how to find the shortest path from i to all other nodes j in a weighted graph $G = (V, E)$ using BFS. What is the runtime of your overall algorithm? How does it compare to the runtime of Dijkstra's algorithm?

Dijkstra's algorithm runs in $O(log(|V|) * |E|$. To combine the previous two concepts and use BFS on an unweighted graph, we can construct a graph with multiple `WeightNode` nodes between each original node and modify BFS to operate as constructed above, traversing the graph and adding distance annotations to real nodes as needed. This will tend to be bounded by $O(|V| * N)$ because the BFS traversal will need to encounter all nodes.

At worst, with every weight very high, this algorithm could perform much worse than Dijkstra's algorithm, on the order of the product of the high weight and the number of nodes. For graphs with extremely low or mostly-1 weighted edges, this might have acceptable performance.

Questions to Follow Up On

- In reading notes to try to answer Exercise 1 I found some mention of a modified version of Prim's algorithm being used to prove that there's only one minimum spanning tree. How does that work?
- I feel very shaky about how breadth-first search works intuitively. Are there good ways to visualize/represent/draw this for practice?
- Is there a more rigorous/formal way to derive the Dijkstra's/BFS comparison above?