CS170 Discussion Section 4: *2/15*

1 Coraysoda

The land of Coraysoda has a network of settlements, S, and roads, R. There is always at least one path from any settlement to any other settlement. Most roads have a specified tax cost T_r that you must pay to traverse it. All roads are bidirectional. It is helpful to consider this as a graph $G = (V, E)$, where $V = S$ and $E = R$.

- The king and queen of Coraysoda have built massive golden statues of themselves in two different settlements. To encourage people to travel by it, any paths that visit either of the two statues will have half of their total trip cost reimbursed (visiting both statues does not reimburse more). Given a starting settlement A, and a destination settlement B, find the cheapest route there.

  First, we should represent this problem as a graph with vertices S and undirected edges R, with weights T_r.

  To solve this question, you can run Dijkstra's a few times. First, you run it normally, to determine the shortest distance from A to B, without considering the statues. Next, find the shortest distance from A to K, and K to B (K is the settlement with the king statue, and Q for the queen). The sum of these two distances divided by half (to account for the reimbursement), is the alternative cost if we take the shortest path that visits the king's statute. Similarly, we can find the shortest distance from A to Q, and Q to B. The sum of these two distances divided by half is the alternative cost if we visit the queen's statue. The minimum cost path of the three potential paths is the cheapest route. We can run Dijkstra's 5 times to get all the necessary distance values, which is the same asymptotic runtime as running it once.

  If we're being clever, we can also run Dijkstra's just twice, once from A as the source node, and once from B as the source node (remember the graph is undirected) to gather the same data.

  The runtime of unoptimized Dijkstra's is $O((|E| + |V|) \log |V|)$. In our case, $|E| = |R|$ and $|V| = |S|$.

- A few settlements are protesting the Coraysoda government by blocking off all the roads coming in and out of the settlement. You want to determine which settlements have been "disrupted". We've defined a settlement is disrupted if it now can only reach less than half of the settlements it used to be able to reach. Can you use the result of the previous part? Your runtime should be $O(S + R)$.

  Analysis: For a settlement to be "protesting" means that the corresponding vertex in the graph is isolated (all of its edges deleted/disabled).

  You cannot use the result of the previous part. Since our graph has changed, the shortest path information is no longer up to date. You can solve this question by doing a 170-style DFS (do not terminate until all vertices have been reached). The result will be multiple DFS trees. All settlements that are protesting are not connected to anything, so you can imagine they will each be their own DFS tree of size 1. As for

non-protesting settlements, they will be in the same DFS tree as anyone who has a valid path to them. Previously, any settlement could reach any other settlement, so if a settlement can now only reach less than $\frac{|S|-1}{2}$ other settlements, it is "disrupted". Therefore, any DFS tree of size smaller than $\frac{|S|}{2}$ can only reach themselves means all vertices in this DFS tree are "disrupted". All of these computations can be piggybacked onto a DFS search. The runtime of DFS is $O(|V| + |E|)$.

- Finally, all the riots are over, and the road network is back to normal. But we want to upgrade the road system. A road can be upgraded into a cutting edge paved road, for a cost of $2T_r$. A paved road is free to traverse, still bidirectional, and allows traversal via car. While spending the least amount possible, which roads should you upgrade to connect every settlement via car? You should make it so that starting from any settlement, it is possible to reach any other settlement without getting out of the car.

  Again, we work with the graph. Something that would accomplish the job at great cost is to upgrade every single road. This is too expensive. Even upgrading the shortest path between any two settlements is also not the cheapest solution. The minimum cost to create a network that spans all the settlements is what we want: the MST is the best solution.

  If we upgrade the MST of the graph, then ensures that all settlements have paved road paths to every other settlement (although these paths may not be direct). And since this is the minimum spanning tree, the cost to upgrade all these roads is also minimal. So we should upgrade all the roads that are in the MST of the graph.

  To find the MST, we could run Kruskal's algorithm or Prim's algorithm. Prim's can run in $O(|E| + |V| \log |V|)$, and Kruskal's can run in $O(|E| \log |E|)$.

2 Special Case Shortest Path

Consider a connected graph G whose edge weights are always 1 or 2. Design a shortest path algorithm to compute the shortest path from the starting vertex S to the destination vertex T. Your runtime should be $O(|E| + |V|)$.

BFS can find shortest paths, if all edges are the same weight. For this particular graph, we can expand each edge of weight 2 into two edges of weight 1, with an additional dummy vertex in the middle. If the shortest path runs through here, this means it runs through the original edge of weight 2. So we can run BFS from S, and return the path it finds to T. In the worst case, for every single edge, we swapped it out for two edges, and also added a dummy vertex. This can result in as much as $2|E|$ edges and $|V| + |E|$ vertices. Asymptotically, this is still linear time.

3 New road

Consider a directed positive-weighted graph $G = (V, E)$, a number k, and a candidate list of new positive-weighted directed edges E' that we can add. Unfortunately, we can only add add k edges to our network, or in other words, we need to choose a subset $S \subseteq E'$, with

$|S| \leq k$. In order to estimate how good these edges are, we consider the improvement of the shortest path from two predetermined nodes s, t from $G = (V, E)$ and $G' = (V, E \cup S)$ (that is, we consider how much shorter the path will become after adding all the new edges).

1. Show how to compute the best improvement efficiently when $k = 1$.
2. Show how to do this efficiently for every other $k > 1$ (optional, explain how to get the best set S)

We can use the same algorithm for both parts. However, there is an easier solution to part (a) that won't work in part (b)

1. Run Dijkstra normally from s (call these distance $dist(s, v)$) Reverse all the edges and run Dijkstra from t (call these distances $dist(v, t)$). Then, the original cost is $dist(s, t)$, while the best edge is $argmin\{(u, v) \in E', dist(s, u) + weight(u, v) + dist(v, t)\}$ which we can return.
2. The idea is to create a new graph with k levels. Create the graph $G' = (V', E')$, with $V' = V \times \{0, 1, \ldots, k\}$. For each edge $(a, b) \in E$ with cost c, create $k + 1$ new edges $((a, i), (b, i))$ also with cost c for each $0 \leq i \leq k$. For each edge $(p, q) \in E'$ with cost r. create k new edges $((p, i), (q, i + 1))$ also with cost r for each $0 \leq i < k$. Now, we claim that comparing the shortest path between $(s, 0)$ and $(t, 0)$ will give us the original shortest path cost, while the shortest path with the new edges is the shortest path between $(s, 0)$ and one of the $k + 1$ nodes $(t, 0), (t, 1), \ldots, (t, k)$ (i.e it's possible that adding new edges may not actually decrease the shortest path).

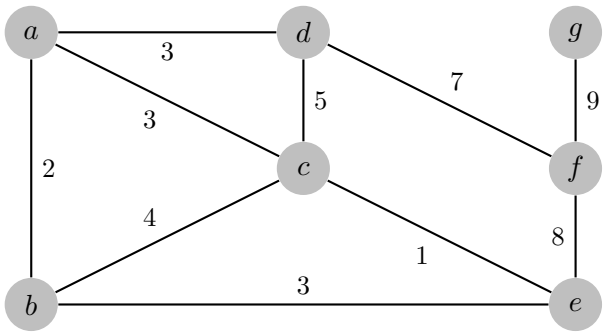
   We now run Dijkstra's starting from node $(s, 0)$. First, we claim that the shortest path from $(s, 0)$ to (v, i) is actually just the shortest path from s to v using exactly i edges from E'. Clearly, this is true when $i = 0$ since we are not introducing any edges going into the topmost level. Now, if we look at our edges, we notice we only ever go down a level whenever we traverse an edge from E', thus this claim is true.

   Now, we can see that $(s, 0)$ to $(t, 0)$ is indeed the shortest path, while $(s, 0)$ to (t, i) is the shortest path using i different edges, thus, we can take the minimum, then subtract the first shortest path from it to get the best improvement. We can see that we increase our graph's size by a factor of k, thus we can see the running time of this is on the order of running Dijkstra's k times.

   To return the best set, we can just backtrack from the best (t, i) and look at which edges we used to go down levels and return that set of edges.

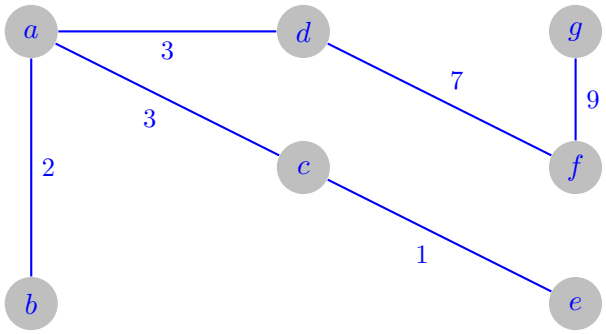
4 Kruskal

Use Kruskal's greedy algorithm to find the minimum spanning tree of the following weighted graph.



Kruskal adds the following edges in order to the MST: (c,e), (a,b), (a,d), (a,c), (d,f), (f,g) Note that there are other valid solutions here.

The result MST looks like:



If you'd like to demonstrate Prim's algorithm on this graph, here's one possible solution. We make an arbitrary choice of vertex a to start the algorithm. We visit the vertices in the following order: a, b, c, e, d, f, g. That is, we add the edges in the following order: (a,b), (a,c), (c,e), (a,d), (d,f), (f,g) This results in the same MST as above.

5 Midterm Prep

Good luck next Tuesday! There is a review session 7:00-9:00 tonight in Hearst Annex A1. Slides will be posted on Piazza afterwards. If there is extra time at the end of discussion, you could ask the TA to review certain topics for the midterm.