CS170–Spring 2015 — Solutions to Homework 5

Kevin Chau, SID 23816929, `kevinchau321@berkeley.edu`

Collaborators: None

1. Negative Edges

(a) Dijkstra's Algorithm **will** succeed in finding shortest paths in a graph where the only negative edges are leaving the starting node and there are no negative cycles either. One way to see this is that we could make all the negative edges positive by uniformly shifting all edges out of s up by $\|min(edge)\|$, and then we'd be left to do dijkstra's algorithm on a positive graph. In general, this shift doesn't work for any graph with negative edges; you may make multiple over-estimations of the shortest path in one path with multiple negative edges that have been shifted compared to a path with only one negative edge shifted (and therefore overestimated). However, since the graph we are considering only has negative edges out of the start, every shortest path is overestimated by the exact same amount. Another way to see this is if we add another node before s called s'. Let the distance from s to s' be equal to $\|min(edge)\|$ (remember, the minimum is the most negative edge). Notice that running Dijkstra's algorithm from s' is nearly the same as running the algorithm from s. The exception here is that to get from the starting node s' to all nodes v for the edges $(s, v) \in E$, the distance is $\|min(edge)\| + l(s, v)$. Because of the choice of the length from s to s', all distances from s' to the nodes v are greater than or equal to zero. Hence what we have done by adding s' is exactly just shifting all negative edges until they are non-negative (by the same amount), so Dijkstra's will work. A more formal proof would go as follows. Suppose we know the shortest distances of a subset of nodes $S \in V$, and we know (u,v) is the edge going out of S to the closest node v. Then the shortest path to v is just the path to u plus edge(u,v). This is still true even if there are negative edges out of the start node s. Suppose for the sake of contradiction that dist(u) + l(u,v) is not the shortest path to v. Then there is another path from s to v through an edge (w,z) where w is in S and z is not. However, the distance from s to z must be greater than the distance from s to v, since v was estimated to be the closest to all nodes in S. Thus by contradiction Dijkstra's algorithm will work on this class of negative graphs.

(b)
   (a) Main Idea: A graph G = (V,E) has no negative edges except for the edges leaving a particular vertex v, which is not the starting vertex s. Given the node t, this algorithm finds the shortest path from s to t.

   (b) Pseudocode: Run Dijkstra's algorithm on G starting from s (i.e. compute dijkstra(G,s) as per the textbook). This will give you the correct distance from s to v, so we can say length(s,v) = dist(v), where the dist() array is the same one updated in Dijkstra's algorithm. Then we can reset all the distances to zero with dist(u) = 0 for all $u \in E$. Next we can run one more pass of Dijkstra's, this time starting at v (i.e. run dijkstra(G,v)). This will return the correct minimum distance from v to t, so we can set length(v,t) = dist(t). Then the minimum distance from s to t is just length(s,v) + length(v,t) and the minimum path is the path found from s to v in the first Dijkstra pass concatenated with the path from v to t found in the second pass. However, we also need to consider the case where the shortest path from s to t does not go through v. So we will need to make one last Dijkstra pass, this time on the graph G' which has all the negative edges removed. If after this dist(t) is less than length(s,v)+length(v,t) as computed before, then this last run of dijkstra's algorithm returns the correct minimum path. Else, our first two passes suffice.

(c) Proof of correctness: We know that the first pass will be successful in finding a minimum path from s to v since there are no negative edges going into v (so we would encounter only positive edges if we were to terminate the algorithm after relaxing v). We also know that the second pass of Dijkstra's will find the right path from v to t since we have proved that in detail in part (a). Thus the shortest path from s to t will either be the path found in the third pass of the graph without any negative edges, or it is exactly the sum of shortest paths from s to v and then from v to t.

(d) Running Time: $O((\|V\| + \|E\|) \log \|V\|)$

(e) Justification: When we reset the value of the distances, this takes an extra O(V) time added to Dijkstra's algorithm, which just gets absorbed by a constant. Furthermore, in total we have 3 passes of Dijkstra's algorithm, which only adds a multiplicative constant that we can drop, so the algorithm runs in the same asymptotic time as Dijkstra's

2. Roads and Planes

1. Main Idea: A graph has cities V, undirected roads R and directed planes P. Roads are strictly positive while planes may be negative. If you take a plane, there is no combination of roads and planes to take you back to where you started; in other words, there are no cycles. We are given a starting city s and an end point t.

2. Pseudocode: Treat the undirected roads as two directed edges between cities. Use a DFS based strongly connected components to find the cycles of cities that are internally connected by roads but seperated from other clusters of cities by one way planes. One we have found the strongly connected components, treat these entire components as nodes (in other word, construct a graph with only plane edges, no road edges). We can linearize this meta-graph DAG starting from city s with a DFS-based topological sorting algorithm that orders the city clusters by decreasing post number. Using this linearization of the plane DAG, we can rank each city cluster in increasing order starting from the cluster that contains s, (s is 1, next city cluster has vertices of rank 2, next city cluster in linearization is 3, etc.). Now we run Dijkstra's algorithm, with the exception that our priority queue sorts by city cluster ranking of each vertex. When multiple vertices share the same ranking, we prioritize by minimum estimated distance as in the original Dijkstra's algorithm. After Dijkstra's has run, return the dist(t) and the pointers of prev nodes from t to find the path.

3. Proof of Correctness: First we note that because taking a plane means you cannot return to a cluster of connected cities and roads, the graph of clusters and plane edges (excluding roads) would be a DAG. So as long as we process the clusters of roads/ cities closer in ranking to that of s before we take directed edges away from said clusters, we are effectively finding the best way to traverse each cluster and the best way to connect these clusters. Another way to see this is that we are essentially doing the same algorithm in problem (1). By dealing with the roads and cities of rank 1 before those of higher rank, we can think of our city clusters as single nodes being connected by possibly negative edges. As in (1) the more formal proof is similar to the proof of Dijkstra's algorithm. Suppose we know the shortest path in some subset of the vertices. Then if we are exploring a new node v outside of this set we know that the ranking and distances we consider in our priority queue ensure that the edge we are exploring is the shortest distance to v. We can suppose for the sake of contradiction that the new edge we are exploring isn't the closest way to v, but then we will find that the ranking we have established means that all other paths must be longer.

4. Running Time: $O((E+V)+(E+V)\log V)$

5. Justification: This algorithm uses 2 DFS based algorithms. The first DFS is to find strongly connected components, and runs in linear time in the size of the number of roads and planes. The second DFS is a topological sort linear in the number of city clusters and planes. Of course, there could be as many as E planes, and as many as V city clusters (that is there are no clusters), so the linearization is O(E+V) as well. Dijkstra's algorithm has been modified for this algorithm but we haven't added any additional operations, so that part takes the same running time. In total, we add the DFS times to the Dijkstra's algorithm time to get $O((E+V)+(E+V)\log V)$.

3.

(a)
(a) Main Idea: n currencies $c_1, ..., c_n$. Given a set of exchange rates $r_{i,j}$, starting currency c_s and end currency c_t, this algorithm finds the sequences of exchanges that maximizes the total worth of the currency in units of c_t.

(b) Pseudocode: Construct a graph consisting of the nodes $c_1, ..., c_n$ with edges $e_{i,j} = (c_i, c_j)$ having weights equal to $-\log r_{i,j}$. Use the Bellman-Ford algorithm on node c_s to find the shortest path between c_s and c_t. Return the sequences of edges by following the prev points from c_t.

(c) Proof of Correctness: In order to get the most money in our sequences of exchanges, we must maximize the product of these exchange rates, which can be represented as $\prod r_{i,j}$. Maximizing this product is equivalent to maximizing the logarithm of the product, which is useful because it lets us split up multiplication of the exchange rates into additions of their logarithms. Since we are now adding log of exchange rates, adding up paths to measure distances between nodes makes sense rather than simply adding the exchange rates. Thus we can make a simple modification to solve this maximum path problem: simply negate the edge weights and find the shortest edge. Of course, since we are dealing with negative edges, we must use the Bellman Ford algorithm, since Dijkstra's is not suited for negative edges. Thus by using the Bellman Ford algorithm to find the minimum cost path of negative log exchange rates, we are equivalently finding the maximum product and therefore finding the best total exchange from s to t.

(d) Running Time: $O(n^3)$

(e) Justification: It takes O(n^2) to build the graph and take logarithms and negations, assuming we are using an adjacency matrix or similar representation. The Bellman Ford algorithm makes $\|V\|$ updates to $\|E\|$ edges, for a running time of $O(\|V\|\|E\|)$. For the graph of the currencies constructed, there are $\|V\| = n$ nodes and $\|E\| = n^2$ edges since there are exchanges between all currencies. Therefore the Bellman Ford algorithm takes $O(n^3)$ time on our graph. Adding n^2 and n^3 just gives us $O(n^3)$

(b)
(a) Main Idea: Detects when a cycle of exchanges results in greater than 1 total exchange rate.

(b) Pseudocode: Run the Bellman Ford algorithm on the graph constructed in part (a). Then do one more iteration cycle of updates to all the graph edges. If any distance from s to any other currency has been updated, then we have detected the anomly and return true. Otherwise, if all distances have stayed the same, there is no anomly and return false.

(c) Proof of Correctness: If after one more iteration of updates after the Bellman-Ford algorithm has ran any of the distances have changed, we know we have detected a negative cycle. A negative cycle means the $\sum -\log r_{i,j} < 0$, which is exactly equivalent to the product of exchange rates being greater than 1. Thus this algorithm correctly detects the required condition for an anomly.

(d) Running Time: $O(n^3)$

(e) Justification: This runs in exactly the time of the Bellman Ford algorithm. As in part (a), the analysis of number of edges is n^2 and the number of vertices is n

4. Maximum Spanning Tree

1. Main Idea: Given a graph G = (V,E) with edge weights w_e, find the maximum spanning tree of the graph.

2. Pseudocode: A simple modification of Kruskal's algorithm where the edges are processed in decreasing order from largest to smallest: Alternatively, one can multiply all the weights of

Algorithm 1 Maximum Spanning Tree

Require: A connected undirected graph G = (V,E) with edge weights w_e
Ensure: A maximum spanning tree defined by the edges X

```
procedure MaxST(G,w)
    for all $u \in V$ do
        makeset(u)

    X = {}
    Sort the edges $E$ from largest to smallest weight
    for all edges $(u, v) \in E$, in decreasing order of weight do
        if f thenind(u) $\neq$ find(v)
            add edge (u,v) to X
            union (u,v)
```

the graph by -1 and run Kruskal's algorithm on the inverted weights. In this inverted list, the largest edge will correspond to the most negative value, which is processed first by Kruskal's algorithm.

3. Proof of Correctness: This algorithm is correct for the exact same reason that Kruskal's algorithm is correct. First we know that this algorithm returns a spanning tree since it adds back all the edges except those that create a cycle. Since the graph has no cycles, yet has all other edges, it is guaranteed to be spanning. In other words, we know removing an edge in a cycle leaves the graph connected. We have removed all such cyclic edges in this algorithm, and the graph is still connected. By induction we know the tree is a maximum tree. When we add the first edge, we know that this is the largest edge and must be part of the spanning tree. Assume we are at some point in the algorithm where the edges added back so far are part of the maximum spanning tree. Then if we add one more edge to this (the next largest, non-cycle creating edge), we still have part of the spanning tree created. Further more, this is part of the maximum spanning tree since if there were another edge after this one that we could have added to build this spanning tree it would be smaller in length anyways. Thus the algorithm is correct.

4. Running Time: $O(\|E\| \log \|V\|)$ or $O(\|E\| \log(\|E\|))$

5. Justification: This algorithm has the same complexity as Kruskal's. The only difference is what order we sort and process the edges. If using a union-find disjoint sets data structure, we can achieve $\|E\| \log \|V\|$ for sorting the edges (with an algorithm like merge sort) and $\|E\| \log \|V\|$ union-find operations, for a total time of $O(\|E\| \log \|V\|)$. Also note that $\log E \leq 2 \log V$ since $E \leq V^2$, so the second time bound is equivalent.

5. Spanning Trees and Shortest Paths

Suppose you are given a weighted graph G = (V,E) where all edge weights are positive and distinct.

1. Prove that G has a unique minimum spanning tree. Proof by Contradiction:
   Suppose G has two minimum spanning trees T and T' that are distinct trees but both have the same minimum spanning value. Since they are distinct trees, there must be an edge e that is the smallest edge present in only one tree but not the other. Without loss of generality, suppose e is in T but not in T' and is the smallest such edge. Since T' is a minimum spanning tree, we know adding e from T will produce a cylce in T'. Furthermore, there is a edge e' contained in this cycle which is heavier than e. We know e' is heavier than e since all edges lighter than e were in both T and T' and did not cause a cycle. Thus we can make T' have a smaller total weight than T by removing e' after we added e. Thus T' is the new minimum spanning tree rather than T, which contradicts the assumption that T and T' were both minimum spanning trees.

2. Suppose you are also given a distinguished vertex s. Is it possible for a tree of shortest paths from s and a minimum spanning tree in G to not share any edges?
   It is **not** possible for a tree of shortest paths from s (found with Dijksta's) to not share any edges with the minimum spanning tree. That is, the shortest path tree from s and the minimum spanning tree must share at least 1 edge. To see this, consider the shortest edge from s to any adjacent node, e=(s,s'). It is safe to say that we know such shortest edge exists, since all edges in the entire graph are unique and positive, so s has a unique and positive shortest edge leaving it. Thus (s,s') is exactly the shortest path between s and s', and thus would also be in the shortest path tree from s. If there were other paths from s to s', they would have to be longer, since all other edges from s are longer than (s,s'). We also know that the minimum spanning tree is unique from Kruskal's algorithm or Prim's algorithm. Since there isn't a shorter way than (s,s') to span s with s', (s,s') must be in the minimum spanning tree as well. Also note that if we did run Prim's algorithm starting on s, (s,s') would be the first edge added in the unique minimum spanning tree formed. Thus we have proved that the minimum spanning tree and the shortest path tree from s share at least one edge.

6. Another MST Algorithm

(a) Proof by contradiction: Suppose we are looking at no particular cycle in G with a heaviest edge e in the cycle. For the sake of contradiction, also suppose that e is in the minimum spanning tree of the graph. Then we can remove e from the MST and replace it with any other edge e' in the cycle and still be left with a spanning tree since both edges were in the same cycle. Since e' is smaller than e by the fact that e is the heaviest, this will be a new minimum spanning tree with less total weight than the tree with e. This contradicts our original assumption that e is in the minimum spanning tree. Thus if we pick the heaviest edge in any cycle we know it will not be in the minimum spanning tree. This argument holds for when the edge lengths are unique. If they are not unique, then the heaviest edge in a cycle may not be a unique heaviest edge, i.e. there may be other edges in the cycle of the same length. Then the MST is not unique either for this graph, and thus if we pick any one of the (same weighted) heaviest edges, there is an MST that does not contain it. Another way to say this is that we can remove the edge we picked from the MST and replace it with another heaviest edge equal in weight and in the same cycle to form a different MST.

(b) Proof of Correctness: First of all, it is clear that this algorithm returns a spanning tree (not necessarily the minimum one). The algorithm looks through all edges e $\in E$ of the graph and if e is in a cycle it removes it from the graph. By removing all the edges that create a cycle, the tree spans the vertices without any extraneous edges. Now we must prove that the tree is a minimum cost tree. After sorting all the edges in decreasing order of weight, the first edge e in a cycle found clearly must be removed to form the minimum spanning tree. In other words, since e is the first edge found in a cycle, it must be the heaviest edge in that cycle (it may be not unique). In part (a) we have proved that this heaviest edge is not present in one of the MST's of the graph. If we remove it, we know we can do so safely and still work towards a spanning tree that is an MST. So we can keep removing heaviest edges from cycles when we find them while still maintaining the possibility of ending up with an MST. After we have removed all edges found in a cycles, we can't remove any more edges from the graph without disconnecting our spanning tree. So if the tree we are left with must be a spanning tree, and it must be minimal from the fact that we removed edges without destroying the possibility that the tree is an MST.

(c) Running Time: The sorting takes $\|E\| \log \|E\|$ time for all the edges, which we only do once. To find whether an edge is in an cycle, we just use a DFS traversal for cycle detection (start the DFS on one of the nodes of the edge, check for no backedges to that node). We know DFS runs in $O(\|E\| + \|V\|)$, and we have to do this $\|E\|$ times since we are looping through all the edges. Thus it takes $O(\|E\|(\|E\| + \|V\|))$. $\|E\| \leq \|V\|^2$ so we can replace $\|V\|$ with $\|E\|^{1/2}$ and get $O(\|E\|(\|E\| + \|E\|^{1/2})) = O(\|E\|^2 + \|E\|^{3/2})$ or just $O(\|E\|^2)$. So the running time is just the addition of the time for sorting and for looping, $O(\|E\|^2 + \|E\| \log \|E\|)$