Parth Ray

Lab 5-1

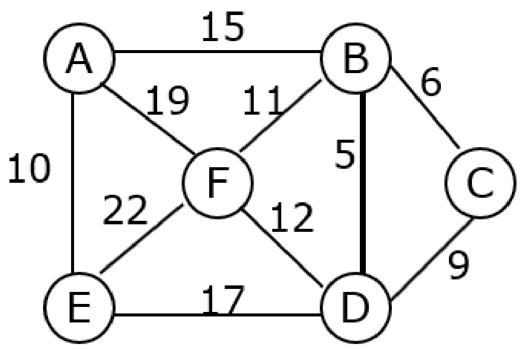
1. Consider an undirected graph g = (V, E) with non-negative edge weights $w_e \geq 0$. Suppose you have computed a minimum spanning tree for G, call it T_{mst} . Suppose each edge of the graph is now increased by 1, the new weights are now $w_e' = w_e + 1$. Does the minimum spanning tree always contain the same edges?
If yes, justify (not a formal proof) your answer. If no, give an example where the edges in the MST change.

    Yes, because if you increase the weights of all the edges, the cut property still holds, and the same minimum edges will be chosen. You will still pick the next lowest edge since all edges were increased.

2. Does Prim's algorithm always work correctly on a connected graph with negative edge weights?
If yes, justify (not a formal proof) your answer. If no, give an example where the edges in the MST are different from those found by Prim's Algorithm.

    Yes, since Prim's takes the lowest weighted edge that does not form a cycle in the tree. The negative edge weights also follow this algorithm since we just want the smallest edge that does not form a cycle with the current MST.

3. Trace Prim;s algorithm as it constructs the Minimum Spanning Tree for the following graph.



Each line of the following show the state of a Priority Queue implemented as an unsorted list where the first entry is the other vertex on the shortest edge that connects to the existing tree. The first column contains the vertices that are already in the tree constructed by Prim

MST	A	B	C	D	E	F
INIT	(–, 0)	(–, ∞)	(–, ∞)	(–, ∞)	(–,∞)	(–,∞)
A	(–, 0)	(A, 15)	(–, ∞)	(–, ∞)	(A, 10)	(A, 19)
E	(DONE)	(A, 15)	(–, ∞)	(E, 17)	(–, 0)	(A, 19)
B	(DONE)	(–, 0)	(B, 6)	(B, 5)	(DONE)	(B, 11)
D	(DONE)	(DONE)	(B, 6)	(–, 0)	(DONE)	(B, 11)
C	(DONE)	(DONE)	(–, 0)	(DONE)	(DONE)	(B, 11)
F	(DONE)	(DONE)	(DONE)	(DONE)	(DONE)	(–, 0)

4. Determine the computational complexity of Prim's Algorithm where the Priority Queue is implemented as an unorder list of vertices. Justify your answer.

   The complexity would be $O(|E| * |V|)$ since you have to go through the list of vertices each iteration and update the priority by checking each edge.

5. Suppose we are given a minimum spanning tree problem on a graph G with all edge costs positive and distinct. Let T be the MST for this instance. Now replace each edge cost by its square that is c_e is replaced by is c_e^2. This creates a new instance of the MST problem. Must T still be a minimum spanning tree for this problem. Justify your answer.

   Yes, T is still the MST for this problem. Since you are squaring all of the edge weights, when you compare the edges to find which edge is the lowest and doesn't form a cycle with the existing MST in Prim's algorithm, the lowest edge does not change. Even though the total weight of the MST is increased, you are still using the same edges from T because the closest edge did not change.

6. Suppose that we represent the graph G- (V,E) as an adjacency matrix, Find a simple implementation of Prim's algorithm that runs in $O(|V|^2)$.

   Start with a vertex v in tree T. Then go through the adj matrix of all the vertices in T and get the minimum edge of a vertex, w, that is not already in T (takes $O(|V|)$ time). Add vertex, w to the spanning tree of T. Repeat until all vertices added to T (repeat |V| times). Total complexity is $O(|V|^2)$.

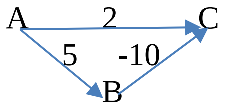
Lab 5-2

1. Consider an undirected graph g = (V, E) with non-negative edge weights $w_e \geq 0$. Suppose you have computed shortest paths to all the nodes from a particular node $s \in V$. Suppose each edge is now increased by 1, the new weights are now $w_e' = w_e + 1$.
Do the shortest paths always stay the same? If no, give an example where they change.
If yes, justify why they do not change.

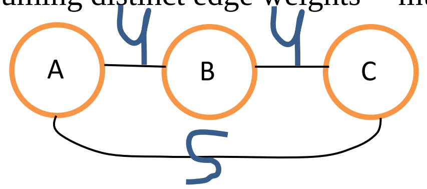
   No, the shortest paths do not always stay the same since we are adding up the edge weights in Dijkstra's algorithm. For example, say we have a graph that has two paths to vertex K, from vertex A. Let the first path be A, (2, B), (3, K) and the second path be A, (1, C), (1, D), (1, E), (1, K). Then, out shortest path would be the second path as 4 < 5. Then if we were to increase the weights of all the edges by 1, the first path would become the shortest path as 8 > 7.

2. Does Dijkstra's algorithm always work correctly on a connected graph that has some negative edge weights? yes/no?
   a. If not give a counterexample.
   b. If it does always work correctly, justify (not a formal proof) your answer.

   No, it does not work on graphs with some negative edges. Consider the graph with three vertices and the directed edges of (A, C, 2), (A, B, 5), (B, C, -10). Dijkstra's algorithm would result in the shortest path for C being the path from A to C which of length 2, but the actual shortest path would be from A to B to C which has length of -5. This is because once we remove C from the pq and put it into the spanning tree, we have no way of updating the distance of C due to the edge from B to C. This is because Dijkstra relies on the fact that if all weights are non-negative, adding an edge can never make a path shorter.



3. a. Give as simple as possible example where Prim and Dijkstra give different spanning trees.
   b. Assuming distinct edge weights -- must the smallest edge be in the SPT?

   a.


   Here the Prim MST would be [A, B, C] with a total edge weight of 4 and the Dijkstra SPT would be [A, B [A], C [A]] to get the shortest path from A for each vertex. Therefore, Prim and Dijkstra give different spanning trees.

   b. No, you can have a shortest path without the smallest edge. For instance, if you have a graph with three vertices and edges (A, B, 2), (A, C, 3), (B, C, 4), the shortest path would be A to C with a path length of 3. As you can see, the shortest edge of (A, B, 2) is not in the shortest path tree.

4. Consider the following proposal for how to find the shortest cycle in an undirected graph with unit edges.

   When a back edge, say (v,w), is encountered in a depth first search, the path from the w to v in the dfs tree along with the edge (v,w) forms a cycle. The length of the cycle is: (***level of w) – (level of v) + 1*** where the level of a vertex is its distance (#of edges) from the root. This suggest the following algorithm:

   - Do a depth first search , keeping track of the level of each vertex
   - Each time a back edge is encountered, compute the corresponding cycle length and save it if it is smaller than the previous smallest cycle seen.

   Show that this strategy does not work by providing a counterexample and a brief (1-2 sentence) explanation.

   Consider a graph with 5 vertices, A – E, with 6 edges ([A, B], [A, E], [B, C], [B, E], [C, D], [D, E]). Using the algorithm above, we would get a cycle of length 4 from B to C to D to E to B. The actual shortest cycle is of length 3 from A to B to E to A. This strategy does not consider the cycles formed by multiple back edges.

5. Consider a weighted graph where the only negative edges are those that leave s, the starting vertex. Will Dijkstra's algorithm always work on such graphs. If not, give a counterexample.
   If yes, give an explanation why it works.

   Yes, it will work on these types of graphs. Since the only negative edges in the graph are those that leave the starting vertex, Dijkstra's algorithm will still work. Well if the negative edges are all coming out of the starting vertex, we are guaranteed to traverse through those edges by the definition of the algorithm. Since we are guaranteed to traverse these negative edges, we can accurately calculate the shortest path as long no other edges are negative. The problem with negative edges in other graphs was that we would finish the path to a vertex without looking at a negative edge since this edge would be later in the algorithm. However, since we are guaranteed to traverse through the negative edges, the algorithm is correct.

6. Consider that breadth first search finds the shortest paths in a graph when all the edge weights are one. In other words, the path length is measured as the number of edges in the path. (This is sometimes called the "edge length".) In this situation, give a linear time O(|E|) algorithm that computes the <u>**number of distinct shortest paths**</u> from a start vertex to each of the other vertices.

   Initialize the distance of the source to be 0 and the rest of the vertices' distances to be a large number. Initialize all the vertices' number of distinct shortest paths to be 1. Using breadth first search, for each vertices' adj vertices, update the distances for the adj vertices. If the distance of the current vertex + 1 is equal to the distance of the adj vertex, then increment the number of distinct shortest paths for the adj vertex by one. If the distance of the current vertex + 1 is less than the distance of the adj vertex, then update the distance of the adj vertex to distance of the current vertex + 1. Then, all the vertices will have several distinct shortest paths assigned to it. Since we check every edge once, our complexity is O(|E|).