## Lab week 5-2: Dijkstra and Prim

1. Consider an undirected graph g = (V, E ) with non-negative edge weights we ≥ 0. Suppose you have computed shortest paths to all the nodes from a particular node s ∈ V . Suppose each edge is now increased by 1, the new weights are now we‘ = we + 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.

1. Does Dijkstra’s algorithm always work correctly on a connected graph that has some negative edge weights? yes/no?
   1. If not give a counterexample.
   2. 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.

A 2 C

5 -10

B

1. 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?

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.

1. 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.

1. 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.

1. 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 **number of distinct shortest paths** from a start vertex to each of the other vertices.

## Low level Pseudo code:

**Algorithm**: Prim-MST (G)

**Input**: Graph G=(V,E) with edge-weights.

// Initialize priorities and place in priority queue.

1. priority[i] = infinity for each vertex i

2. Insert vertices with priorities into priorityQueue;

// Set the priority of vertex 0 to 0.

3. priorityQueue.decreaseKey (0, 0) //

// Process vertices one by one in order of priority

4. **while** priorityQueue.notEmpty()

// Get "best" vertex out of queue.

5. v = priorityQueue.extractMin()

6. Add v to MST;

// Explore edges from v.

7. **for each** edge e=(v, u) in adjList[v]

8. w = weight of edge e=(v, u);

// If it’s shorter to get to MST via v, then update.

9. **if** priority[u] > w

10. priorityQueue.decreaseKey (u, w)

11. predecessor[u] = v

12. **endif**

13. **endfor**

14. **endwhile**

15. Build MST;

16. **return** MST

**Output**: A minimum spanning tree of the graph G.

Source: pseudo code modified by tjk <http://www.seas.gwu.edu/~simhaweb/alg/lectures/module8/module8.html>

## Low level Pseudo code:

**Algorithm**: Dijkstra-SPT (G, s)

**Input**: Graph G=(V,E) with non-negative edge weights and designated source vertex s.

// Initialize priorities and place in priority queue.

1. priority[i] = infinity for each vertex i;

2. Insert vertices with priorities into priorityQueue;

// Source s has priority 0

3. priorityQueue.decreaseKey (s, 0)

// Process vertices one by one in order of priority

4. **while** priorityQueue.notEmpty()

// Get "best" vertex out of queue.

5. v = priorityQueue.extractMin()

6. Add v to SPT;

// Explore edges from v.

7. **for each** edge e=(v, u) in adjList[v]

8. w = weight of edge e=(v, u);

// If it’s shorter to get to u from s via v, update.

9. **if** priority[u] > priority[v] + w

10. priorityQueue.decreaseKey (u, priority[v]+w)

11. predecessor[u] = v

12. **endif**

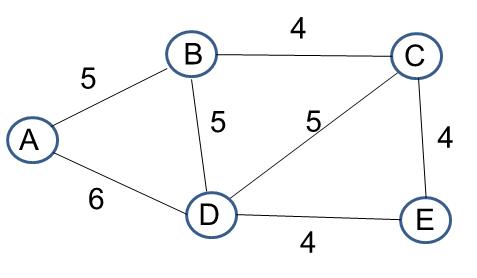
13. **endfor**

14. **endwhile**

15. Build SPT;

16. **return** SPT

**Output**: Shortest Path Tree (SPT) rooted at s.



PRIM’S ALGORITHM (prev, dist)

MST A B C D E

INIT ( , ) ( , ) ( , ) ( , ) ( , )

A (0, -) (5, A) (-, -) (6, A) (-, -)

B (Done) (0, -) (4, B) (5, B) (-, -)

C (Done) (Done) (0, -) (5, B) (4, C)

E (Done) (Done) (Done) (4, E) (0, -)

D (Done) (Done) (Done) (0, -) (Done)

DIJKSTRA’S ALGORITHM (prev, dist)

MST A B C D E

INIT ( , ) ( , ) ( , ) ( , ) ( , )

A ( , ) ( , ) ( , ) ( , ) ( , )

( , ) ( , ) ( , ) ( , ) ( , )

( , ) ( , ) ( , ) ( , ) ( , )

( , ) ( , ) ( , ) ( , ) ( , )

( , ) ( , ) ( , ) ( , ) ( , )