Walker Hertel
Raymond Schooley
David Dean

# Project 3 Write-Up

**I would just like to say that we had to include  a separate Main class with our code that is needed for the extra-credit option.

## How we test our code

The general strategy was to read the assignment description over and over to make sure that our code was conforming to expectations in every way we could imagine.  Did we go overboard?  Hard to say, all I know is that our test class ended up being close to the same size as the MyGraph class.  The first thing we did with the test class was create a setup method that runs before every other method to ensure uniform values for graph.  It seems helpful to split the test discussion into two categories the valid and the invalid input.

## Valid Input:

We tested the constructor by simply making sure that our graph is not a null pointer and that it is an instance of a MyGraph.  Then we check for how the graph handles redundant information.  When you try to feed in duplicate vertexes and edges it should not save the redundant stuff and it appears to be behaving correctly.

We tested shortestPath six ways from Sunday.  First we tested it with the most simple two vertex path, then with more.  The most interesting tests where the ones in which the graph had to choose to traverse more vertices to find a path with less weight.  We tested that scenario with a three vertex run and then again with significantly more vertices. Finally we tested for theshortest path when the source and destination vertex are one and the same.

## Invalid Input:

To begin the test of invalid input we check to make sure there are no edges that refer to vertices that are not in the graph.  Since our group decided to implement all of the error handling having to do with edges in a single method, it seemed like a good idea to create some custom exceptions.  This way in the test class we can make sure that things are failing correctly and not returning InvalidArgmentException for one of 3 reasons.  So we tested for edges with weights less than 1, self-looping edges, and edges with already saved vertices but a different weight.  We also checked and threw a NullPointerExcection if a null pointer somehow got saved in the list of edges.

The last thing we check for is an unreachable nodes or the “null path”.  We do a simple test with 2 vertices that are connected but we ask it to go the wrong direction.  Then we use a vertex that we created specifically for this test that is out in the nosebleed section of the graph.  We ask the graph to traverse from on side to another right up to the destination vertex, only to find that it is of in-degree 0.  In our very last test we ask the graph to find a shortest path with when the source vertex is not in the graph and our graph doesn’t seem to get tripped up.

# Asymptotic Complexity

adjacentVertices:
For adjacentVertices you make sure that you have the vertex in your graph, and then looking at all the vertices you can reach from the source vertex. We used a hashMap to map each vertex in our graph to an ArrayList of edges that all have that vertex as their source. It's constant time to check if the vertex in in the hashMap, and the worst-case running time is O(|V-1|) if somehow that vertex was connected to every other one in the graph.

edgeCost:
Similar logic as adjacentVertices hold for edgeCost. Assuming that the source vertex is in your graph, it can potentially be connected every other vertex, which gives it |v-1| edges. It is possible that the destination vertex you are seeking is the last one in the source vertex's adjacency list and so you would have to iterate through all of them making this operation O(|V-1|).

shortestPath:
The shortest path is tricky. Since Dijsktra's algorithm is a greedy algorithm it guarantees that your only need to "touch" each edge once. This is supported by our teams choice of using the priority queue. Each time we get a vertex from the priority queue, we know that it is the lowest cost and ready to become part of the set of vertices that are "known". It takes O(|E|log(|V|) to visit each edge and O(|V|log(|V|) to visit each vertex. The worst-case complexity of running Dijkstra's with the use of a priority queue should be O(|V|log(|V|) + |E|log(|E|)).