Results

Introduction:

For our CS 225 Final Project, our group decided to implement the BFS traversal, Dijkstra's Shortest Path Algorithm, and the Page Rank Algorithm using data sets from Stanford SNAP (http://snap.stanford.edu/data/web-Google.html) and the University of Utah (https://www.cs.utah.edu/~lifeifei/SpatialDataset.htm). Apart from writing the code for all our algorithms, our group also focused on creating comprehensive test cases. Because we encountered a relatively large Google Web dataset, we developed a function which would randomly load a smaller portion of the data with a seed. The corresponding test case consisted of randomly selecting two "chunks" of 500 data points each and making sure that they were truly random and not duplicates.

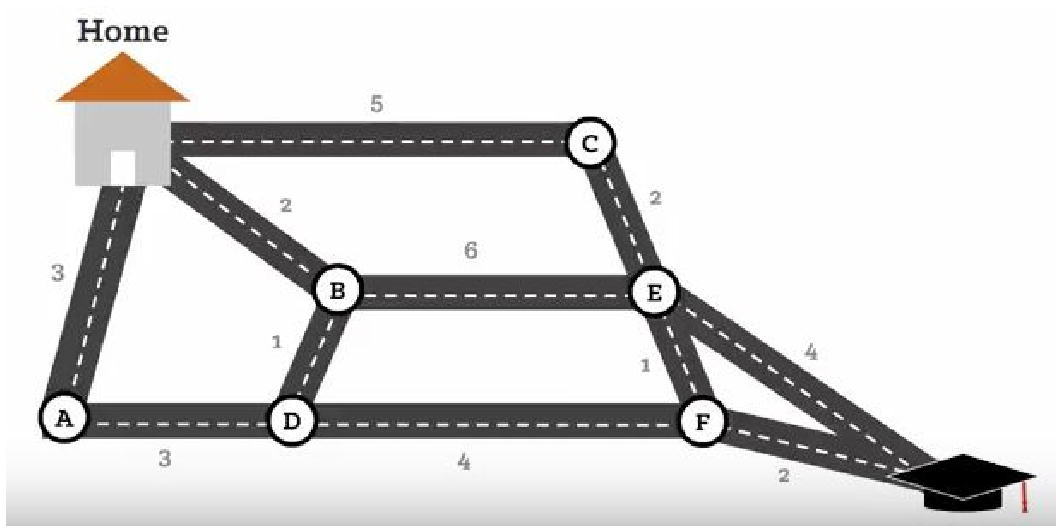
BFS Tests/Results:

For the BFS implementation, our group chose to write a constructor which would take in a graph and mark every vertex as "false" indicating that it was unexplored, and set each edge to "UNEXPLORED" as well. Our code then made use of a helper traversal function which would update edges as "CROSS" or "DISCOVERY", and would update vertices' visited property while pushing the visited vertices into a queue called bfsPath. In order to test our BFS traversals, our group accounted for the following layouts: a simple cycle graph, a complex graph (seen in the lecture notes by Professor Evans), and a disconnected undirected graph. Because the BFS traversal started at a random vertex, our group checked the number of "CROSS" and "DISCOVERY" edges for each layout. From our results we were able to determine that the BFS traversal was visiting and updating edges and vertices correctly. Results returned are a list of edges and their labels, either "CROSS" or "DISCOVERY".

Dijkstra's Shortest Path Tests/Results:

For the Dijkstra's Shortest Path Algorithm, our group utilized an unordered map of distances and a separate unordered map of visited vertices. By using a priority queue of 'nodeDistance' and a custom comparator, we are able to find the smallest distance of an unvisited vertex, and then then check the adjacent vertices to update distances. By repeating this until all vertices have been visited.. For the edge case in which the map is disconnected, and the path is nonexistent, the shortest distance would not change while iterating through the vertices. As a result, we can simply rephrase any remaining vertices that have infinity after iterating to print out "path is not connected" instead.

For testing, we made two test cases. In the simple test case, we made a graph starting from point A, going in alphabetical order to point E, and required the distance to increase by one each time. For the complex case, we used a complex graph from Brilliant.org to test for accuracy (See below):

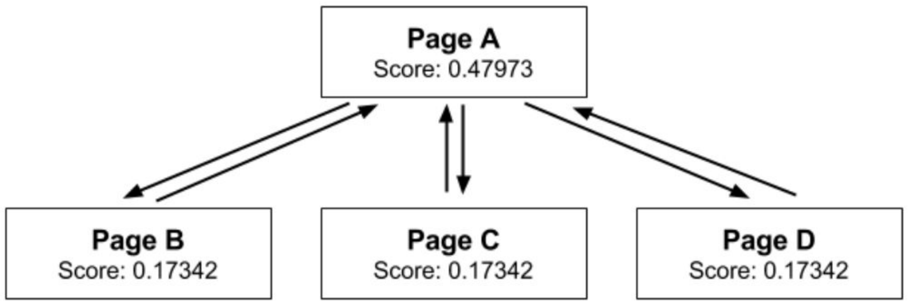


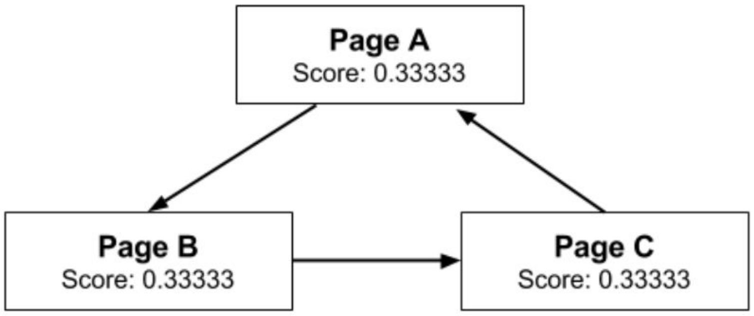
Source: **https://brilliant.org/wiki/dijkstras-short-path-finder/**

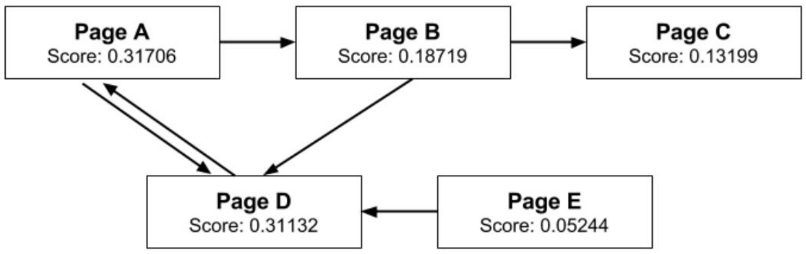
Results returned is a list of distances (the shortest possible) from the starting point to every corresponding vertex. For the above example, it would be 9 from Home to School.

PageRank Tests/Results:

After our group had implemented our Page Rank algorithm, we performed test cases involving a few different graph layouts. These layouts included the following: a simple graph (where every vertex had an outgoing edge), a simple cycle graph, and a complex graph (where some vertices may not have outgoing edges). In addition to accounting for different graph layouts, we also tested the accuracy of our algorithm by performing iterations and checking the range of updated rank values of the 2nd, and 100th iterations for our graphs. Our results indicated that all of our updated ranks were correct for each corresponding iteration because they fell in the designated optimal range provided by the PageRank formula. Below are the graphs that the test cases covered.







Results returned are all vertices with their corresponding PageRank and the top 3 ranked vertices.
Source of algorithm followed:
https://courses.cs.washington.edu/courses/cse373/17au/project3/project3-3.html

Conclusion/Summary:

All algorithms are implemented with data structures that result in optimal runtimes. To build a graph from data, we used the adjacency list. PageRank is implemented using the adjacency list as well. We originally attempted to implement an algorithm that uses an adjacency matrix for matrix multiplication, but discovered that that the runtime would be disastrous. Dijkstra's shortest path algorithm is implemented utilizing a priority queue and an unordered map of vertex to bool (whether the vertex has been visited).