Results.pdf:

- **Output and correctness of each algorithm:**
  - **BFS**:
  -

```
queue<int> q;    //create a queue for BFS, and mark curren
visited[s] = true;
q.push(s);

while(!q.empty()) {
    int v = q.front();
    if(v == d) {
        return printBFS(parent, v, d);
    }
    q.pop();

    for(size_t it = 0; it < adjMatrix[v].size(); it++) {
        if(!visited[it] && adjMatrix[v][it] != 0) {
            visited[it] = true;
            q.push(it);
            parent[it] = v;
        }
    }
}
```

    - **Target Efficiency**: O(V + E)
    - **Actual efficiency**: While the queue is not empty (having initially filled the queue with the starting location) , we search each vertex stored on the queue, and check the edges to that vertex. For every vertex we check, we mark that vertex as “visited”, and push that vertex onto the queue. Each time we check a new vertex, we pop off the previous, allowing us to keep the most efficient array of vertices to our destination on our queue. This efficiency is O(V^2), since we are checking each vertex (marking it as visited) in our adjacency matrix. This is significantly slower than we would have liked, but the adjacency matrix makes up for it in accessing elements of O(1). If we had used an adjacency matrix, it likely would have taken O(V^3) time, because we would not have been able to check if two vertices had an edge in O(1) time to mark them visited.
    - **Tests:** We ran tests on a small graph, and a larger graph of which we assigned edges to. We ran individual tests, as well as tests on the entire graph for BFS, and matched them up to our individual mapping using visuals.
      - Ex)

```
int src = 2, dest = 3;
cout << "Testing BFS 1: "<<endl;
cout<< "Should print: 3"<<endl;
g1.findShortestPathBFS(src, dest);
```

- **Dijkstra**:

```
for(int i = 0; i < V-1; i++) {
    int u = getMin(distance, visited);
    visited[u] = true;
    for(int v = 0; v < V; v++) {
        int currDistance = distance[u] + adjMatrix[u][v];
        if(!visited[v] && adjMatrix[u][v] != 0 && (distance[v] > currDistance)) {
            distance[v] = currDistance;
            parent[v] = u;
        }
    }
    if(u == dest) {break;}
}
```

-

```
int Graph::getMin(int distance[], bool visited[])
```

  - **Target Efficiency**: O((V + E) * log(V))
  - **Actual Efficiency**: Using a helper function getMin, which returns a minimum node given a array of visited vertices with an array of distances of the current traversal, we looped through the remaining V-1 vertices (after assigning the first node in path as the first vertex) to find the shortest path that the user could take given a start and a destination. The program breaks when you have reached the given destination. The worst case efficiency is O((V+E) * V), if two nodes that are closest are at the start and end of the adjacency matrix. On average though, we meet our goal which is O((V + E) * log(V)), since we will be hitting our destination with the shortest path before we search every single node on the outer loop.
  - **Tests**: We ran tests on a small graph, and a larger graph of which we assigned edges to. We ran individual tests, as well as tests on the entire graph for Dijkstra's, and matched them up to our individual mapping using visuals. Doing this with a simple graph and a complex graph allowed us to make sure the algorithm was working before we applied it to the full dataset.
    - Ex)

```
cout<<"The path with the shortest distance starting at " << src << " and ending at " << dest << " is: " << endl;
dijkstraVector = g1.dijkstra(src, dest);
```

- **Pagerank**:
  - **Function Output:** We have provided two PageRank functions for our implementation. One function takes no arguments while the other function takes the number of iterations PageRank should be computed, as well as a damping factor. Our PageRank algorithm outputs a vector of cities sorted in order from most popular to least popular, based on the number of roads and their lengths to that city. This vector can be used to find how

popular a given city is compared to the rest of the cities in the dataset. Although we initially wanted to output a ranking of the cities that can be reached from a given starting city, our graph representation of the dataset consists of one large connected component and so, every city can be reached from any given city in the dataset.

- **Target Efficiency:** O($V + E$), where V is the number of nodes and E is the number of edges.
- **Actual Efficiency:** O(nV^2), where V is the number of nodes and n is the number of iterations used in power iteration. Running PageRank on a dataset consisting of 10,000 cities takes 439 seconds or 7.32 minutes to compute. The large difference between the target efficiency and the actual efficiency of PageRank was mainly due to us being unsure of how to best implement the algorithm to fit our graph-based dataset. Since we decided to implement our graph using an adjacency matrix, we chose to implement PageRank using a power iteration method with a time complexity as described above.
- **Tests:** To test the main PageRank functions, we wrote multiple test cases for smaller subsets of our dataset whose results we could manually confirm. We also wrote test cases for all of the matrix operations used by PageRank to ensure that they also worked as intended.

- Answer to leading Question:

Our leading question was– How can we give people the shortest route to their destination, given their starting location?

Reflecting back on our leading question, we feel like we were able to accomplish this. As a matter of fact, we allow the user to have 3 ways to get a shortest route to their desired destination. Using BFS, we were able to allow the user to navigate a route that would avoid traffic, since BFS would pass through the fewest cities while taking a short path in comparison to its neighboring nodes. Using Dijkstra's, we were able to allow the user to navigate a route that was the shortest path (Distance) possible. This allows the user a chance to watch their gas economy while they are driving, while likely taking the least amount of time to get to their destination. Using Pagerank allows the user to determine how popular their city is compared to other cities within the network. Furthermore, PageRank can also be used to determine a list of the most popular cities, which can be used when it comes to planning road trips, vacations, and such.

We also discovered that there are many factors that go into the REAL world algorithms that Google or Apple use for their maps, since just using an algorithm as Dijkstra's does not take into account for traffic, or accidents, or construction (something that we know all too well of driving down I-57 to UIUC).

Overall, although there were some inefficiencies (especially with the way we stored and used the large dataset), we came to the conclusion that our program gives a diverse way for the

user to reach their destination. If we were to do it again, the first thing we would change would be using a priority queue for Dijkstra's. Because we use a minimum helper function, the worst case run time is O((V + E) * V), and the average is about O((V + E) * log(V)). If we were to use a priority queue or even a minimum heap, we would have our worst case run time as O((V + E) * log(V)) and an average run time of somewhere bound by that worst case and O(V+E). Because of this inefficiency with both Dijkstra and Pagerank, we were not able to use the full 50,000 line dataset. Generating, running and producing an output just takes too long. By that point you would be halfway to your destination. We also found a loophole in the BFS algorithm for weighted graphs, which is why we had to prompt the user that it will help with traffic, and not necessarily gas economy. BFS will give you the fewest nodes traversed to your destination, since BFS will not work correctly on a weighted graph.

In conclusion, from start to finish, this was a great project. It allowed us to explore different topics that we learned in class, and actually apply them, from scratch, to a program of our choice. We started out with Amazon Fine Food reviews (and quickly learned that the dataset was a bad choice as we tried to use a BTree with a graph), and eventually made our way to the Road Network dataset of North America. As a group, this could have been our undoing. But we made the transition seamlessly, and all did our part to put out the final product. Something we could have done better, is definitely meet more often at the very beginning of the project. The last couple weeks were perfect. We met multiple times a week, and that was great for development. But at the start, we definitely could have picked up on the errors in the Amazon dataset earlier if we had met more often. Having said this, we feel, as a group, that we worked extremely well together, and felt that the work distribution and communication was better than expected.