**Problem Set 3:**

Before getting into detail, I wanted to clarify that I had a different main file for each section of the problem set. All code is written in Python3.

**Represent Graph as an Adjacency List**

This part of the assignment involves simply reading in the vertex and edge data for the

graph and representing it as an adjacency list

When compiling, the only thing the user needs to do is hit run. It will first print out the information gathered from each txt file and will then print out an adjacency list.

The output for the adjacency list is listed below

Text

Description automatically generated

These are the classes that I used to implement the adjacency matrix. The class Node was able to define the name and ID number and then return that. The Graph class is able to store the adjacency list as a dictionary, assuming that the edge is bidirectional.

**Text

Description automatically generated**

**Breadth-First Search**

In this part of the assignment I was able to print out the shortest path from Arad to Sibiu, from Arad to Craiova, and finally from Arad to Bucharest utilizing the breadth-first search.

When compiling the code, all the user needs to do is hit run. When looking at the output, the shortest path from each location will be the last item on the list. For example, the shortest path from Arad to Craiva would be Arad to Sibiu to RimniCuVilcea. The code automatically presumes that the next destination after RimniCuVilcea would be Craiva.

Text

Description automatically generated

The code below is able to show my implementation of this type of search. I used only one definition titled “bfs\_shortest\_path”. This definition is able to find the shortest path between two cities. The explored array can keep track of all the nodes explored during the while loop. This loop continues to iterate until all possible paths have been found. The last path found will be the one returned as it is the best-case scenario.

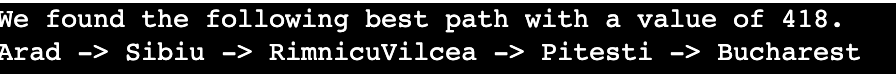
Text

Description automatically generated

**Dijkstra’s Algorithm**

During this section of the project, I was able to find the shortest path from one city to another utilizing there weights in Dijkstra’s Algorithm.

When running this code, all the user needs to do is hit run. It will then print out the shortest path from the starting to the ending location. When finding the best path from Arad to Bucharest, I was able to find the weight of the best path was 418 and the cities in which the path went through. This output can be found below.

****

This path was different from the breadth-first search implemented above. While this search relied only on the weights given by the weight on the Romania Vertices text file, the other algorithm relied only on the least number of vertices. In the breadth-first search, the fastest route found only has two stops (Sibiu and Fagaras). Without knowing the weights of routes, it was easily presumed that the least number of stops would be the fastest path. However, when using Dijkstra’s algorithm, we found that going through RimnicuVilcea and Pitesti would actally be faster than Fagarus because of the weight. This was completely expected to happen when examining the graph on the front of the Problem Set 3 instructions.

Below is the code utilized in this portion of the assignment. This code created a similar graph class which initialized variables and constructed the graph. It was also able to get the nodes and return the neighbors of those nodes and the value of an edge between two nodes (the weight). The definition of the class first used the max\_value to find what the “infinity” value of unvisited nodes where. It then initialized the starting node’s value with 0. During the “while unvisisted\_nodes” the algorithm is implemented until all nodes have been visited. The minimum node is found and stored during each iteration to find the shortest path between destinations.

Text

Description automatically generated