Bryan Guzman
ICSI 435-Homework 1
Student ID: 001265918

The file HW1_G1_G2 performs depth first search (dfs) and breadth first search (bfs). The file imports collections so that stack and collections can be used.

For depth first search using a stack, I created a function that takes 3 parameters: graph, start and goal. The function then initializes 3 lists, visited , states and stack. I then start up a while loop that ends when the stack is empty which happens when all nodes have been visited or the goal state has been reached. In the while loop, the stack pops the node into a variable called node and appends the node to the state to be kept for the state expansion. If the node is not in the visited list it is appended on. The node is then used to find all connected nodes using a for loop and appending any unvisited nodes to the visited list. The while loop ends once there are no more nodes or the goal node is reached and the expanded states and path are returned. For graph 1 and 2 the output is:

```
Stack DFS Vertex List (Graph 1)
State Expanded:
S,D,E,H,Q,P,P,R,F,G
Path Returned:
S-D-E-H-Q-P-R-F-G

Stack DFS Vertex List (Graph 2)
State Expanded:
S,D,E,H,P,Q,Q,R,F,G
Path Expanded:
S-D-E-H-P-Q-R-F-G
```

For dfs using recursion I used three functions to get the expanded states and the path. Starting with the expanded states, the function has 2 parameters, graph and start and initializes visited to None. Once in the function, an if statement checks if visited equals none. If it does, visited is initialized to a list and appended with start. Visited here is used to keep track of the visited nodes and will call recursively call the function dfs_recursive_states if the node has not been visited. When the function is called recursively path must be passed to the function so that it can keep an accurate record of the visited nodes. Otherwise, visited will be continually initialized as none if nothing is passed. dfs_recursive_paths uses yield to recursively generate all possible paths in the graph. The output of dss_recursive_paths is saved in a list which is passed to the function shortest_path. Shortest path then goes through the list and finds the shortest path and returns it. The output for recursive dfs on a vertex list is as follows:

```
Recursive DFS Vertex List (Graph 1)
State Expanded:
S,P,H,Q,E,R,F,C,A,B,D,G
Path Returned:
['S-D-C-F-G', 'S-E-R-F-G']
```

```
Recursive DFS Vertex List (Graph 2)
State Expanded:
S,P,Q,E,R,F,C,A,G,H,D,B
Path Returned:
['S-E-R-F-G']
```

Bfs on the vertex list follows a similar format that the recursive dfs funtions did. In bfs_queue_expanded the function takes 3 parameters called graph, start and goal. Path is also initialized to an empty list. Once in the function, visited and queue are initialized with the starting node. A while loop is then started that will terminate when the queue is empty and nothing remains in it. The node is the popped off and stored in the variable vertex. A for loop is then used to go through the dictionary and find all connected nodes. If there is a connected node that has not been visited it is appended to the visited list and added to the queue. This continues until the goal node is found or the queue is emptied. The function returns the path taken to the goal node or just the path explored in the graph if the goal node is not found. bfs_queue_paths takes three parameters, graph, start and goal. In the function, queue is then in initialized with the start node and list of connected nodes. A while loop is then started that runs until queue is empty. Vertex and path are initialized by popping off the node and a for loop is started that is used to find all connected nodes. The subtraction overload method used in the for loop ensures that visited nodes are not gone to again. Using yield, all possible paths are generated. The paths are stored in a list and sent to the function shortest_path which will iterate through the list and return the shortest path generated by the method. The output for bfs_queue_paths and bfs_queue_states is as follows:

```
Queue BFS Vertex List (Graph 1)
State Expanded:
S,P,E,D,H,Q,R,C,B,F,A,G
Path Returned:
['S-D-C-F-G', 'S-E-R-F-G']

Queue BFS Vertex List (Graph 2)
State Expanded:
S,P,E,D,H,Q,R,C,B,F,A,G,S,P,E,D,Q,R,H,C,B,F,A,G
Path Returned:
['S-E-R-F-G']
```

dfs_stack_adj operates in a similar fashion that the dfs using stack for the vertex list operates. The function takes 3 parameters, graph, start and goal. In the function, stack is initialized with the starting row of the matrix. If path has not been initialized it will be initialized as a list. Visited is initialized as a list with the same number of indexes as the length of the matrix. A while loop goes through the stack, popping off the node element and appending it to the states list using the letter list so that element numbers can be translated to the nodes. If the node is the goal node G, the function returns the path and expanded states. If the index of visited node that is equivalent to node is equal to false, that index is set equal to true and appended to path using letter_list. The for loop then goes through the columns to check for any connections to

other nodes by checking if the node is not equal to 0. If so, the node is added onto stack. The while loop ends when the goal node is found or all nodes have been explored. dfs_stack_adj outputs the following:

```
Stack DFS Adjacency List (Graph 1)
State Expanded:
S,P,S,Q,P,H,Q,P,E,S,R,F,R,G
Path Returned:
S-P-Q-H-E-R-F-G

Stack DFS Adjacency List (Graph 2)
State Expanded:
S,P,S,Q,P,H,Q,P,E,S,R,F,R,G,S,P,Q,E,R,F,G
Path Returned:
S-P-Q-E-R-F-G
```

dfs_recursive_adj takes 3 parameters: graph, curr_node and goal. It also initiates visited and path to none and states as an empty list. In the function, if visited is equal to none, it is initiated to the size of the matrix and each index is occupied with false. If path is equal to none it is initiated to an empty list. If when the index of the node when inputted to visited is equal to false, it is changed to true and appended to path. If the current node is equivalent to the index of goal, path is saved to a global variable calls dfs_recursvie_states for the expanded states. If the current node is not the goal node, a for loop begins and goes through the matrix columns looking for unvisited indexes. If that is found, the graph, node, goal, visited list and path list are passed to dfs_recursive_adj and recursively called until the goal index is found or all nodes have been visited. The output for dfs_recursive_adj is as follows:

```
Recursive DFS Adjacency List (Graph 1)
State Expanded:
S,D,B,A,C,F,G,R,E,H,P,Q
Path Returned:
S-D-B-A-C-F-G

Recursive DFS Adjacency List (Graph 2)
State Expanded:
S,D,B,A,C,E,H,P,Q,R,F,G
Path Returned:
S-D-B-A-C-E-H-P-Q-R-F-G
```

bfs_queue_adj_states_path has 3 parameters: graph, start and goal. Visited is initiated to the size of graph and each index is occupied with false. Queue is initiated with the starting row from the graph matrix. States is appended with the starting node when it is equal to 0. The while loop will run as the queue is not empty. The index of the node is then stored in node. If the index in visited equivalent to node is equal to false, it is set to true and appended to path. The for loop then goes through the matrix column of the node looking for any connections to the node represented by a 1. If the connection is not logged in visited, it is added to queue and

append to states. The while loop continues until all nodes have been visited and returns path and states. bfs_queue_adj_states_path returns the following:

```
Queue BFS Adjacency List (Graph 1)
State Expanded
S,D,E,P,B,C,E,H,R,H,Q,A,A,F,Q,F,G
Path Returned:
S-D-E-P-B-C-H-R-Q-A-F-G

Queue BFS Adjacency List (Graph 2)
State Expanded
S,D,E,P,B,C,E,H,R,H,Q,A,A,F,Q,F,G
Path Returned:
S-D-E-P-B-C-H-R-Q-A-F-G
```

The file HW1_G3_G4 imports collections and from queue imports Queue and PriorityQueue. This file will perform a Uniform Cost Search (UCS) on a vertex list and adjacency matrix. The vertex list stores the graph in a dictionary with keys being the nodes and values are tuples storing the connected nodes and weights. The adjacency matrix stores the weights in the indexes.

ucs_vertex_list has three parameters: graph, start and goal. In the function, q is initialized as a priority queue and the starting key put into it. Visited and state are both initialized and state initialized with the starting index. Starting up a while loop while q is not empty, the current node and weight are stored. If the current node is not in the visited list, it is appended into it. If the current node is the goal visited and state are returned. The for loop takes the next connected node and its weight from the inputted graph and if it has not been visited yet, it is appended to visited and the node and its weight are put into the priority queue. This continues until the goal is found or all nodes have been visited. ucs_vertex_list outputs the following:

```
Priority Queue UCS Vertex List (Graph 3)
State Expanded:
S,E,P,D,E,C,B,A,C,F,F,H,R,H,R,R,G,R,G
Path Returned:
S-D-B-A-C-E-F-G

Priority Queue UCS Vertex List (Graph 4)
State Expanded:
S,P,D,E,B,C,E,A,H,R,H,R,P,Q,P,Q,Q,Q,Q,F,G
Path Returned:
S-D-B-A-C-E-H-P-Q-R-F-G
```

ucs_adjacency_list has 3 parameters: graph, start and goal. In the function, q is initialized as a priority queue and the index and weight of the starting node are stored. Visited in initialized with the starting index and state stores the starting node index from the matrix translated into the letter of the node using letter_list. The while loop runs while q is not empty. The current node and weight are stored from q. If the current node is not in visited it is appended on to the list. If the current node is the goal, a list comprehension translates the visited state index numbers into their respective letters and stores it in path. It is returned along with the state expansion. The for loop runs for the number of columns present in the graph matrix looking for any connections. The connections are signified by a number that isn't 0 and is the weight of the connection to another node based on the index in the matrix. If it isn't in the visited list it is appended to the state list and translated by letter_list. The node is then put into the priority queue. ucs_adjacency_list output is as follows:

```
Priority Queue UCS Adjacency Matrix (Graph 3)
State Expanded:
S,D,E,P,B,C,E,A,C,F,F,H,R,H,R,G,R,G,R
Path Returned:
S-D-B-A-C-E-F-G


Priority Queue UCS Adjacency Matrix (Graph 4)
State Expanded:
S,D,E,P,B,C,E,A,H,R,H,R,P,Q,P,Q,Q,Q,Q,F,G
Path Returned:
S-D-B-A-C-E-H-P-Q-R-F-G
```