William Dahl
Artificial Intelligence ICSI 435
HW1

1.) **Explanation of Implementation**

- **DFS**
  - **Recursive Implementation**
    - start at S
    - loop through the neighbors of the current node adding each neighbor to the path
    - check if the neighbor had already been visited
    - if not, recursive call the function with the neighbor as the start
    - check if start is the goal
    - if true return
    - if there are no more neighbors that haven't been visited then go back up a recursive level and pop the most recent node off the path
  - **Stack**
    - start at S
    - loop through stack popping on each iteration
    - if the node just popped is the goal return
    - else loop through neighbors for the node popped
    - add those to the stack
    - if no neighbors for node that haven't been visited pop most recent node off of path
- **BFS**
  - **Recursion**
    - start at S
    - check if start contains any entries
    - loop through the nodes in start
    - if node is goal get path by retracing through parent list and return
    - else loop through neighbors of the current node adding each new node to the next level
    - also add new node as an entry tot he parents list with a pointer to its parent
    - recursive call function with next level as the start
  - **Queue**
    - start at S
    - loop through queue popping the first entry in every iteration
    - check if node had been visited
    - check if node is goal
    - if node is goal make path by retracing path through parents list
    - reverse path from parents list so that it is the right way
    - else loop through the neighbors of the current node and add them to the queue
- **UCS**
  - **Undirected**
    - start at S
    - get neighbors of S
    - loop though the neighbors getting their weighted edges and adding them to the propriety queue
    - loop through the queue
    - pull highest priority
    - check if node is goal
    - if node is goal loop through the queue pulling the highest priority and comparing the cost to the cost to get to the goal
    - if cost is equal add to a list
    - loop through the list and find each nodes parent in the visited nodes starting at S
    - add G if not already in the path

        - return
        - else get the neighbors for the current node and add them along with the total cost to get to them into the priority queue
    - **Directed**
        - start at S
        - get neighbors of S
        - loop though the neighbors getting their weighted edges and adding them to the propriety queue
        - loop through the queue
        - pull highest priority
        - check if node is goal
        - create path by retracing through the parents list
        - reverse path so that it is the right way
        - return
        - else get the neighbors for the current node and add them along with the total cost to get to them into the priority queue
        - add the neighbor as an entry into he parents list with a pointer to its parent
    - **get_neighbors()**
        - loops through he neighbors of the node and returns a list containing the the neighbors to the node
    - **get_weight_of_edge()**
        - returns the weight of the edge between two nodes in the graph
- **PriorityQueue**
    - uses heapq to create a priority queue using a heap
    - is_empty() returns true if the queue is empty and false otherwise
    - insert_with_priority() pushes the edge along with its weight onto the heap
    - pull_highest_priority() returns the lowest priority element in the heap multiplied by -1 because in the UCS algorithm the lower the number the higher the priority

2.) **Results**

- **BFS**
    - **Undirected**
        - States Expanded: ['S', 'D', 'E', 'P', 'B', 'C', 'H', 'R', 'Q', 'A', 'F', 'G']
        - Path Returned: ['S', 'D', 'C', 'F', 'G']
        - *out put as list/matrix and recursion/queue was the same
    - **Directed**
        - States Expanded: ['S', 'D', 'E', 'P', 'B', 'C', 'H', 'R', 'Q', 'A', 'F', 'G']
        - Path Returned: ['S', 'E', 'R', 'F', 'G']
        - *out put was same for list/matrix and recursion/queue
- **DFS**
    - **Undirected**
        - States Expanded: ['S', 'D', 'B', 'A', 'C', 'F', 'G']
        - Path Returned: ['S', 'D', 'B', 'A', 'C', 'F', 'G']
        - *output was same for list/matrix and recursion/stack
    - **Directed**
        - States Expanded: ['S', 'D', 'B', 'A', 'C', 'E', 'H', 'P', 'Q', 'R', 'F', 'G']
        - Path Returned: ['S', 'D', 'E', 'R', 'F', 'G']
        - *output was the same for list/matrix and recursion/stack
- **UCS**
    - **Undirected**
        - States Expanded: ['S', 'P', 'D', 'B', 'H', 'E', 'A', 'R', 'C', 'Q', 'F', 'G']
        - Path Returned: ['S', 'D', 'E', 'R', 'F', 'G']
        - *Output was the same for list/matrix

- **Directed**
  - States Expanded: ['S', 'P', 'D', 'B', 'E', 'A', 'R', 'F', 'C', 'G']
  - Path Returned: ['S', 'D', 'E', 'R', 'F', 'G']
  - *Output was the same for list/matrix

3.) **Imported modules**

- heapq