Project 3: Graph algorithms

COT 4400, Fall 2023

Due: Dec 2nd, 2023

By: *Aidan Khalil (*U9240-8495)

PROJECT REPORT (*Modeling the Problem*)

1. What type of graph would you use to model the problem input (detailed in section 3.1), and how would you construct this graph? (i.e., what do the vertices, edges, etc., correspond to?) Be specific here; we discussed a number of different types of graphs in class.

To model this problem input, I would implement an adjacency list. The adjacency list will be constructed from reading and parsing the input file stream "input.txt." Additionally, I will maintain an edge list and a vertex list to ensure a robust graph implementation. I chose an adjacency list over a matrix because it would be more space-efficient, especially considering the sparse nature of the provided "Apollo and Diana" sample inputs.

Graph Construction would consist of the following:

- Vertices: Each arrow in the maze corresponds to a vertex in the graph. Vertex attributes include color ('R' or 'B') and direction ("N", "E", "S", "W", "NE", "SE", "SW", "NW"). The bullseye at the bottom-right has no outgoing edges, a color of 'O' and no direction.
- Edges: Directed edges connect vertices based on the maze arrows. Red arrows point to blue arrows, and blue arrows point to red arrows (direction logic for traversing the matrix will be implemented). Edges represent valid moves, enforcing the alternating color pattern. \
- Adjacency List: For each vertex, maintain a list of adjacent vertices to represent the edges.

2. What algorithm will you use to solve the problem? You should describe your algorithm in pseudocode. Be sure to describe not just the general algorithm you will use, but how you will identify the sequence of moves Apollo must take to reach the bullseye goal. Your algorithm must be correct, and it must have the minimum possible complexity.

I will use a breadth-first search algorithm to solve the problem by starting from the top-left arrow, enqueuing it and creating a loop of "while the queue is not empty": dequeue a vertex (arrow). If the vertex is the bullseye, break the loop. Enqueue unvisited neighbors, marking them as visited and updating parent pointers would allow for backtracking on endless loop detection.

To specifically construct the moves, the algorithm would traverse the path stack and return the moves as a string stream. A full pseudocode is provided on the page below:

Pseudocode for function bfs():

```
    initialize queu and path stack

    start BFS from the top-left arrow (enqueue top-left vertex)
    mark top-left vertex as visited (.visited = 1;)
    while queue is not empty:
      current = dequeue from queue
       if current is bullseye:
          break (found a move to the goal!)
       if current not in path:
          push current to path stack
          for each neighbor of current:
             if neighbor is not visited:
                enqueue neighbor, mark as visited
                set neighbor's parent to current
                set neighbor's move weight
       else:
          pop from path stack


move = construct_moves(current, path stack):
    initialize string called moves or travelPath
   while path stack is not empty
      vertex = pop from path stack
      append move weight and direction to moves (i.e. 5E, 4SW, ..)
    return moves

    return move
```