CS 330 HW 3

Chandini Toleti
U29391556

(1.1) ALGORITHM

Input ← Graph G, an undirected graph where each one of three colors: r,b,g
Output ← Paths, a 2-D array that stores tuples of 4, representing any *colorful* paths outlined in the problem (returns an empty grid if no colorful paths)

```
def colorful_paths(G):

Initialize:
paths← Initialize an empty 2-D array or similar dynamic data structure

do:
for u in G do:
        //loop through every node in G
        for v in G[u] do:
                //first check neighbors for red edges between them
                if G[u][v] == 'r' do:
                        for l in G[v] do:
                                //if there's a red neighbor, we check the nieghbor's neighbors
                                to see if a blue path between them exist
                                if G[v][l] == 'b' do:
                                        for m in  G[l] do:
                                                //similarly, if a blue path exists after a red path,
                                                we check the node's neighbors again to see if the
                                                path ends in a green edge
                                                if G[l][m]== 'g' do:
                                                        //finally, at this point we've found a path
                                                        of four that follow the colorful path
                                                        outlined in the problem
                                                        path.append((u,v,l,m))
return path
//algorithm done! Returns any paths found (or an empty grid if none are found)
```

(1.2) PROOF OF CORRECTNESS

Consider the direct proof:

Assertion 1: If a path Is not 'colorful' the algorithm will not count it. Consider a path like blue, red, green. The algorithm will observe the first edge as blue, which doesn't match the pattern, and therefore won't check for the rest of the pattern. Instead, it will move on. Say the red edge has its own path red, green, blue. The algorithm will observe the red edge, then continue to check the next step of the pattern. It will not find blue after and will continue to a different node. Say another node has the path red, blue, blue. The algorithm will observe the red edge as the first step of the pattern.It will see the next edge, blue, matches the pattern. However, the algorithm will see the last edge is not a match and move on without appending. In any instance where a path does not meet the pattern, the algorithm will not count it.

Assertion 2: If a 'colorful' path exists in Graph G, the algorithm will find it. Consider the direct proof: The first for-loop implies every node in G is considered. Additionally, the second for-loop implies every edge of every node is also considered. If an edge is red and meets the first step, then the next for-loop will consider all of the red node's edges. If any of these edges are blue, they meet the next step. Consequently, the algorithm will check all of the blue nodes' edges. Finally, if any of these meet the final step and are green, the algorithm will append the found path to "paths". If any path meets the criteria, it will be saved and outputted. Any other path will have no impact on the output.

(1.3) SPACE AND TIME ANALYSIS

To analyze space complexity, first consider the inputs and outputs.

1) Graph G will have a space complexity of O($n + m$), for n nodes and m edges
2) Paths[][] will have a space complexity of O($4p$)= O(p), where p is the number of valid paths found. In the worst case, p can have an extreme upper bound of O(m· D^2)

Finally, the algorithm operates in constant space O(1), as no extra space is created besides the inputs and outputs. Finally, the overall space complexity is O($n + m + p$), wherein the rare, worst-case p = m· D^2($if\ every\ path\ is\ colorful, the\ \mathrm{m} \cdot \mathrm{D}^2\ is\ the\ maximum\ number\ of\ paths$).

To analyze time complexity, consider all steps of the algorithm. In this instance, *n* will represent the number of nodes, *m* will represent the number of edges, and *D* will represent the maximum degree of a graph

(1) (for u in G) this step traverses *every node* and has a time complexity of *O(n)*
(2) (for v in G[u]) this step traverses *every edge* of node u and therefore has a worst-case time complexity of O(m)

Together the first two steps have time complexity O(n+m)

(3) (for l in G[v]) this step traverses every edge of node v. This is also known as the *degree* of v and therefore has a worst-case time complexity of O(D)

(cont. on next page)

(4) for m in G[l]: this step also traverses every edge of node l. This is also known as the *degree* of l and therefore has a worst-case time complexity of O(D)

In the worst-case, finding all the colorful paths therefore takes $O(n + m \cdot D^2)$ time. This is because, in the worst-case, we traverse the degree of every edge twice and every node once.

(2.1) ALGORITHM

Input ← Map *M* represented by an *n x n* 2-D array

Output← Pairs is a Hash Set storing pairs of islands where bridges can be built between them

#This algorithm implements BFS, but specifically an implementation of BFS where the algorithm takes a position of water (M[i][j] in the form of i,j,and M) and returns an accurate adjacency list of islands reachable within at most 2 steps. This implementation is written in pseudocde as a helper function below:

```
def bfs (i,j,M):
bfsAdjList ← initialize to empty #will store adjacent islands
directions = {(1,0),(-1,0), (0,1), (0,-1)}

#iterate through all pairs in directions
for (a,b) in directions do:
        stepCount←0
        x←i
        y←j

        while stepCount < 2 do:
            x+= (a)
            y+= (b)
            stepCount++

            if x>=0 AND x<n AND y>=0 AND y<n do:
                if M[x][y] != '—' do:
                    append. bfsAdjList(M[x][y])
                    break; #cannot go through islands
return bfsAdjList

#this function resembles the main algorithm, where we work with our BFS implementation to return the pairs outlined in outputs using Map M outlined in inputs

def islands(Map M):
Initialize:
Pairs←Hash Set (key, value), where no pair can be repeated regardless of order
```

(cont. on next page)

```
for i in range (n) do:
        for j in range(n) do:
                if M[i][j]== '—' do:
                        #initialize adjList with output of my bfs implementation
                        adjList ← bfs(i,j,M) * # thrown away after each iteration of j

                        #put adjList values into HashSet, so pairs cannot be repeated
                        for k from 0 to length(adjList) – 1 do:
                                for m from k + 1 to length(adjList) – 1 do:
                                        tempPair← (list[k], list[m]) #initialize temp pair, thrown
                                away when m iterates
                                        add tempPair to Pairs
return Pairs
```

(2.2) PROOF OF CORRECTNESS

Consider the direct proof:

Assertion 1: If there are two or more islands less than 2 steps away from each other, their letters will be added to the adjacency list. The function, islands, checks every element in the Map for water and calls BFS when water is found. By observation of this BFS implementation, we see that BFS finds every single island within 2 steps of that block of water. Since we check every water element, if there exists two islands within a 2 step reach, these two functions will add the island to the adjacency list.

Assertion 2: If there exists duplicate or identity pairs, they will not appear in the final output. By observation of a HashSet, duplicate pairs will be reduced to unique pairs. For example, {(3,2),(2,3),(3,2)} may be reduced to (2,3). Additionally, identity pairs like (3,3) will not be added.

By assertion 1, every island reachable will have a letter added to the adjacency list. By observation of the code, every element of the adjacency list will be paired with every other element of the adjacency list and fed into a HashSet. By Assertion 2 of HashSets, we see the final product will only be unique, non-identity pairs. In the end, we see unique pairs that represent pair Islands close enough to build a bridge between them.

(2.3) SPACE AND TIME ANALYSIS

To analyze space complexity, first consider the inputs and outputs

(1) Map M: a 2-D array of space O(n^2)
(2) Pairs: a HashSet of pairs has space complexity O(2p)= O(p), where p is the number of pairs that a bridge can be built between

Now, consider additional space allocated within the algorithm

(3) bfsAdjListL: worst-case space efficiency of O(8)= O(1)
(4) adjList: worst-case space efficiency of O(8)= O(1)
(5) dircetions: constant efficiency O(8)= O(1)

(cont. on next page)

All-in-all, the algorithm has worst-case space efficiency O(n^2).

To analyze time complexity, consider each step of the algorithm

(1) The first two "for" loops can consider every element in Map m and therefore have a worst-case time efficiency of O(n^2)
(2) Next, we call bfs. Bfs has worst-case time efficiency O(8), so we consider this call an operaton of constant time O(1)
(3) The next two nested for loops fill the has table with the current adjacency list. In the worst case, the adjacency list will contain 8 different islands for the 8 different comparisons made (see (2) above). Therefore, we say this operation runs in constant time O(1)

All-in-all, my runtime in the worst-case scenario adds up to O(n^2).

(cont. on next page)