COMP 6651 - ALGORITHM DESIGN TECHNIQUES
(WINTER 22)
PROJECT - PART 1

TEAM MEMBERS:

1. Abdul Aziz Raza Shaik - **40182726**
2. Dhananjay Narayan - **40164521**
3. Mohammed Rafique Contractor - **40189407**
4. Nadib Hussain - **40186920**
5. Tanzia Ahmed - **40166924**

CASE 1

1a. Describe the graph that you need to build in order to reformulate the problem as a one-way street problem, with a formal definition of the set of nodes and the set of edges.

Floor Map 1 : Rose Medical Building

G = (V, E) where V : set of nodes & E : set of undirected edges

V = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 }

E = { (0,1), (0,12), (1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,8), (8,9), (9,10), (10,11), (11,12) }

Floor Map 2 : Victoria Hospital

G = (V, E) where V : set of nodes & E : set of undirected edges

V = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

E = { (0,1), (0,6), (0,10), (1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,8), (8,9), (9,10) }

1b. Describe a greedy or a DFS algorithm (DFS/Greedy Algorithm 1) that allows the checking that one feasible (wrt to the one-way street problem) graph orientation exists. Discuss whether your algorithm is an exact one or a heuristic one.

- To check one feasible graph orientation for a one-way street problem, we need to check - i) if the graph is strongly connected and ii) if there are no bridges in the graph[1]. If the graph is not fully connected our program will not process any further. Also in Case1, if there are any bridges found in the graph it will violate the one-way street problem and the program will halt.
- To check the strong connectivity of the graph we have implemented a depth-first search (DFS) algorithm:

```
Start <- input node
Connected -> output/false

Visited <- All nodes false
Stack <- start node

While stack is not empty:
    U <- stack.pop()
    If u is not visited:
        Visited[u] <- true
        For each v in adjacent(u):
            If v is not visited:
                stack.push(v)
If visited_nodes_count == V:
    Connected <- true
Return connected
```

- Therefore, it will assure that the graph is fully connected and possibly we can get a one/two-way solution. Now, we need to check if there is any bridge in this graph. And we can check it by a modified DFS algorithm as follows:

```
bridge():
Disc[ ] <- V : current node //discovered node
Low [ ] <- V : it stores the minimum node connected with the node
Parent [ ]: V
While u in V:
    If u is not visited:
        For each v adjacent(u):
```

```
                    If v is not visited:
                        bridgeUtil(i, visited, disc, low, parent)
bridgeUtil(u, visited, disc, low, parent):
    Visited[u] <- true
    disc[u] <- low[u] <-  time <-  time+1
    For each v in adjacent(u):
        If v is not visited:
            Parent[v] <-  u
            bridgeUtil(v, visited, disc, low, parent)
            Low [u] <- min (low[u], low[v])
            If low[v]>disc[u]:
                Return (u,v) //bridge edge
            else if v is not parent[u]:
                low[u] <-  min(low[u], disc[v])
```

- Therefore, if the bridge function returns null then there is obviously a one-way path possible for the graph.
- Our algorithm is an exact one. We have not considered any heuristic for this algorithm.

1c. What is the complexity of that algorithm? Provide the justification of that complexity.

- The time complexity for checking the first condition of the algorithm i.e. if the graph is strongly connected is O(E+V) (as we are using an adjacency list graph).Since we are using the DFS algorithm only to check the connectivity only the algorithm traverses the non-visited nodes and its adjacent nodes.
- The time complexity for checking the second condition takes about the same time since we are using the DFS algorithm here also with addition of three more arrays. So, the complexity is O(E+V).
- Therefore, the total complexity becomes O(E+V)+O(E+V)=~O(E+V).

1d. Describe the algorithm (Orientation Algorithm 1) that defines the orientation. Provide its complexity and the justification of it.

- The Orientation 1 algorithm has been implemented with the depth-first search logic. As we traverse from the start node all through its child and grand-child nodes, we keep drawing an edge from the parent to the child node. For example, if the DFS algorithm traverses from A (parent) to B(child) then the graph will add B adjacent to A like A->B.
- For the last node, if it is connected to the start node, in other words, if start and end nodes are the same, then it is directed towards the start node in spite of the start node being already visited.
- For case 1, we considered the orientation to be in an anti-clockwise direction. This means that it starts from 0->1 and ends in 12->0. For example, 0->1->2->3…->11->12->0. This forms a one-way path orientation.
- The complexity of this algorithm is O(E+V). Since, we are using depth first search algorithm on an adjacency list graph the traversal is o(E+V) in the worst case scenario.

1e. Implement both algorithms and make them running for the data sets. Provide the source files, figures of the floor map and of the graph, as well as the direction of the edges.

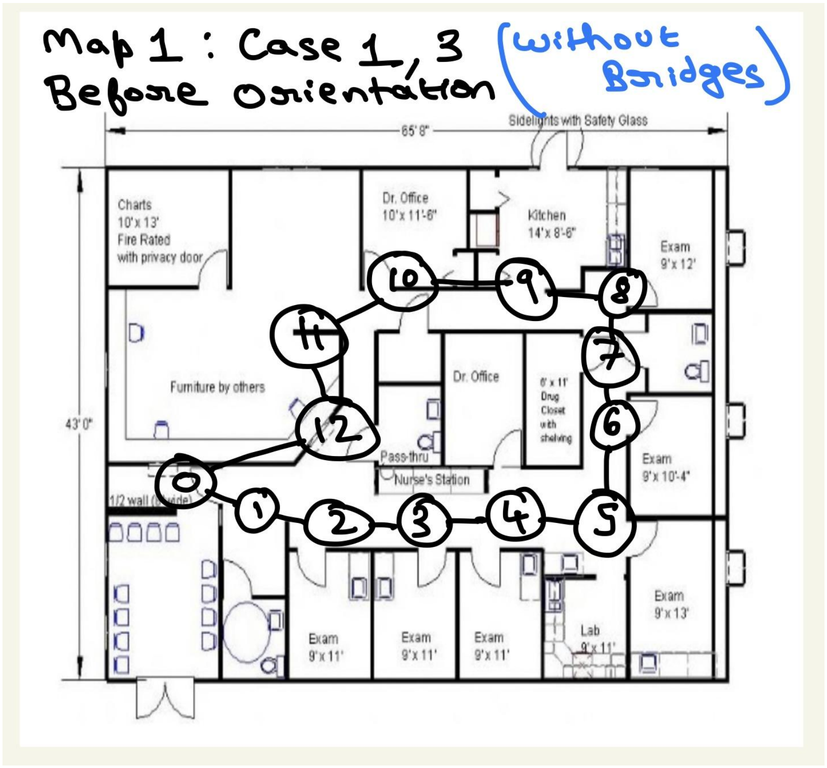

Fig 1: Map 1 - Undirected Graph for Case 1

Map 1 : Case 1
After Orientation

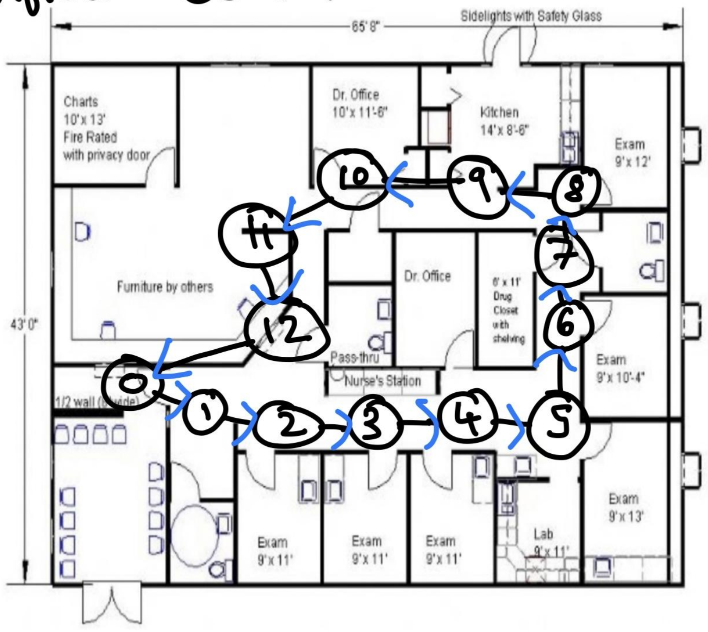

Fig 2: Map 1 - Directed Graph for Case 1

Map 2 : Case 1,3 Before Orientation

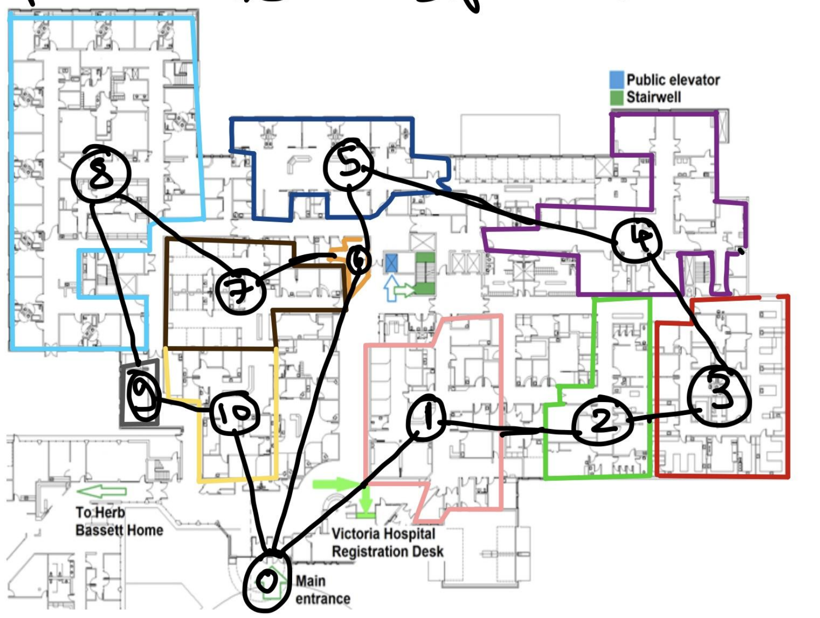

Fig 3: Map 2 - Undirected Graph for Case 1

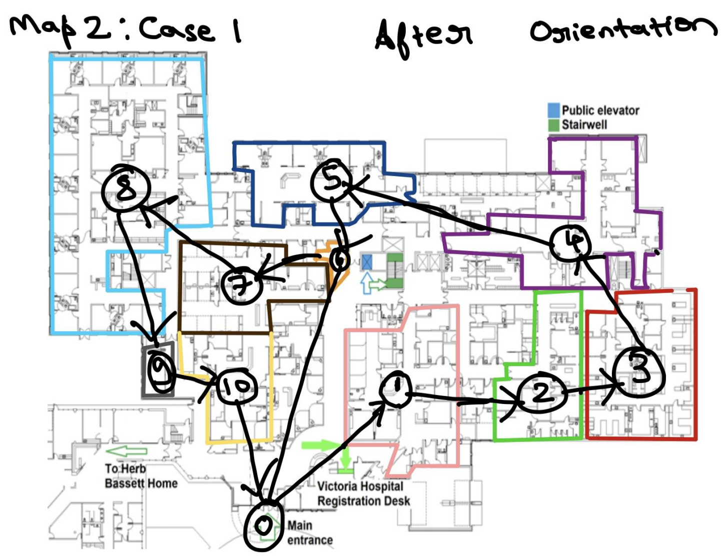

Fig 4: Map 2 - Directed Graph for Case 1

1f. Provide an example in which the problem is not equivalent to the one-way street problem, and for which, if you solve it as a one-way street problem, you may require more than what is needed.

The graph in the following figure is not equivalent to the one-way street problem. It would not satisfy the two conditions of the one-way street problem. Though the graph is a connected one, the edge “ (3, 4) “ forms a bridge. Upon removing this edge, we would get two graphs. Solving it as a one way problem, the edge “ (3, 4)“, eventually should become bi-directional, which is not ideal for the problem. So any example of a graph with a bridge would not be equivalent to a one-way street problem.

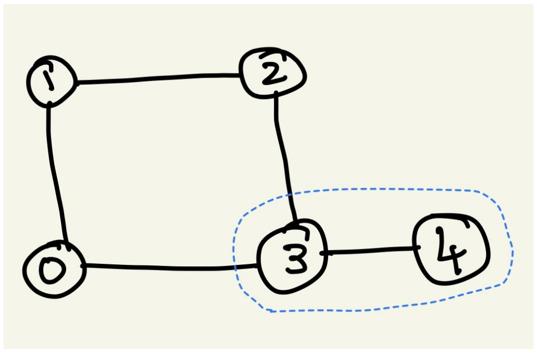

Fig 5: Example of a Graph that cannot have one-way orientation

CASE 2

2a. Describe the graph that you need to build in order to reformulate the problem as a one/two-way street problem, with a formal definition of the set of nodes and the set of edges.

Floor Map 1 : Rose Medical Building

G = (V, E) where V : set of nodes & E : set of undirected edges

V = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}

E = { (0,1), (1,2), (1,13), (2,3), (3,4), (4,5), (5,6), (6,7), (7,8), (8,9), (9,10), (10,11), (11,12), (11,13) }

Floor Map 2 : Victoria Hospital

G = (V, E) where V : set of nodes & E : set of undirected edges

V = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

E = { (0,1), (0,6), (0,10), (1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,8), (8,9), (9,10), (10,11) }

2b. Describe a DFS or a greedy algorithm (DFS/Greedy Algorithm 2) that allows the checking that one feasible (wrt to the one/two-way street problem) graph orientation exists. Discuss whether your algorithm is an exact one or a heuristic one.

- To check one feasible graph orientation we need to check - i) if the graph is strongly connected and ii) if there are no bridges in the graph. If the graph is not fully connected our program will not process any further. Also in Case2, bridges are not a problem.
- To check the strong connectivity of the graph we have implemented a depth-first search (DFS) algorithm:

```
Start <- input node
Connected <- output/false

Visited <-  All nodes false
Stack <- start node

While stack is not empty:
        U <- stack.pop()
        If u is not visited:
                Visited[u] <- true
                For each v in adjacent(u):
                        If v is not visited:
                                stack.push(v)
If visited_nodes_count == V:
        Connected <- true
Return connected
```

- Therefore, it will assure that the graph is fully connected and possibly we can get a one/two-way solution. Now, we need to check if there is any bridge in this graph. And we can check it by a modified DFS algorithm as follows:

```
bridge():
Disc[ ] <- V : current node //discovered node
Low [ ] <- V : it stores the minimum node connected with the node
Parent [ ]: V
While u in V:
        If u is not visited:
                For each v adjacent(u):
                        If v is not visited:
                                bridgeUtil(i, visited, disc, low, parent)
bridgeUtil(u, visited, disc, low, parent):
        Visited[u] <- true
        disc[u] <-  low[u] <- time <-  time+1
        For each v in adjacent(u):
                If v is not visited:
                        Parent[v] <- u
                        bridgeUtil(v, visited, disc, low, parent)
                        Low [u] <- min (low[u], low[v])
                        If low[v]>disc[u]:
                                Return (u,v) //bridge edge
                        else if v is not parent[u]:
                                low[u]  <- min(low[u], disc[v])
```

- Therefore, if the bridge function returns null then there is obviously a one-way path possible for the graph. But if there is any bridge we have to make it two-way street.
- Our algorithm is an exact one. We have not considered any heuristic for this algorithm.

2c. What is the complexity of that algorithm? Provide the justification of that complexity.

- The time complexity for checking the first condition of the algorithm i.e. if the graph is strongly connected is O(E+V) (as we are using an adjacency list graph).
- Since we are using the DFS algorithm only to check the connectivity only the algorithm traverses the non-visited nodes and its adjacent nodes.
- The time complexity for checking the second condition takes about the same time since we are using the DFS algorithm here also with addition of three more arrays. So, the complexity is O(E+V).
- Therefore, the total complexity becomes O(E+V)+O(E+V)=~O(E+V).

2d. Describe the algorithm (Orientation Algorithm 2) that defines the orientation. Provide its complexity and the justification of it.

- The Orientation 2 algorithm has been implemented with the depth-first search logic. As we traverse from the start node all through its child and grand-child nodes, we keep drawing an edge from the parent to the child node. For example, if the DFS algorithm traverses from A (parent) to B(child) then the graph will add B adjacent to A like A->B.
- Moreover, the nodes where bridges are formed are connected with its parent node with a backedge. Thus, making the directions bidirectional. For example, if A->B is a bridge then we make it A ⇔ B.
- For the last node, if it is connected to the start node, in other words, if start and end nodes are the same, then it is directed towards the start node in spite of the start node being already visited.
- For case 2, we considered the orientation to be in an anti-clockwise direction. And used backedges wherever required to make it a two-way street.
- The complexity of this algorithm is O(E+V) the same as before since the complexity of adding back edges is 1 due to the adjacency list.

2e. Implement both algorithms and make them run for the data sets. Provide the source files, figures of the floor map and of the graph, as well as the direction of the edges.

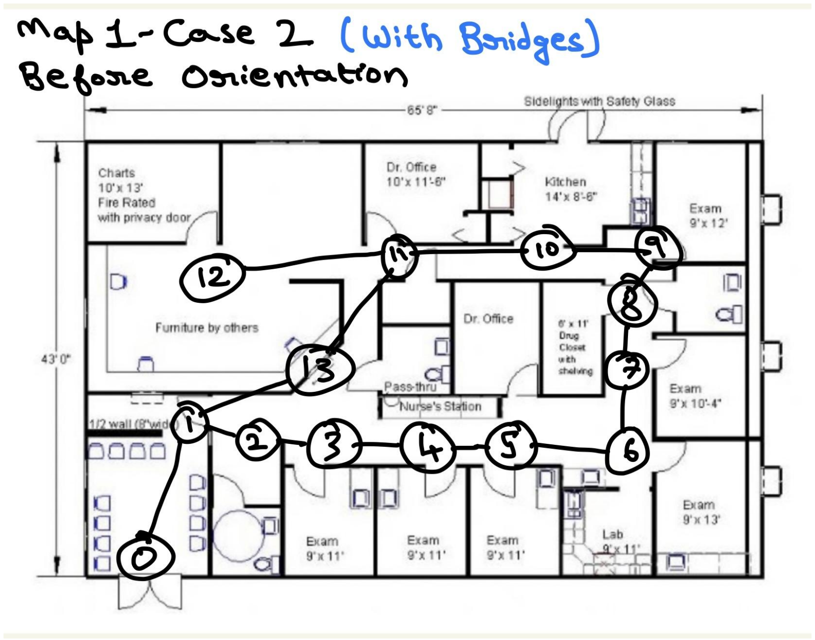

Fig 6: Map 1 - Undirected Graph for Case 2

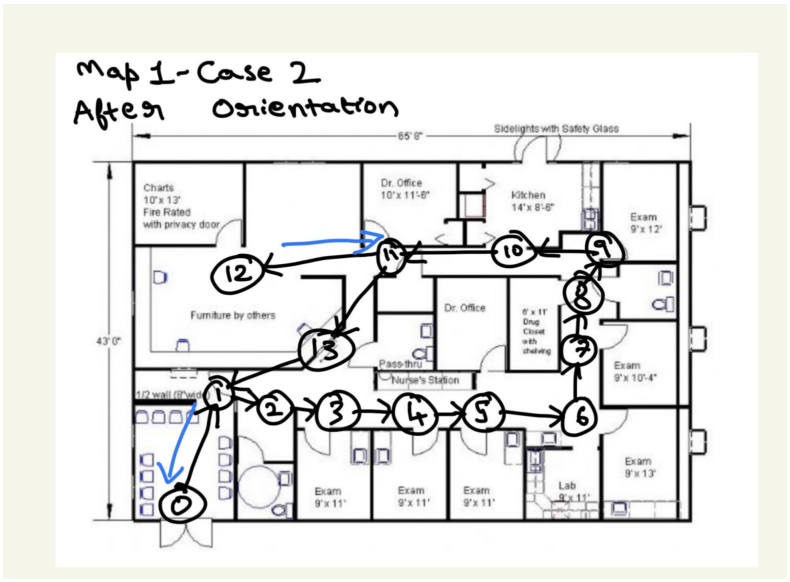

Fig 7: Map 1 - Directed Graph for Case 2

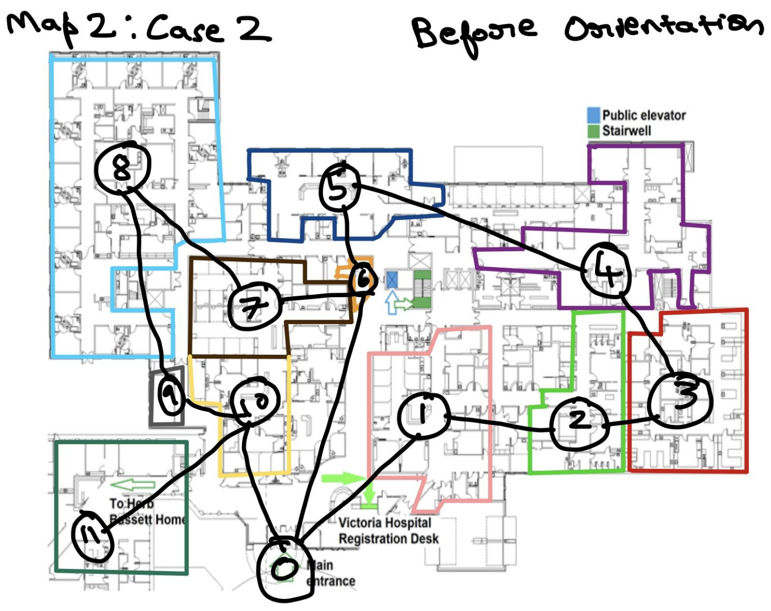

Fig 8: Map 2 - Undirected Graph for Case 2

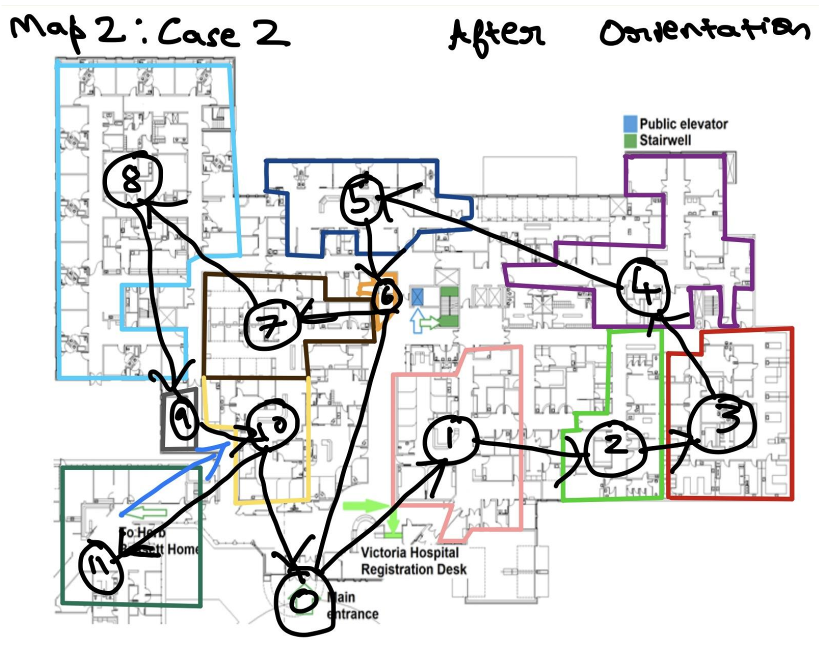

Fig 9: Map 2 - Directed Graph for Case 2

<u>CASE 3</u>

3a. Describe the graph (if different from previous questions) that you need to build in order to reformulate the problem as a one/two-way street problem, with a formal definition of the set of nodes and the set of edges.

<u>Same Graph Definition as Case 1</u>

3b. Describe a DFS or greedy algorithm (DFS/Greedy Algorithm 3) that allows the checking that one feasible (wrt to the one/two-way street problem) graph orientation and coloring exists. Discuss whether your algorithm is an exact one or a heuristic one.

- To check one feasible graph orientation we need to check - i) if the graph is strongly connected and ii) if there are no bridges in the graph.
- If the graph is not fully connected our program will not process any further. Also in Case3, if there are any bridges found in the graph it will violate the one/two-way street problem and the program will halt.
- To check the strong connectivity of the graph we have implemented a depth-first search (DFS) algorithm:

```
Start <- input node
Connected <- output/false

Visited <- All nodes false
Stack <- start node

While stack is not empty:
    U <- stack.pop()
    If u is not visited:
        Visited[u] <- true
        For each v in adjacent(u):
            If v is not visited:
                stack.push(v)
If visited_nodes_count == V:
    Connected <- true
Return connected
```

- Therefore, it will assure that the graph is fully connected and possibly we can get a one/two-way solution. Now, we need to check if there is any bridge in this graph. And we can check it by a modified DFS algorithm as follows:

```
bridge():
Disc[ ] <- V : current node //discovered node
Low [ ] <- V : it stores the minimum node connected with the node
Parent [ ]: V
While u in V:
    If u is not visited:
        For each v adjacent(u):
            If v is not visited:
                bridgeUtil(i, visited, disc, low, parent)
bridgeUtil(u, visited, disc, low, parent):
    Visited[u] <- true
    disc[u] <- low[u] <- time <-  time+1
    For each v in adjacent(u):
        If v is not visited:
            Parent[v] <- u
            bridgeUtil(v, visited, disc, low, parent)
            Low [u] <- min (low[u], low[v])
            If low[v]>disc[u]:
                Return (u,v) //bridge edge
            else if v is not parent[u]:
                low[u] <-  min(low[u], disc[v])
```

- Therefore, if the bridge function returns null then there is obviously a one-way path possible for the graph.
- Our algorithm is an exact one. We have not considered any heuristic for this algorithm.

3c. What is the complexity of that algorithm? Provide the justification of that complexity.

- The time complexity for checking the first condition of the algorithm i.e. if the graph is strongly connected is O(E+V) (as we are using an adjacency list graph).
- Since we are using the DFS algorithm only to check the connectivity only the algorithm traverses the non-visited nodes and its adjacent nodes.
- The time complexity for checking the second condition takes about the same time since we are using the DFS algorithm here also with addition of three more arrays. So, the complexity is O(E+V).
- Therefore, the total complexity becomes O(E+V)+O(E+V)=~O(E+V).

3d. Describe the algorithm (Orientation/Coloring Algorithm 3) that defines the orientation and coloring. Provide its complexity and the justification of it.

- In our algorithm, we have constructed paths for both covid and non-covid patients. The paths are designed as such that the two paths do not collide in the same direction ever. Two paths can exist in the same lane/corridor but in different directions.
- The path for covid patients is red and for non-covid patients is marked as green.
- We have decided to choose an anti-clockwise direction for the red path and clockwise direction for the green path.
- Now, for the anti-clockwise path the direction is oriented in the same way as mentioned in 1d. But for clockwise direction, we have used the same DFS algorithm with a modification.
- Our graph contains a common exit/entry node. If we traverse in anti-clockwise direction from the entry node we will get the path for covid patients. Whereas, if we traverse in clockwise direction from the entry point we will get the path for non-covid patients. This choice of direction is simply done by which child node of entry/exit node we expand in the DFS algorithm. If the first child of the entry/exit node is expanded from the stack then we traverse the anti-clockwise or RED path. Otherwise, we traverse the clockwise or the GREEN path.
- For example, covid patient path = {0,1,2,3,4,5,6,7,8,9,0} and non-covid patient path = {0,9,8,7,6,5,4,3,2,1,0}.
- The complexity is like that of the DFS algorithm since we traverse two times for two different paths. That is, complexity is O(E+V) + O(E+V) = ~ O(E+V)
- Since we call the DFS function two times the complexity is higher than the other orientation algorithms.

3e. Implement both algorithms and make them running for the data sets (see Section 4). Provide the source files, figures of the floor map and of the graph, as well as the direction of the edges.

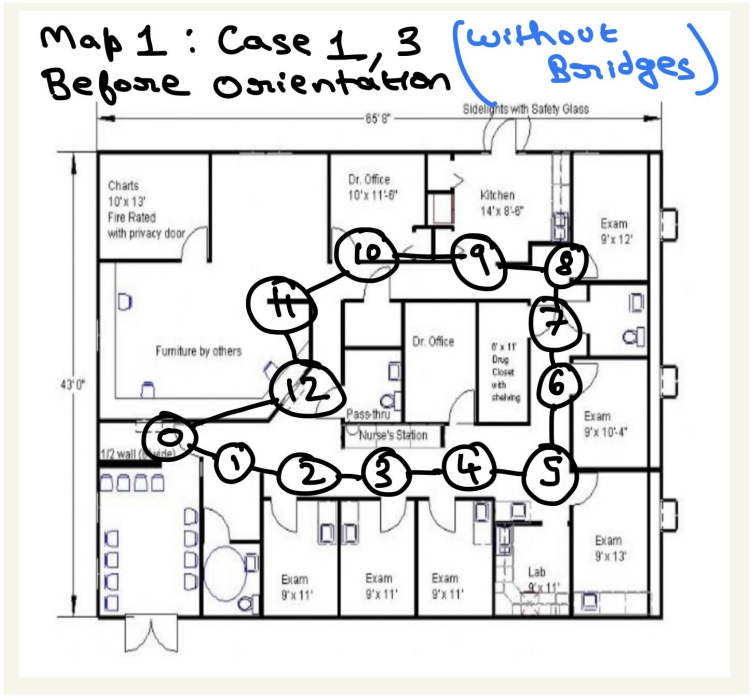

Fig 10: Map 1 - Undirected Graph for Case 3

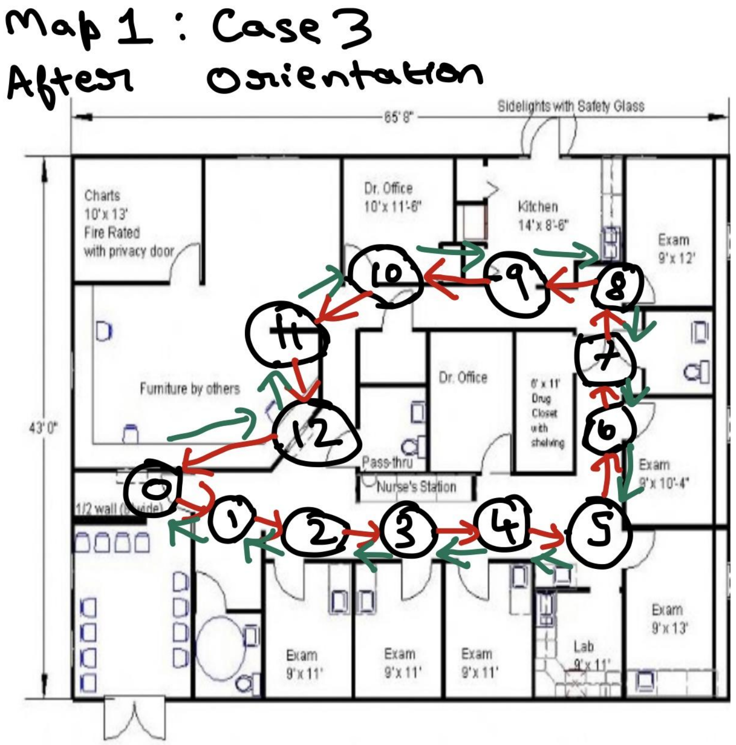

Fig 11: Map 1 - Directed Graph for Case 3

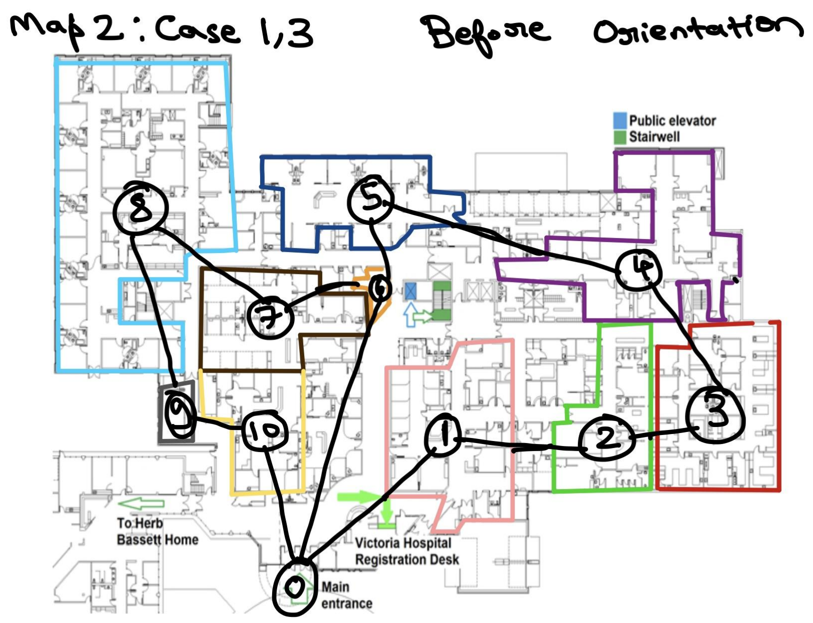

Fig 12: Map 2 - Undirected Graph for Case 3

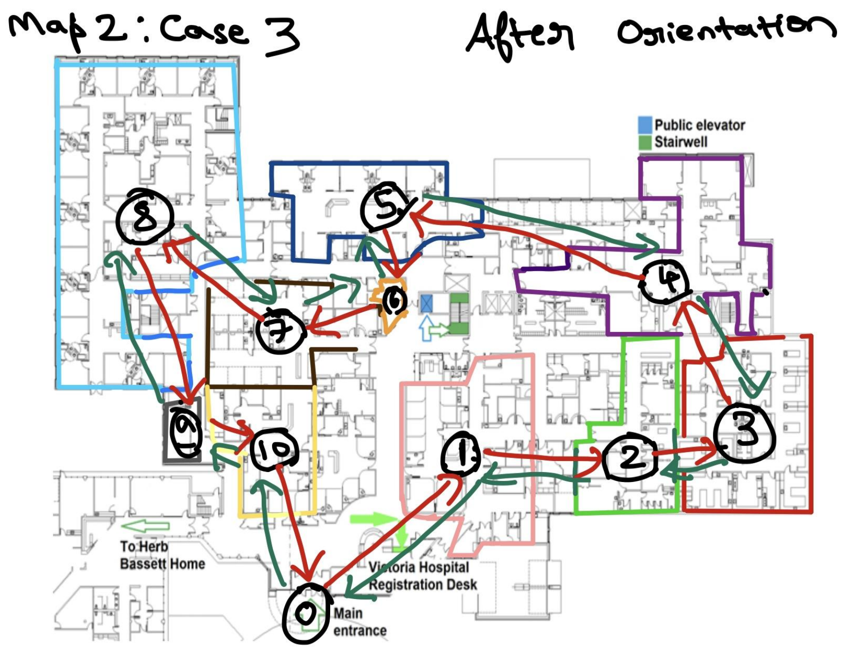

Fig 13: Map 2 - Directed Graph for Case 3

REFERENCES

[1]. http://appliedgraphtheory.weebly.com/one_way_road.html
[2]. https://www.geeksforgeeks.org/bridge-in-a-graph/