Path Orientation with Minimum Link Sharing

1st Manan Dineshkumar Paruthi
40192620
MACS *Comp 6651 Winter 2022*
Concordia University
Montreal, Canada
manan.paruthi@gmail.com

2nd Harman Preet Kaur
40198317
MACS *Comp 6651 Winter 2022*
Concordia University
Montreal, Canada
harmanpreetk08@gmail.com

3rd Rajkumar Rokali
40172740
MACS *Comp 6651 Winter 2022*
Concordia University
Montreal, Canada
rokalirajkumar@gmail.com

4th Hanna Hanna
40217905
MSCS *Comp 6651 Winter 2022*
Concordia University
Montreal, Canada
k.k.hannah33@gmail.com

5th Geetika Rathi
40206213
MACS *Comp 6651 Winter 2022*
Concordia University
Montreal, Canada
geetika.rathi94@gmail.com

***Abstract*—The increased spread of the coronavirus has necessitated social distancing and other safety precautions in gathering spaces, particularly in hospitals where both Covid and Non-Covid people will be present. As a result, health centres have decided to orient and label (colour) their hallways/corridors for COVID or non-COVID patients, so that each group of patients move along a path that is labelled specifically for them. In order to implement this, in our project we study the following graph problem: Given a graph G with two sets of node pairs SD1(non-COVID set) and SD2 (COVID set), we are to define the orientation of edges and colouring of the resulting links so that the number of shared edges and alternating edges is minimized, thereby increasing the number of people reaching the destination per unit time as well as minimizing the interactions to reduce the spread of virus. We designed a heuristic algorithm for the orientation of the path with a complexity of O(N^3) , where N is the max(V,E). The Colour of paths is determined by the Coloring algorithm which has a complexity of O(N^2). We also made use of several well-known algorithms like DFS to look for paths, Tarjan to detect bridges and K path algorithm to find paths with minimum overlappings.**

I. Introduction

In this paper, we address a problem that has arisen from the need for social distancing in hospitals to avoid the increased spread of covid. Both covid and non-covid people visit a hospital. To avoid interaction between these two groups of patients, a few rooms (node pairs) are assigned for covid people while the rest are allocated to non-covid people. However, while travelling along the path connecting the source and destination, there will be some links or edges that must be shared by covid and non-covid people and our goal is to minimize this sharing of edges. Given the map of a hospital, we developed an undirected multigraph in which rooms or intersections in the hospital are the nodes and the corridors are the edges. It is possible to have many edges between two nodes in a multi-graph, i.e. multiple lanes in a corridor, assuming the corridor is large enough. The resulting graph, together with two-node pair sets allocated to covid and non-covid people, is utilised as the input for our approach. The output we deliver is a mixed multigraph with directions and colour labelling for the paths depending on whether it is used by covid or non-covid people, or both. Our graph will be having the following types of edges: 1) one-directional edges, used by just one category of patients (covid or non-covid), 2) Alternating edges, which are the edges with no orientation and are used by just one category of patients in both direction, 3) Shared edges which are shared among either just one group of patients or both, but in one direction and 4) Edges which are both alternating and shared. In the case of shared edges when covid people are passing, non-covid people have to wait in some assigned waiting areas and vice versa. Similarly, in the case of alternating edges also called essential edges, links can be used only in one direction at a time, so people who have to travel in the other direction have to wait. As a result, these kinds of edges are reducing the throughput, and the number of people reaching the destination per unit of time. So our goal is to give orientations and define paths such that the number of shared or alternating edges is reduced and thereby increasing the throughput.

II. Problem Statement

As a step toward controlling the spread of the virus in hospitals, the authorities are dividing it into two sections namely, SD1 and SD2, where SD1 denotes the node pairs of Non-Covid sections and SD2 denotes the node pairs of Covid sections. We need to implement a heuristic algorithm that will give an orientation to the edges and also the colouring of the links in such a way that it will reduce the contact between covid and non-covid patients.

While designing the algorithm, We must ensure that a route from the source to the destination exists for every node pair in SD1 and SD2, with a specified label(colour) based on which type of patient can use the path. The occurrence of shared links between paths of the same colour or of different colours needs to be minimized as they correspond to convergent flows, and thus a cause of flow deceleration. If the sharing occurs

between paths of different colours, a waiting area is required at the origin of sharing edge so that one group of people can wait there while other groups use the path. Traversing in both directions on an edge can lead to alternating or essential edges, which are used in a single direction at any given point in time. So there will be a waiting area at the endpoints of these edges so that people who have to travel in the other direction can wait there. This delay will in turn minimize the throughput which is the number of patients reaching their destination per unit time. The orientation of the edges and the colouring of the links should be done in order to maximize the throughput.

III. Literature reviews

A. DFS algorithm

DFS is a visitation algorithm that is implemented using a stack. Given a graph G and source node s V [G], DFS visits all nodes of G that are reachable from s. DFS follows depth-first order during visitation. That means a node's children are visited before its siblings. DFS is used to find paths between given node pairs, that is it checks for connectivity between nodes. In our project, we use DFS in the Colouring algorithm to traverse a path and decide the colour to be assigned to the path depending on if it is shared or alternating, Covid or Non-Covid etc. The complexity for DFS is O(E+V).

```
Input: A graph and a starting vertex root of Graph
Output: Goal state. The parent links trace the shortest path
back to root
1: procedure DFS_iterative(G, v) is
2:   let S be a stack
3:  S.push(iterator of G.adjacentEdges(v))
4:   while S is not empty do
5:     if S.peek().hasNext() then
6:       w = S.peek().next()
7:       if w is not labeled as discovered then
8:         label w as discovered
9:          S.push(iterator of G.adjacentEdges(w))
10:    else
11:       S.pop()
```

Fig. 1. DFS Algorithm

B. Tarjan's algorithm

Tarjan's algorithm contains two interleaved traversals of the graph. First, a depth-first search traverses all edges and constructs a depth-first spanning forest. Second, once the root of a strongly connected component is found, all its descendants that are not elements of previously found components are marked as elements of this component. This second traversal is implemented by using a stack, where each node is stored when entered by the depth-first search. When a root of a component is exited, all nodes down to the root are removed from the stack and they form the component. We are using Tarjan's Algorithm for the detection of bridges as they have the potential to disconnect graphs. Removal of a bridge(s) could lead to a disconnected node(s) as there is only one possible path to that node, so we have made the bridges as alternating edges which can be used in both directions. Therefore, the bridges will give a lower bound on the number of alternating edges in our graph. The Complexity of Tarjan's Algorithm is O(V+E).

Below is an implementation of Tarjan's Algorithm:

```
(1)  procedure VISIT(v);
(2)  begin
(3)    root[v] := v; InComponent[v] := False;
(4)    PUSH(v, stack);
(5)    for each node w such that (v, w) ∈ E do begin
(6)      if w is not already visited then VISIT(w);
(7)      if not InComponent[w]
           then root[v] := MIN(root[v], root[w])
(8)    end;
(9)    if root[v] = v then
(10)     repeat
(11)       w := POP(stack);
(12)       InComponent[w] := True;
(13)     until w = v
(14) end;
(15) begin / * Main program * /
(16)   stack := ∅;
(17)   for each node v ∈ V do
(18)     if v is not already visited then VISIT(v)
(19) end.
```

Fig. 1. Tarjan's algorithm detects the strongly connected components of graph $G = (V, E)$.

C. K-approximation algorithm for MSE (Best k path algorithm)[1]

For a given graph G, with source s and sink t, the algorithm finds k paths between S and T such that the number of edges shared among the paths is minimum. This is an NP Hard problem and the approximation algorithm for MSE is obtained by transforming a network flow problem, called "Minimum Edge-Cost Flow".

In our project, we use the K approximation algorithm to find 2 paths with minimum overlapping between every Source-Destination pairs in SD1 and SD2. This will help us to consider just 2 best paths between Source-Destination instead of finding all possible paths which will require a lot of time. Here we take k=2. From[1], it is known that if $k \leqslant |E|$, we can simply assume complexity of the algorithm is $O(N^2)$.

IV. Proposal

A. Minimum Shared Edges (MSE) Algorithm

In this algorithm, for each node pair in SD1 and SD2, we find the best two minimum-overlapping paths using the concept of the best K-path algorithm, where k=2 and store the resulting paths in a map. For each node pair, we will take the paths from the map and then we will merge them with the next node pair's paths to generate all possible combinations(2 paths of s_i d_i * 2 paths of s_{i+1} d_{i+1} = 4 merged paths) and after that, we will be selecting two merged paths with minimum shared edge count. We are calculating the minimum

shared edge count for the merged path by counting the edges which are used by more than one path. For all combinations of merged paths, separate adjacency lists are maintained. The algorithm searches for the number of edges that are used by more than one path and selects the combination in which the number of sharing is minimum.

B. Variable Definition For MSE

- SD1: It consists of node pairs(source and destination) of non covid sets
- SD2: It consists of node pairs(source and destination) of covid sets
- map: It stores key value(k,v) pairs where k consists of source and destination and value stores the two paths with minimum overlapping edges in them
- lastIndex: It is used to traverse on all node pairs one by one to fetch their respective paths
- finalGraph: it will be the resulting graph with the minimum shared edges for the path of all node pairs in SD
- data: It will store two attributes - number of shared/alternating edges for each combination (sharedEdgeCount) and their corresponding paths(mergedPaths)
- firstMin and secondMin: It stores the two paths with minimum shared/alternating edges
- Result: After considering each new node pair's path in the graph, it will store paths with minimum shared edges so far
- $Paths_{S_iD_i}$: it will have the paths for an SD pair

C. Time Complexity Explanation

- FindPathWithMinSharedEdges(): bestkpathAlgo takes O(N^2) complexity which is inside a for loop which takes O(N) complexity while IntersectingPaths() take O(N^2) complexity so it will be O(N)*O(N^2) + O(N^2) = O(N^3)
- IntersectingPaths(): result and $Paths_{S_iD_i}$ array will have 2 values in each so for loop of $Paths_{S_iD_i}$ and for loop of result, the complexity will be O(1) as its constant 2 for both while AddPathInGraph() takes O(N^2) complexity and its recursive function so it will be O(N) so it will be O(1)*O(1)*O(N^2) + O(N) = O(N^2)
- AddPathInGraph(): for mergedPath it takes O(N) while CountSharedEdges() takes O(N^2) complexity so it will be O(N) + O(N^2) = O(N^2)
- CountSharedEdges(): for loop on all edges takes O(N) while to check if edge is used in more than one path, we need to go through S and D of the edge and check all its connected nodes which takes O(N) so it will be O(N) * O(N) = O(N^2)

Algorithm 1 Find path with minimum shared edges

```
result ←∅
Initialize map        ▷ where key = s_i d_i & value = paths between src to dst
lastIndex ←∅

function FindPathWithMinSharedEdges()                          O(n^3)
    for (s_i, d_i) ∈ SD do                          ▷ where SD ∈ [SD_1, SD_2]
        source ← s_i
        destination ← d_i
        map[s_i d_i] ← bestKPathAlgo(s_i, d_i, k = 2)        ▷ Finds two least
overlapping paths
    end for

    Paths_{S_iD_i} ← map[SD_lastIndex]
    lastIndex ← lastIndex − 1
    result ← result + Paths_{S_iD_i}
    finalGraph ← IntersectingPaths()
    Display(finalGraph)
end function

function IntersectingPaths()                                   O(n^2)
    if lastIndex = -1 then
        return result[0]                                ▷ Stores the best path
    end if
    Paths_{S_iD_i} ← map[SD_lastIndex]          ▷ Considering next S_iD_i pair
 for paths ∈ result do
    for currPath ∈ Paths_{S_iD_i} do
        data ← addPathInGraph(paths, currPath)
        sharedEdgeCount ← data[0]
        mergedPaths ← data[1]
        if sharedEdgeCount > firstMin then :
            firstMin ← mergedPaths
            secondMin ← first
        else
            if sharedEdgeCount < secondMin then:
                secondMin ← mergedPaths
            end if
        end if
    end for
 end for
 result ← [firstMin, SecondMin]
 IntersectingPaths()
end function

function addPathInGraph(paths, currPath)                       O(n^2)
    mergedPaths ← paths + currPath
    sharedEdgeCount ← CountSharededges(mergedPath)
    return sharedEdgeCount, mergedPaths
end function

function countSharededges(mergedPaths) ▷ mergedPath ← adjacencylist   O(n^2)
    count ← 0
    for edge in mergedPath do
        count += 1 if (edge used in more than one path) else 0
    end for
    return count
end function
```

D. Colouring Algorithm

The algorithm assigns three colors to edges: Red, Green and Blue.

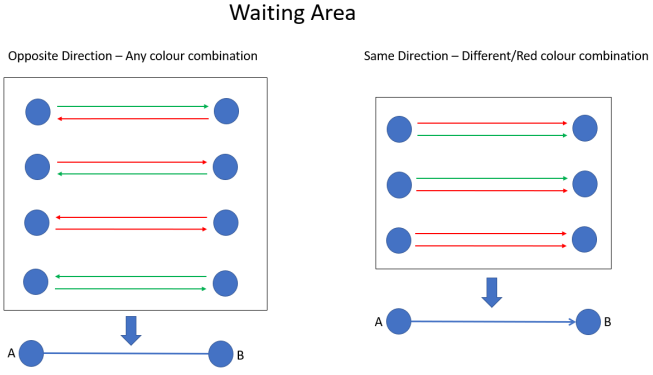
Case RED: Path connecting covid node pairs

Case GREEN: Path connecting non-covid node pairs

Case BLUE:

1) If an edge is shared by 2 covid paths or by covid and non covid paths in the same direction, blue directed edge is used.

2) If an edge is used in both directions(alternating edge) by paths of any color, blue undirected edge is used.



E. Types of sharing

An edge may be shared by different paths of the same color or of different colors. In our algorithm, we are giving the same priority to all kinds of sharing and are trying to minimize them as much as possible. But in the real case scenario, one type of sharing should be discouraged more as compared to the other. We discuss it in the following cases:

Case 1: Sharing between Green paths: This edge is shared between Non Covid paths, so only proper social distancing is required. No need for waiting areas.

Case 2: Sharing between Red paths: This edge is shared between Covid paths. Usage of waiting areas is required as when one covid patient is using the path, others need to wait. But the number of covid node pairs is limited, therefore we can assume that there will be a limitation on the number of covid people visiting a hospital, thereby less impact on the throughput even if waiting is required.

Case 3: Sharing between Red and Green path: This edge is shared between Covid and Non Covid paths. This is the sharing that needs to be discouraged the most. In this case there should be waiting areas where one group of patients wait while the other group is using the edge. This sharing is going to have a higher impact on throughput as compared to previous case as a large number of non covid people have to wait while covid people are moving.

The order in which the 3 cases of sharing are to be discouraged is as follows: $Case3 > Case2 > Case1$.

F. Variable Definition For Colouring Algorithm

- SD1: It consists of node pairs(source and destination) of non covid sets
- SD2: It consists of node pairs(source and destination) of covid sets
- color: It stores the color to be assigned to an edge between two nodes
- currNode and prevNode: While DFS traversal from Si to Di, the currNode is the current node which is being traversed and prevNode is the the last traversed node
- ANYCOLOR: either red or green color
- G: Green
- R: Red
- B: Blue

G. Time Complexity Explanation

- ColoringAlgo(): for loop on all SD pairs takes O(N) while DFS() also takes O(N) complexity so it will be O(N) * O(N) = O(N^2)
- DFS(): EdgeExists() takes O(N) time complexity while MakeColoredEdge() takes O(1) so it will be O(N) + O(1) = O(N)
- EdgeExists(): we are checking for all connected edges of currNode so it takes O(N) complexity
- MakeColoredEdge(): adding value into map takes O(1) time complexity

Algorithm 2 Coloring Algorithm

O(n^2)

```
function ColoringAlgo()
    for (s_i, d_i) ∈ SD do                          ▷ where SD ∈ [SD_1, SD_2]
        source ← s_i
        destination ← d_i
        if (s_i, d_i) ∈ SD_1 then
            color ← GREEN
        else
            color ← RED
        end if
        DFS(s_i, d_i, color, s_i)
    end for
end function
```

O(n)

```
function DFS(s_i, d_i, color, prevNode)
    if currNode = d_i then
        break
    else
        if EdgeExists(currNode, prevNode, ANYCOLOR) then
            MakeColoredEdge(prevNode, currNode,UNDIRECTED,B )
        else if EdgeExists(prevNode,currNode, G) AND color=R then
            MakeColoredEdge(prevNode, currNode, DIRECTED, B)
        else if EdgeExists(prevNode,currNode, R) AND color=G then
            MakeColoredEdge(prevNode, currNode, DIRECTED, B)
        else if color = R then
            MakeColoredEdge(prevNode, currNode, DIRECTED, R)
        else if color = G then
            MakeColoredEdge(prevNode, currNode, DIRECTED, G)
        end if
    end if
    DFS(s_i, d_i, color, currNode)
end function
```

O(n)

```
function EdgeExists(currNode, prevNode, color)
    for edge ∈ map[currNode] do
        if edge[node] = prevNode AND edge[color] = color then
            return true
        end if
    end for
    return false
end function
```

O(1)

```
function MakeColoredEdge(s_i, d_i, direction, color)
    if direction = DIRECTED then
        map[s_i].add((d_i, color))
    else
        map[s_i].add((d_i, color))
        map[d_i].add((s_i, color))
    end if
end function
```

1

H. SAMPLE EVALUATION OF ALGORITHM

Input from the Project Part1

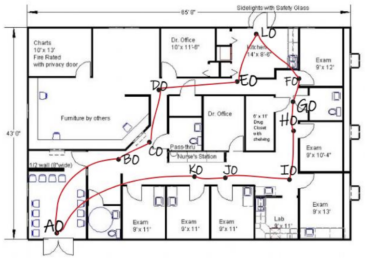


Fig. 2. Input

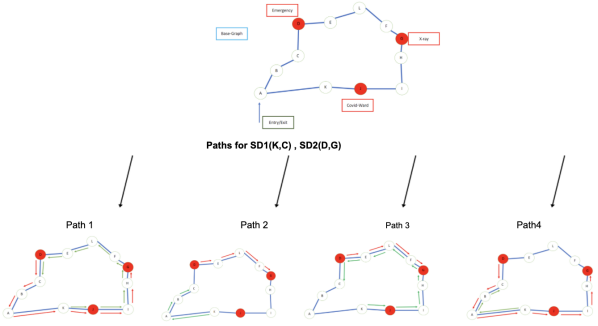


Fig. 3. Paths for SD1(K,C) and SD2(D,G)

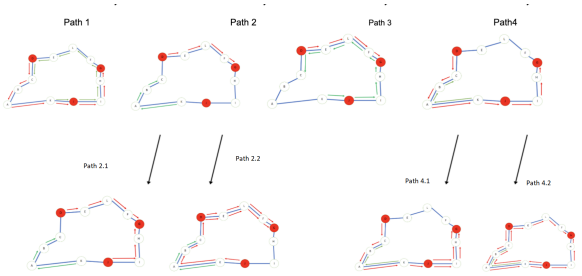


Fig. 4. selecting path 2 and 4 as they have minimum sharing edges and adding path SD2(J,G)

Here,we are selecting two graphs with min shared edge count.

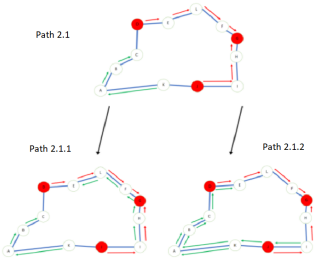


Fig. 5. Picking path 2.1 since it has minimum shared edges and adding the possible paths of (I,E)

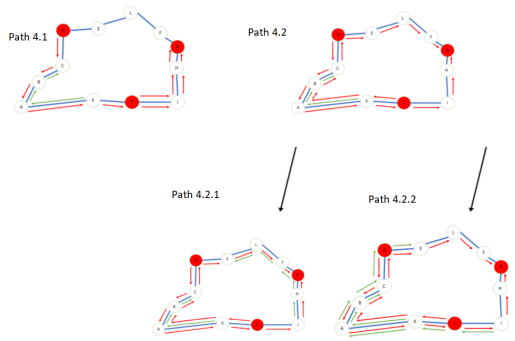


Fig. 6. Picking path 4.2 since it has minimum shared edges and adding the possible paths combinations of node pair (I,E)

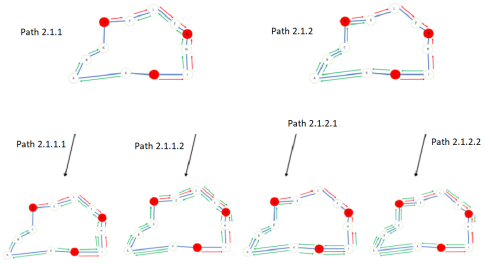


Fig. 7. Picking path 2.1.1 and 2.1.2 since they have minimum shared edges and adding the possible paths combinations of node pair (K,H)

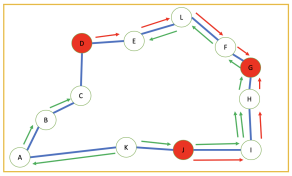


Fig. 8. As we have now considered all the node pairs of SD1 and SD2, at the end we are selecting the best graph with minimum shared edge count

I. Why Heuristic ?

An algorithm is heuristic if it comes rapidly to a solution expected to be close to the best possible answer or the best solution we can get in a reasonable amount of time.
In our algorithm, for each node pair of SD we are considering the 2 best paths using best-K path algorithm instead of all possible paths so that we do not need to go through/calculate for all combinations and we can come rapidly to a solution. Thus it is heuristic and not an optimal solution / approach.

V. Conclusion

We attempted to solve the problem of minimizing the spread of covid-19 infection in a hospital. For this reason, we provide orientation and coloring of the hospital's hallways/corridors for covid and non-covid patients. Our goal was to provide orientation in which the routes of covid(SD2 node pairs) and non-covid(SD1 node pairs) should have minimum overlapping. In this paper, we developed a heuristic approach with complexity O(N^3) for finding paths with the minimum shared edges (Algorithm 1) and an optimal coloring algorithm (Algorithm 2) with complexity O(N^2). In Algorithm 1, we utilize the best-k path algorithm and shared edge count to define orientation with minimum shared and alternating edges in the graph.In Algorithm 2, we are utilizing DFS to go from source to destination, and we are coloring the edges based on the shared edges, alternating edges and also taking into account the directions of the edges.

By decreasing shared and alternating edges, our approach produces a graph which has the best possible path with least amount of contact between covid and non-covid patients and increased throughput.For all node pairs, our algorithm does not investigate all possible paths, in order to obtain faster results and reduce time complexity. An alternate approach is to find all possible paths between node pairs using algorithms like DFS/BFS and then look for minimum sharing, but this would increase time complexity drastically. Thus we proceeded with heuristic approach.

References

[1] Omran, Masoud Sack, Jörg-Rüdiger Zarrabi-Zadeh, Hamid. (2011). Finding Paths with Minimum Shared Edges. Journal of Combinatorial Optimization. 26. 567-578. 10.1007/978-3-642-22685-4_49.
[2] R. Tarjan, "Depth-first search and linear graph algorithms," 12th Annual Symposium on Switching and Automata Theory (swat 1971), 1971, pp. 114-121, doi: 10.1109/SWAT.1971.10.
[3] https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm
[4] E. M. Arkin R. Hassin, 'A Note on Orientations of Mixed Graphs'. 2004.
[5] https://doi.org/10.1016%2F0020-0190%2894%2990047-7 .