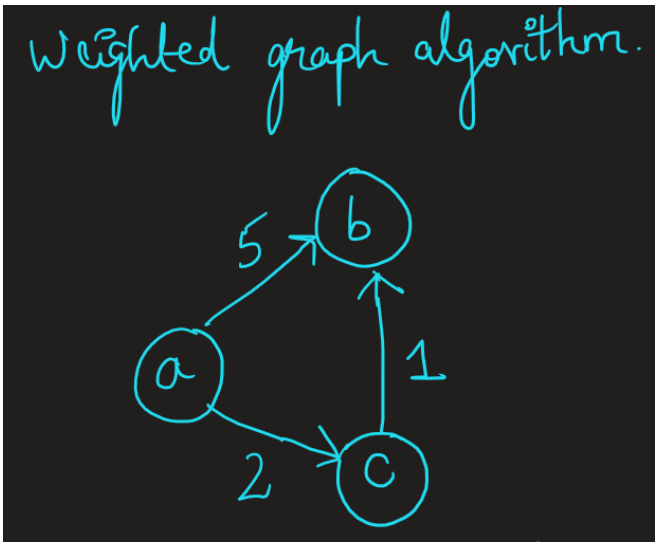
Task 1 – Path Planning Algorithms

Swaminathan S K

22CS30057

Djikstra's Algorithm

Djikstra's Algorithm can be used for minimizing the cost (and find the shortest path) in a weighted graph or a grid.



The Algorithm

- Djikstra's Algorithm considers the minimum cost path for every node at a particular time. For example, consider that a node X has been reached already and the optimum path/ least cost path is stored till X.
- Now, from X we try to look at all its neighbours (in the case of a graph, these will be the children nodes). If the neighbour is not already visited, then set the minimum cost path to be from X to that neighbour.
- If the neighbour has been visited, then we see if the path from X to the neighbour (say Y) costs lesser than the already stored path cost. If so, then we update the newer path to be from X to Y. Else, we leave it be the same. This ensures an optimal path for each of the vertex/node/cells from the start.
- We stop the algorithm once we reach the end point (specified by the user)

Implementing the algorithm

- For implementing the algorithm, store the information about the graph as a linked list
- Each node will have a linked list associated with it
- Each element in the linked list represents nodes connected to that node

- Along with the pointer to the next element, each element contains the node number and the weight/ cost to reach that node.

Pseudocode

```
1  function Dijkstra(Graph, source):
 2
 3      for each vertex v in Graph.Vertices:
 4          dist[v] ← INFINITY
 5          prev[v] ← UNDEFINED
 6          add v to Q
 7      dist[source] ← 0
 8
 9      while Q is not empty:
10          u ← vertex in Q with min dist[u]
11          remove u from Q
12
13          for each neighbor v of u still in Q:
14              alt ← dist[u] + Graph.Edges(u, v)
15              if alt < dist[v]:
16                  dist[v] ← alt
17                  prev[v] ← u
18
19      return dist[], prev[]
```

A* Algorithm

A* maintains a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. A* selects the path that minimizes –

$$f(n) = g(n) + h(n)$$

where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal.

Heuristic function

Heuristic function is a approximation or guess that we make about how much cost it would take for the path starting from the current point to the goal. We can do this by several methods. Some of them are –

- Euclidean Distance Heuristic - H(xn , yn) = sqrt((xn - xg)^2 + (yn - yg)^2)
- Manhattan Distance Heuristic - H(xn , yn) = |xn - xg | + | yn - yg |

A* Algorithm is an extended case of Djikstra's algorithm, whereby not only the history g(n) is recorded, but also a guess to the future is included. (In Djikstra's algorithm, h(n) = 0 for all n).

Implementation will be similar to Djikstra's algorithm, with slight variations in calculating the f(n).

Pseudo code

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path

// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from node n.
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather than a hash-set.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from the start
    // to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current best guess as to
    // how cheap a path could be from start to finish if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        // This operation can occur in O(Log(N)) time if openSet is a min-heap or a priority queue
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from current to neighbor
```

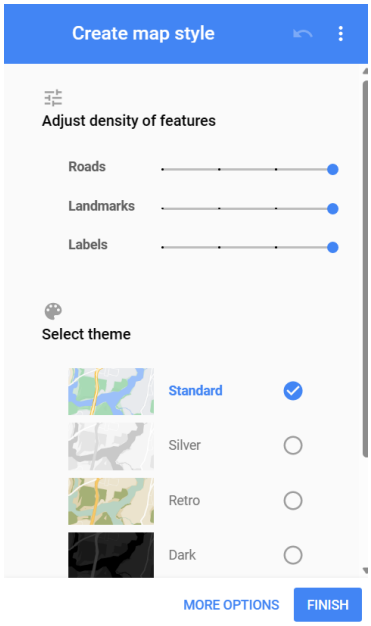
```
            // tentative_gScore is the distance from start to the neighbor through current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one. Record it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := tentative_gScore + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```

Map Image Collection and Filtering

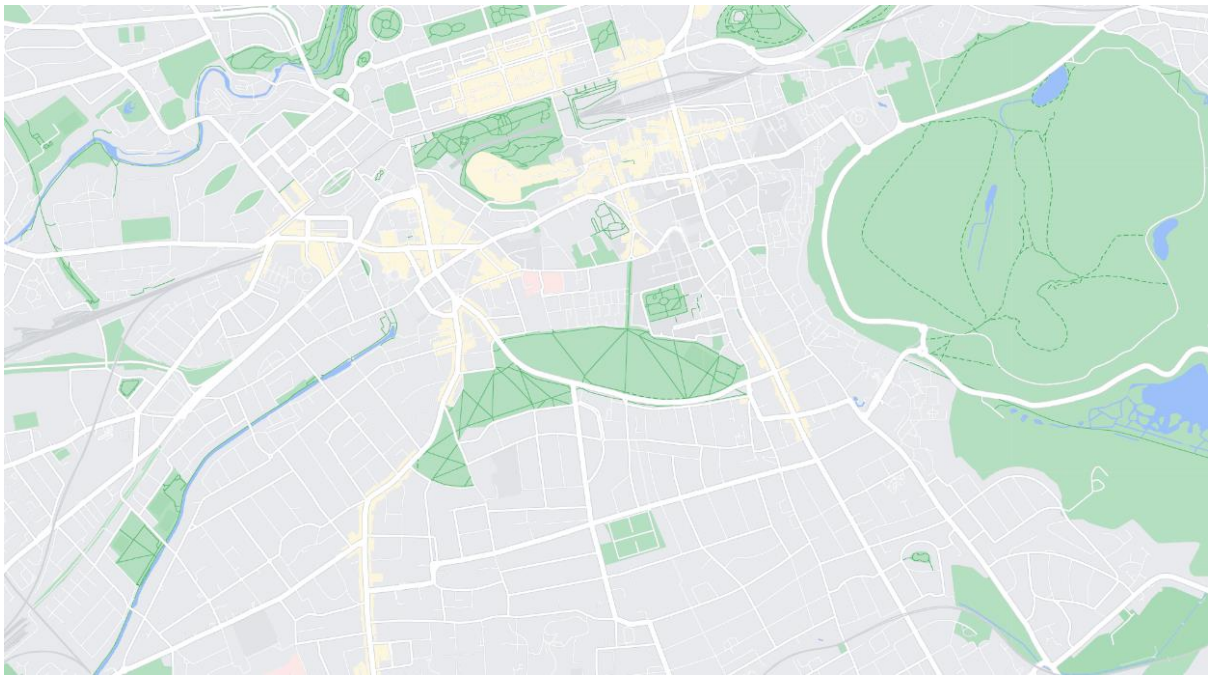
Initial Scouting:

This was the place we scouted for executing Djikstra's and A* algorithms. The first hurdle to cross was to remove the labels and icons. we searched up the internet for software that allows for us to do this.

We used the Legacy JSON styling wizard, [1] (mapstyle.google.com) to accomplish this task.



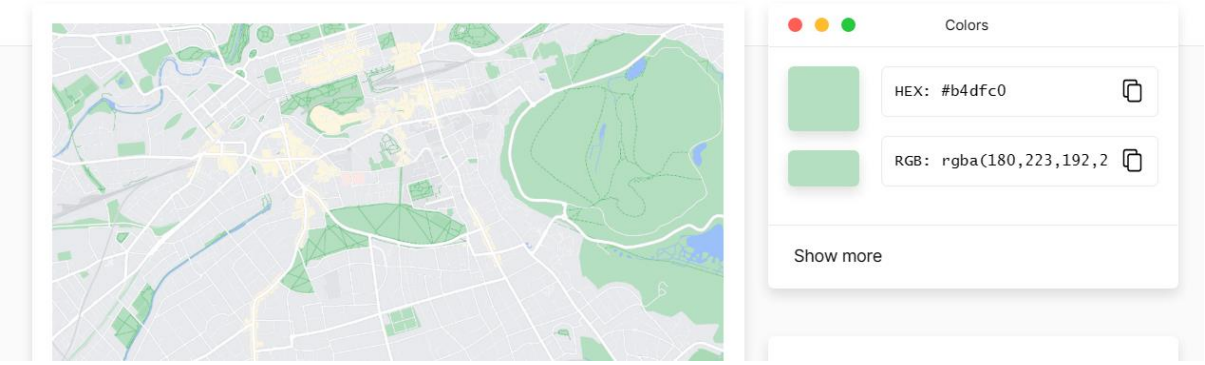
By appropriately adjusting the visibility parameters, we got the following image-



Now, onto the task of converting this image into an image whereby the roads are highlighted in white and the rest in black. We first applied the brute force method of converting only the (255, 255, 255) pixels into white pixels and the rest of the image to black. This, however, gave broken roads and incomplete paths with bad resolution.

Image filtering using colour picker software:

To get a better accurate image and resolve issues such as deviations in the roads' colours from (255, 255, 255) to some other greyish colour close to white, we decided to omit all the parts of the image that is uniformly coloured with some other colour apart from pure white. This was hard-coded.



We used the following online software to pick the colour [2] (imagecolorpicker.com). It must also be mentioned that the image was blurred using Gaussian Blur before operating on them.

Code for the Map Filter

```
# Importing the required libraries
import cv2 as cv
import numpy as np

# Reading the Image, and Blurring it to remove noise
map_default = cv.imread("image.png")
map_default = cv.GaussianBlur(map_default, (5, 5), 0)

# I used a color picker software to get these colors that indicate patches of colors that do not contribute to the pathways
colors = [[227, 208, 233], [249,192,156], [224,247,254], [192,223,180], [237,234,232], [231,227,225], [181,218,168], [230,232,252]]
height, width = map_default.shape[:2]

# Final touches by converting the corresponding colors to black and also other miscellaneous colors with any value less than 230 to black
for i in range(height):
    for j in range(width):
        for color in colors:
            if (map_default[i][j] == color).all() or map_default[i][j][0] <= 230 or map_default[i][j][1] <= 230 or map_default[i][j][2] <= 230:
                map_default[i][j] = [0, 0, 0]

# Storing the image into another file
cv.imwrite("image_gray.png", map_default)

#Displaying the image
cv.imshow("Final Map", map_default)
cv.waitKey(0)
cv.destroyAllWindows()
```

Challenges in Coding the Algorithms

Djikstra's Algorithm

Our initial approach to implementing this algorithm was similar to our implementation of the BFS algorithm in Winter School 2023. This works fine since, the weights of each traversal is only 1. However, since we had to take this approach and proceed into A* algorithm whereby we add a heuristic term. We had to recode the algorithm to fit the original description of Djikstra's algorithm with the open list, closed list and corresponding parents, cost dictionaries.

A* Algorithm

We modified the node picking part of the Djikstra's algorithm to include the heuristic function. However, the heuristic function we used initially was $|x - x0| + |y - y0|$ which overestimated the cost from node to the final node, since even diagonal traversals are allowed. This was found after

counting the path length and comparing for both Djikstra and A*. So the best fit underestimated heuristic for this would be the Euclidean Distance , i.e.

$$\sqrt{(x - x0)^2 + (y - y0)^2}$$

After making this change, both algorithms were complete and optimal, with A* performing better in terms of time complexity.

Results

The image after filtering

Djikstra's Algorithm

A* Algorithm

The thin yellow line is the optimal path. And the blue coloured paths are the ones reached by the algorithm.

References

[1] Color Picker online | HEX Color Picker | HTML Color Picker (imagecolorpicker.com)

[2]Styling Wizard: Google Maps APIs (mapstyle.withgoogle.com)

[3]Dijkstra's algorithm - Wikipedia

[4]A* search algorithm - Wikipedia

[5]A* Search | Brilliant Math & Science Wiki