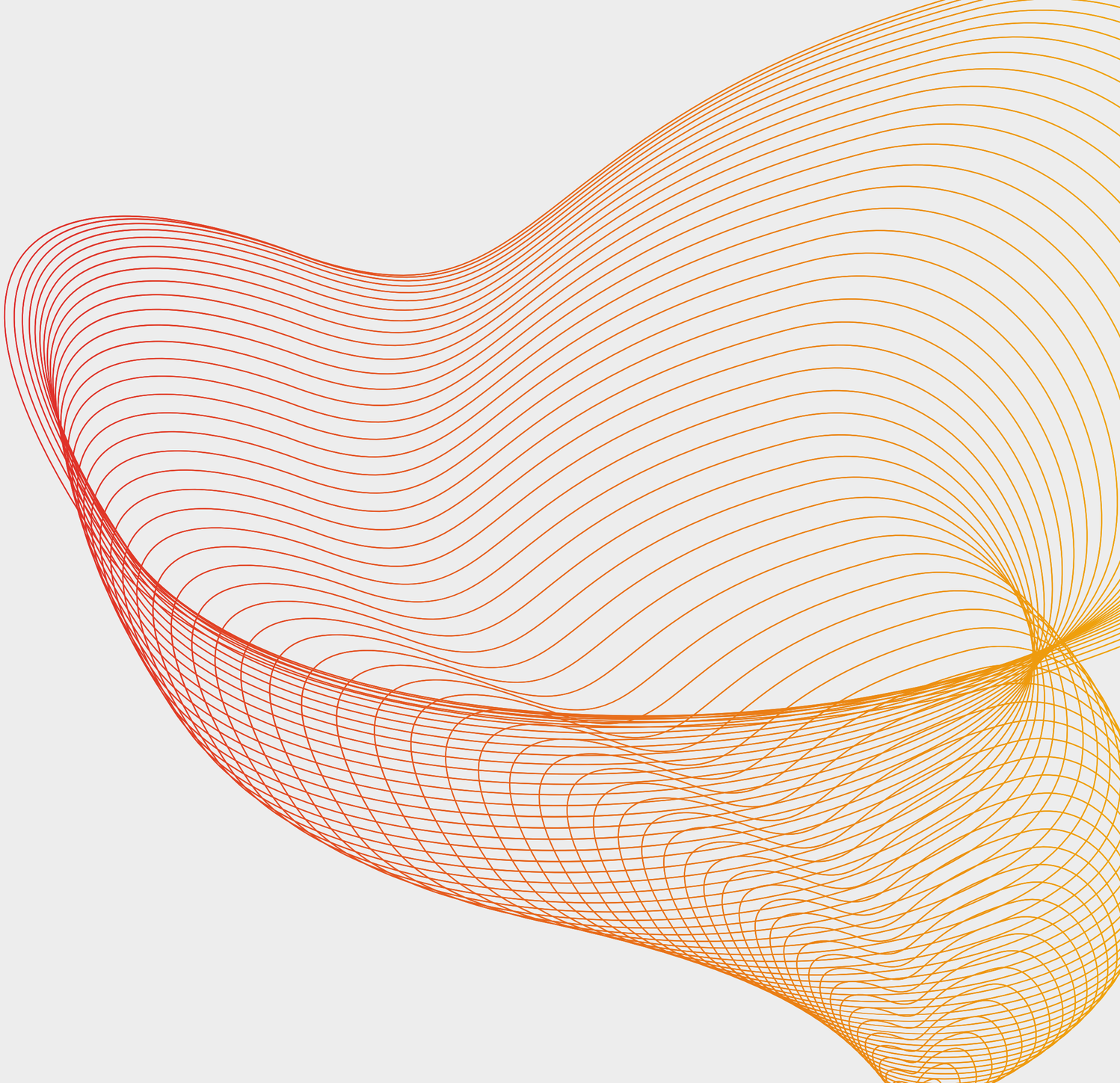


Google Maps!

but lite

Created by
Arjun Doshi J053
Aryan Shah J047

Table of Content

About

The idea is to create a version of Google Maps using search algorithms to find the optimal path from source to destination.

Approach

Comparing 4 algorithms to see optimal paths

Algorithms

Depth First Search

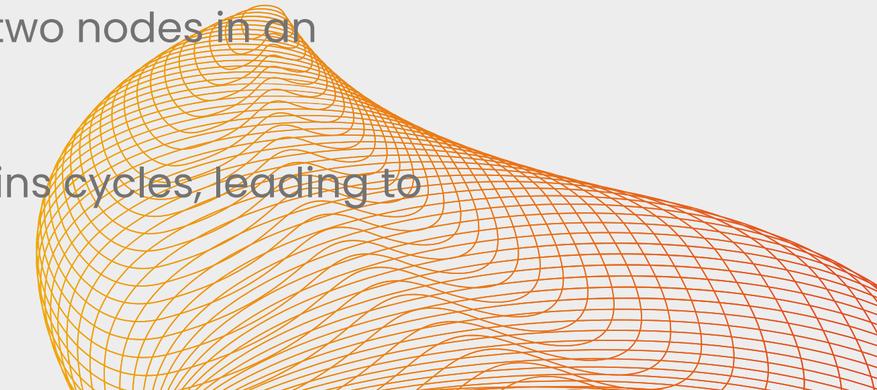
Depth first search uses a stack data structure because it needs to visit all the adjacent nodes of a node before moving on to the next node. Stack allows us to backtrack easily to the previous node to explore its other adjacent nodes.

Advantages:

1. Depth first search can use less memory compared to breadth first search because it explores the graph depthwise and doesn't need to store all the nodes at the same level.
2. Depth first search can be used to generate mazes or solve puzzles, where exploring all paths is important.

Disadvantages:

1. Depth first search may not always find the shortest path between two nodes in an unweighted graph.
2. Depth first search may go down an infinite path if the graph contains cycles, leading to an infinite loop.



Depth First Search

```
def dfs(self, source, destination):
    # A 'stack' data structure is used as a frontier in DFS. It is a LIFO system.
    # last-in first-out similar to a pile of books.

    # store the source element in the stack
    stack = [source]
    self.visited[source.name] = 1 # mark as visited

    # Run a loop unitl all the nodes are explored or until the path is found
    while stack:
        # Remove the element at the top of the stack
        element = stack.pop()
        # print("Currently exploring: ",element.name)

         # Check if the element is destination, if yes the goal has been found break the loop
        if element.name == destination.name:
            # add the element being explored to the path
            self.path.append(element)
            print("Goal has been found")
            break

        # If not then:
        # Loops over adjacent elements
        for neighbour, distance in self.adj_list[element.name]:
            # Checks if visited before
            if self.visited[neighbour.name] != 1:

                stack.append(neighbour)
                self.visited[neighbour.name] = 1 # Marks that neighbour as visited
                self.parent[neighbour.name] = element.name
```

Breadth First Search

Breadth first search uses a queue data structure (first-in, first-out) because it needs to visit all the nodes at the same level before moving to the next level. Queue ensures that nodes are visited in the order they are added to the queue.

Advantages:

1. Breadth first search always finds the shortest path between two nodes in an unweighted graph.
2. Breadth first search is guaranteed to find a solution if it exists in a finite graph.

Disadvantages:

1. Breadth first search may use a lot of memory if the graph is large and complex.
2. Breadth first search is not optimal for solving problems in weighted graphs as it doesn't consider the weights of edges while finding the shortest path.

Breadth First Search

```
def bfs(self, source, destination):
    queue = [source]
    self.visited[source.name] = 1

    while queue:
        element = queue.pop(0)
        # print("Currently exploring: ",element.name)

        if element.name == destination.name:
            self.path.append(element)
            print("Goal has been found")
            break

        for neighbour, distance in self.adj_list[element.name]:

            if self.visited[neighbour.name] != 1:

                queue.append(neighbour)
                self.visited[neighbour.name] = 1 # Marks that neighbour as visited
                self.parent[neighbour.name] = element.name
```

Greedy Best First Search

Greedy best first search uses a priority queue data structure to decide which node to visit next. It evaluates nodes based on the heuristic function, which is an estimate of the distance to the goal node. The node with the lowest heuristic value is chosen first, making it a greedy algorithm. The priority queue allows the algorithm to visit the nodes with the most promising heuristic values first. The heuristic function is domain-specific and aims to estimate the remaining distance or cost to reach the goal.

Advantages:

1. Greedy best first search can be very efficient in finding a solution as it always moves towards the node that appears to be closest to the goal.
2. Greedy best first search can be used to solve problems with large state spaces where an exhaustive search is not feasible.

Disadvantages:

1. Greedy best first search may not always find the optimal solution as it doesn't consider the cost of reaching the current node from the start.
2. Greedy best first search can get stuck in local optima or infinite loops if the heuristic function is not well-designed.

Greedy Best First Search

```
def best_first_searchHaversine(self, source, destination):
    # Priority Queue is a variant of Queue that retrieves open entries in priority order (lowest first).
    # Entries are typically tuples of the form:  (priority number, data).
    pq = PriorityQueue()
    pq.put((0,source))
    self.visited[source.name] = 1

    while pq.empty() == False:
        element = pq.get()[1]
        # print("Currently Exploring:", element.name)

        if element.name == destination.name:
            print("Goal has been found")
            break

        for neighbour, distance in self.adj_list[element.name]:
            if self.visited[neighbour.name] != 1:

                h_dist = Graph._find_haversineDistance(neighbour,destination)

                pq.put((h_dist, neighbour))
                self.visited[neighbour.name] = 1 # Marks that neighbour as visited
                self.parent[neighbour.name] = element.name
```

A Star Search

A* search uses a priority queue data structure to decide which node to visit next. It evaluates nodes based on the sum of the actual cost from the start node to the current node (g-value) and the estimated cost from the current node to the goal node (h-value). The sum of these values is called the f-value. The priority queue allows the algorithm to visit nodes with the lowest f-value first. The heuristic function is used to estimate the h-value, which is the cost from the current node to the goal node. The f-value helps the algorithm to balance between finding the shortest path and exploring promising nodes

Advantages:

1. A* search is guaranteed to find the shortest path between the start and goal nodes in both weighted and unweighted graphs if the heuristic function is admissible and consistent.
2. A* search is a complete algorithm, meaning that it can always find a solution if it exists.

Disadvantages:

1. A* search can be computationally expensive if the graph is large and the heuristic function is not well-designed.
2. A* search requires a lot of memory as it needs to store all the visited nodes and their f-values. This can be a problem in large graphs.

A Star Search

```
def a_star_searchHaversine(self, source, destination):

    pq = PriorityQueue()
    pq.put((0,source))
    self.visited[source.name] = 1

    while pq.empty() == False:
        element = pq.get()[1]
        # print("Currently Exploring:", element.name)

        if element.name == destination.name:
            print("Goal has been found")
            break

        for neighbour, distance in self.adj_list[element.name]:
            if self.visited[neighbour.name] != 1:

                self.g_dist[neighbour.name] = self.g_dist[element.name] + distance
                h_dist = Graph._find_haversineDistance(neighbour,destination)
                f_dist = self.g_dist[neighbour.name] + h_dist

                pq.put((f_dist, neighbour))
                self.visited[neighbour.name] = 1
                self.parent[neighbour.name] = element.name
```

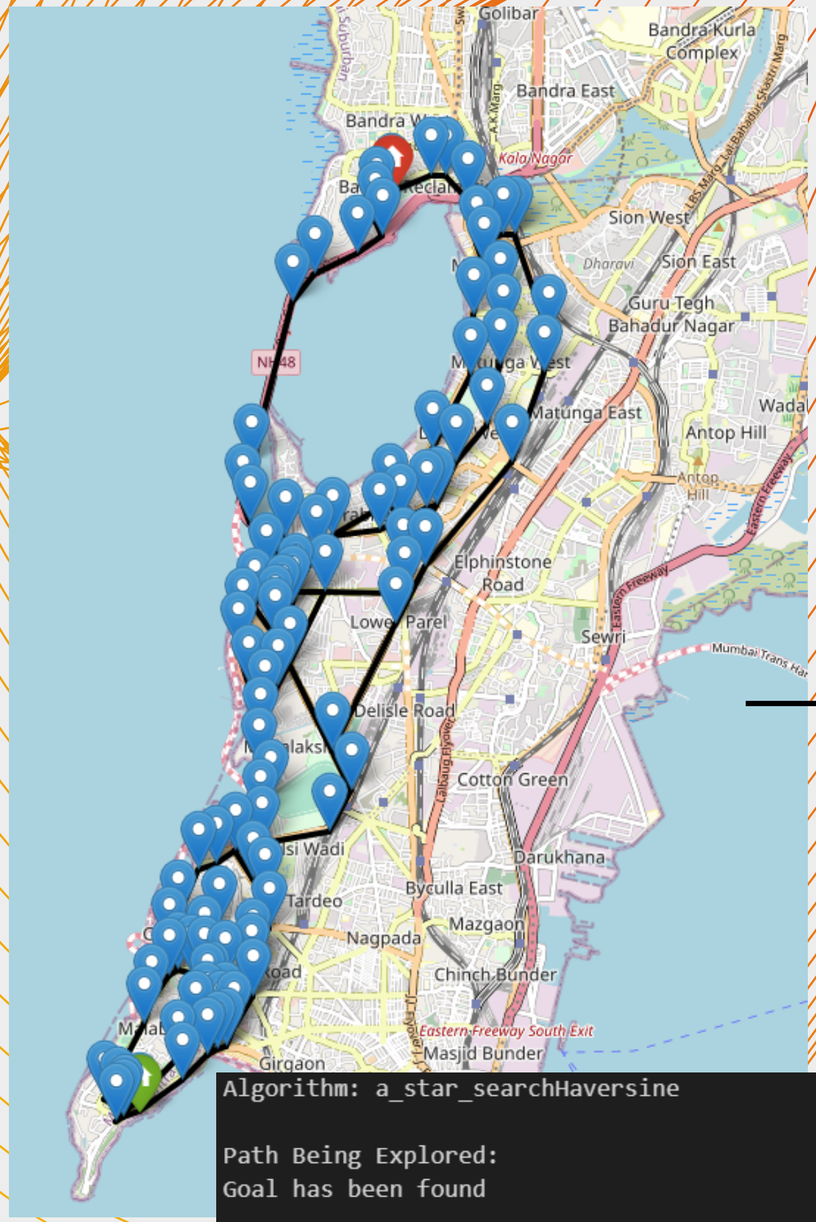
Dictionary of Nodes

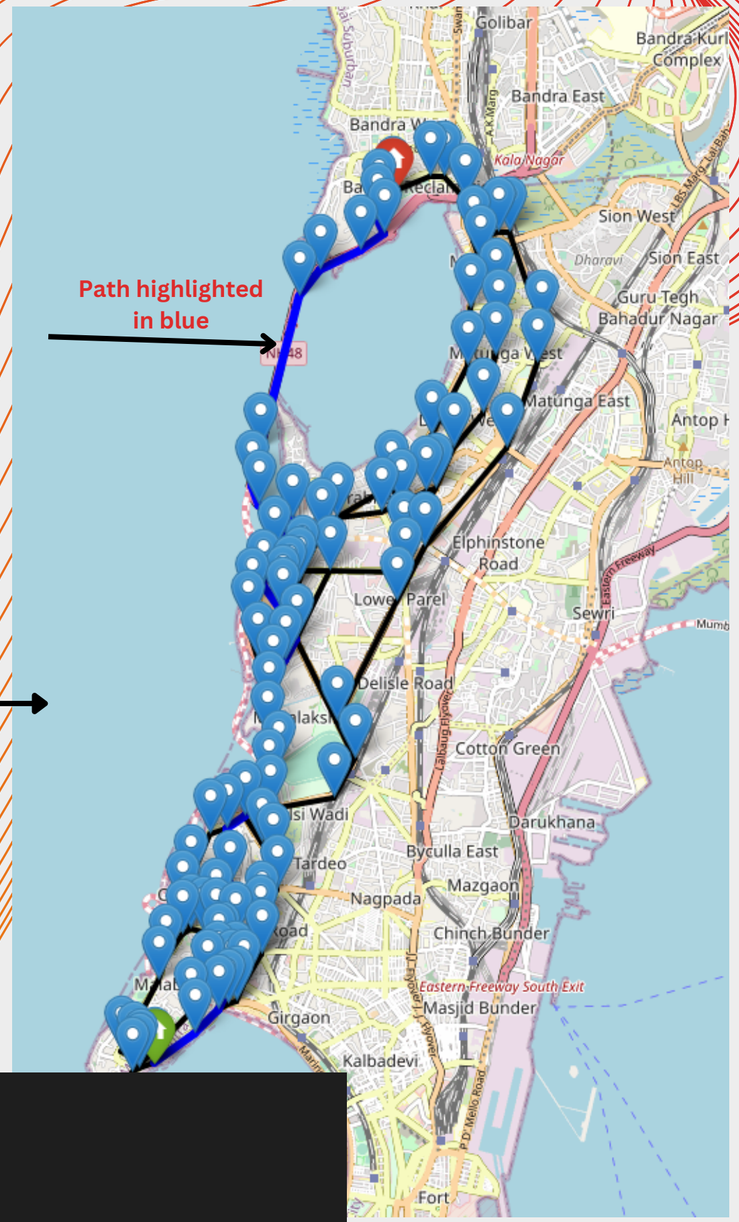
```
def coords():
    coords = {'A':[18.948940, 72.800114], 'B':[18.952430, 72.805009], 'C':[18.955162, 72.808014], 'D':[18.958068,72.809458], 'E':[18.959034,72.809951],
        'F':[18.961364,72.808486], 'G':[18.964083,72.807628], 'H':[18.966315,72.807671], 'I':[18.969602,72.809387], 'J':[18.976258,72.808958],
        'K':[18.977711,72.811188], 'L':[18.981412,72.814149],'M':[18.983604,72.815265], 'N':[18.987134,72.813978], 'O':[18.990543,72.814192],
        'P':[18.993574,72.814582], 'Q':[18.996171,72.812780], 'R': [18.999912,72.811566], 'S':[19.002509,72.812090], 'T':[19.004646,72.813427],
        'U':[19.008640,72.814731], 'V':[19.012373,72.816962], 'W':[19.013972,72.812956], 'X':[19.016260,72.812123], 'Y':[19.020520,72.813024],
        'Z':[19.038282,72.817905], 'AA':[19.041284,72.820436], 'BA':[19.043604,72.825452], 'CA':[19.045503,72.828385], 'DA':[19.047393,72.827545],
        'EA':[19.049016,72.827614], 'FA':[19.050496,72.829458], 'GA':[18.948327,72.798081], 'HA':[18.947827,72.797239], 'IA':[18.948852,72.798354],
        'JA':[18.954918,72.804458], 'KA':[18.958056,72.806611], 'LA':[18.959034,72.807726], 'MA':[18.961124,72.808022], 'NA':[18.949619,72.797523],
        'OA':[18.950350,72.795913], 'PA':[18.958506,72.800640], 'QA':[18.960958,72.801363], 'RA':[18.964002,72.803507], 'SA':[18.964546,72.804477],
        'TA':[18.964180,72.806021], 'UA':[18.967265,72.803490], 'VA':[18.970252,72.804443], 'WA':[18.975536,72.806844], 'XA':[18.955783,72.809179],
        'YA':[18.956148,72.810208], 'ZA':[18.958035,72.811014], 'AB':[18.961615,72.813211], 'BB':[18.961615,72.813211], 'CB':[18.963588,72.809934],
        'DB':[18.964387,72.813030], 'EB':[18.964387,72.813030], 'FB':[18.966035,72.813215], 'GB':[18.969055,72.815184], 'HB':[18.972813,72.814587],
        'IB':[18.974611,72.813185], 'JB':[18.978486,72.814219], 'KB':[18.979781,72.822164], 'LB':[18.984302,72.824746], 'MB':[18.988814,72.822576],
        'NB':[18.998399,72.817497], 'OB':[18.995948,72.816116], 'PB':[19.001353,72.815773], 'QB':[19.002789,72.829919], 'RB':[19.009119,72.833324],
        'SB':[19.020601,72.843490], 'TB':[19.030412,72.847170], 'UB':[19.034858,72.847822], 'VB':[19.045795,72.843867], 'WB':[19.045714,72.842495],
        'XB':[19.044740,72.839278], 'YB':[19.049681,72.838308], 'ZB':[19.052156,72.835735], 'AC':[19.052293,72.834114], 'BC':[19.006291,72.821608],
        'CC':[19.006311,72.818198], 'DC':[19.002846,72.815891], 'EC':[19.004084,72.816727], 'FC':[19.004928,72.817894], 'GC':[19.010673,72.820639],
        'HC':[19.012536,72.822561], 'IC':[19.016268,72.829252], 'JC':[19.014077,72.830459], 'KC':[19.015794,72.834676], 'LC':[19.022021,72.834121],
        'MC':[19.030119,72.838650], 'NC':[19.036805,72.838976], 'OC':[19.042485,72.840177], 'PC':[19.015514,72.833595], 'RC':[19.020642,72.836890],
        'SC':[19.024602,72.840579], 'TC':[19.031288,72.842037], 'UC':[19.035020,72.842380], 'VC':[19.038525,72.842045],
        'XC':[19.006117,72.830874], 'YC':[19.009192,72.830797], 'ZC':[19.013043,72.828107]
}
    return coords
```

RESULTS

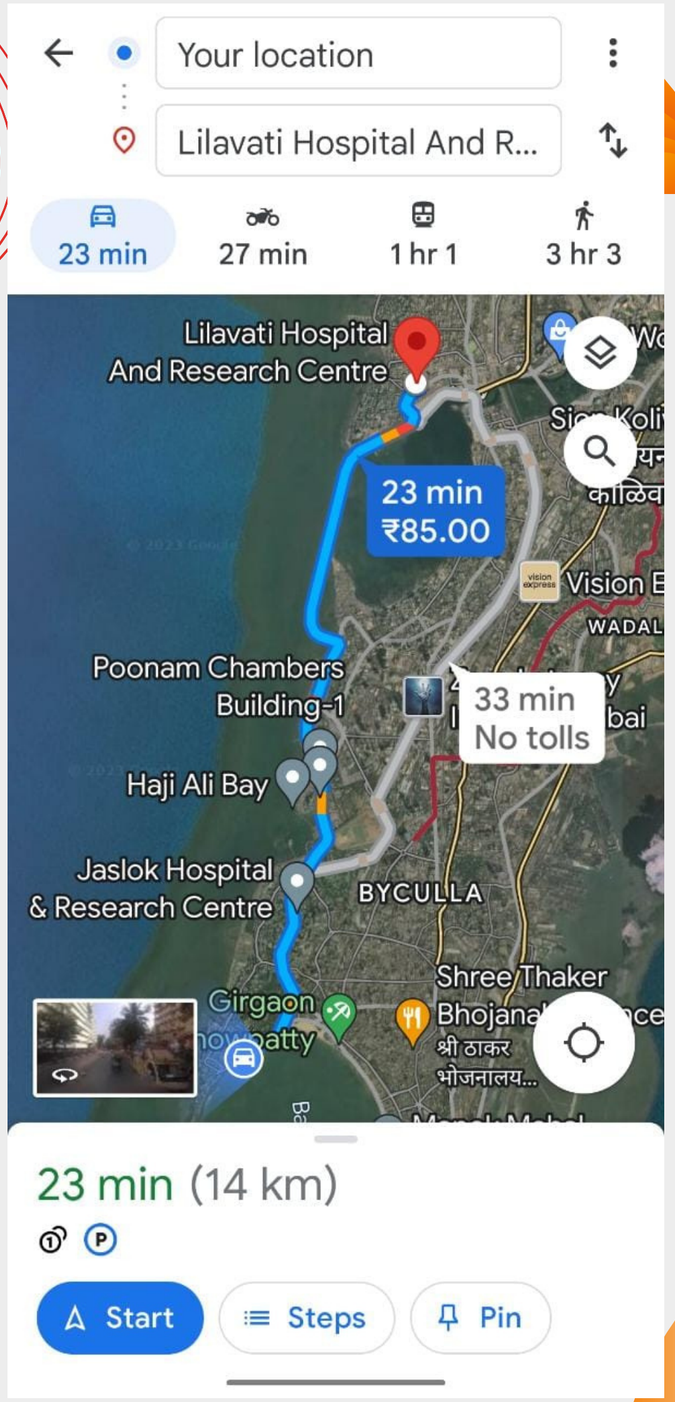
```
Algorithm: a_star_searchHaversine

Path Being Explored:
Goal has been found

Path :
A B C D E F G H I J K L M N O P OB NB PB T U V W X Y Z AA BA CA DA EA FA

Distance:
13.46 Kms / 13460.0 m
----------------------------------------------------------------

Algorithm: best_first_searchHaversine

Path Being Explored:
Goal has been found

Path :
A B C D E F G H I J K L M N O P OB NB BC XC YC KC RC SC TC UC VC OC XB YB ZB AC FA

Distance:
14.67 Kms / 14670.0 m
----------------------------------------------------------------
```

```
Algorithm: bfs

Path Being Explored:
Goal has been found

Path :
A B C D E F G H I J K JB KB LB MB QB RB SB TB UB VB WB XB YB ZB AC FA

Distance:
15.43 Kms / 15430.0 m
----------------------------------------------------------------

Algorithm: dfs

Path Being Explored:
Goal has been found

Path :
A GA HA IA NA OA PA QA RA SA UA VA WA J K JB KB LB MB QB RB SB TB UB VB WB XB
OC VC UC TC SC RC KC PC JC ZC HC GC CC FC EC DC PB T U V W X Y Z AA BA CA DA EA FA
Distance:
27.4 Kms / 27400.0 m
```



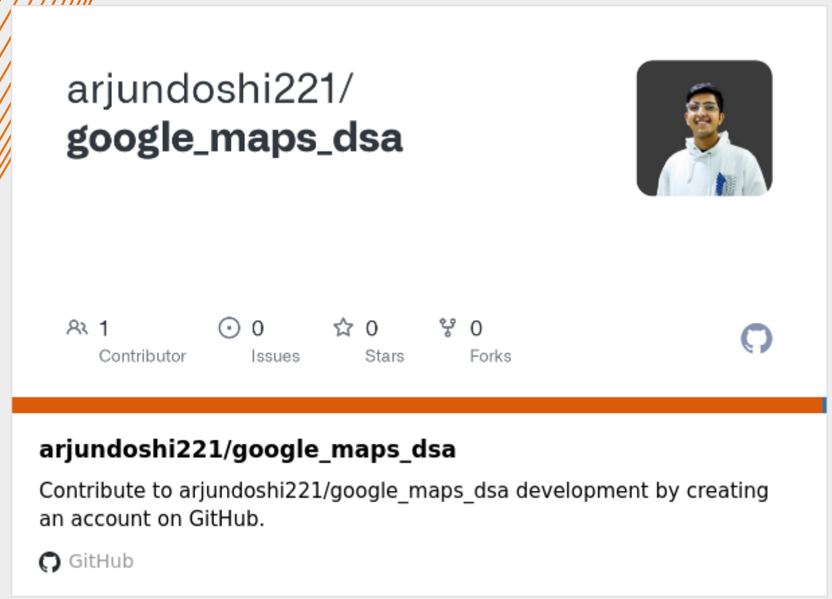



```
Algorithm: a_star_searchHaversine

Path Being Explored:
Goal has been found

Path :
A B C D E F G H I J K L M N O P OB NB PB T U V W X Y Z AA BA CA DA EA FA

Distance:
13.46 Kms / 13460.0 m
--------------------------------------------------------------------------
```



Code Demonstration

https://github.com/arjundoshi221/google_maps_dsa



Thank You