OPTIMAL PATH

B.S. (CS) Project Report

Submitted by:

Name: Hareem Imran	**Seat number: 2024442**
Name: Insharah Riaz	**Seat number: 2024444**
Name: Ujala Shah	**Seat number: 2024468**

November 2022

DEPARTMENT OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING

JINNAH UNIVERSITY FOR WOMEN

5-C NAZIMABAD, KARACHI 74600

Table of Contents

INTRODUCTION:

This project uses search-based algorithms that, if they find an optimal path, always return it. In order to discover the best route for the robot to take from the user-defined start point to the finish point, path finding algorithms including Breadth-First Search (BFS), Dijkstra and A* have been utilized. The majority of the need for planning while travelling is eliminated by optimal-path maps, which show robots or people the optimum route to take from any location within a given terrain area to a desired point. The process of using a computer program to map the shortest path between two places. The operating means of transportation will be bikes (bykea drivers, food panda riders, delivery persons) on Karachi routes.

REQUIREMENTS:

- Python3
- Python Libraries
- Python classes

A* ALGORITHM:

We are use A* Algorithm in our Project. A* is a best-first search algorithm that is informed, meaning it is written in terms of weighted graphs. It starts at a particular starting node in the graph and seeks to find the shortest path to the specified goal node (least distance travelled, shortest time, etc.). A* also constructs a lowest-cost path tree from the start node to the target node, much like Dijkstra. The fact that A* utilizes a function f (n) f(n) f(n) for each node to estimate the overall cost of a path involving that node sets it apart from other search engines and makes it better for many queries. Finding the best route between two nodes in a network can be done using the straightforward and effective A* Search Algorithm. It will be applied to discover the shortest path.

DEFINE & ALGORITHM:

Now, the following steps need to be implemented:

- First, we are construct Class class name is **Node**. Under This Node class, we are defined init functions. **def __init__(self, name, parent, g, h, f):** __init__ is called whenever an object of the class is constructed. To declare all the possible attributes in the __*init*__ method itself. Even if you are not using them right away, you can always assign them as None.
- The open list must be initialized.
- Put the starting node on the open list (leave its f at zero). Initialize the closed list.
- Follow the steps until the open list is non-empty:
- Find the node with the least f on the open list.
- Remove from the open list.
- Produce eight descendants and set as their parent.
- Define all paths also define relative paths.
- For every descendant:

i) If finding a successor is the goal, cease looking

ii)Else, calculate g and h for the successor.

successor.g = q.g + the calculated distance between the successor and the curr.

successor.h = the calculated distance between the successor and the goal. We will cover three heuristics to do this: the Diagonal, the Euclidean, and the Manhattan heuristics.

successor.f = successor.g plus successor.h

totalcost += curr_node.g

```
    path.append(curr_node.name)

    curr_node = open_list.pop()

    closed_list.append(curr_node)

    curr_node = graph.getNode(curr_node.parent,heuristics, end)

    open_list.append(curr_node)

    if(curr_node.name == end):

        path.append(curr_node.name)

        break
```

iii) Skip this successor if a node in the OPEN list with the same location as it but a lower f value than the successor is present.

iv) Skip the successor if there is a node in the CLOSED list with the same position as the successor but a lower f value; otherwise, add the node to the open list end (for loop).

- Push Q into the closed list and end the while loop.

We will now discuss how to calculate the Heuristics for the nodes.

Heuristics

We can easily calculate g, but how do we actually calculate h?

There are two methods that we can use to calculate the value of h:

Determine h's exact value (which is certainly time-consuming).

A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If f(n) represents the final cost, then it can be denoted as :

$f(n) = g(n) + h(n)$, where :

$g(n)$ = cost of traversing from one node to another. This will vary from node to node

$h(n)$ = heuristic approximation of the node's value. This is not a real value but an approximation cost.

CODE:

```
#  Node Class
import matplotlib.pyplot as plt
from PIL import Image

Image1 = Image.open("project_pic.png")
Image1.show()

class Node:

    def __init__(self, name, parent, g, h, f):                              # Initializing the class
        self.name = name
        self.parent = parent
        self.g = g                                                # Distance to start node
        self.h = h                                                # Distance to goal node
        self.f = f                                                # Total cost

    def __eq__(self, other):                                             # Comparing two nodes
        return self.name == other.name

    def __lt__(self, other):                                             # Sorting nodes
        return self.f < other.f

    def __repr__(self):                                                  # Printing nodes
        return ('({0},{1})'.format(self.name, self.f))

    def printNode(self):                                                 # Customized Printing of nodes
        print(self.name, end = " - ")
        print(self.parent, end = " : ")
        print(self.g, end = " : ")
```

```
        print(self.h, end=" : ")
        print(self.f)



#  Graph Class

class Graph:

    def __init__(self, graph_dict=None, directed=True):                # Initialize the class
        self.graph_dict = graph_dict or {}
        self.directed = directed
        if not directed:
            self.make_undirected()

    def make_undirected(self):                # Create an undirected graph by adding
symmetric edges
        for a in list(self.graph_dict.keys()):
            for (b, dist) in self.graph_dict[a].items():
                self.graph_dict.setdefault(b, {})[a] = dist

    def connect(self, A, B, distance=1):                # Add a link from A and B of given
distance, and also add the inverse link if the graph is undirected
        self.graph_dict.setdefault(A, {})[B] = distance
        if not self.directed:
            self.graph_dict.setdefault(B, {})[A] = distance

    def get(self, a, b=None):                # Get neighbors or a neighbor
        links = self.graph_dict.setdefault(a, {})
        if b is None:
            return links
        else:
```

```
            return links.get(b)

    def nodes(self):                                            # Return a list of nodes in the graph
        s1 = set([k for k in self.graph_dict.keys()])
        s2 = set([k2 for v in self.graph_dict.values() for k2, v2 in v.items()])
        nodes = s1.union(s2)
        return list(nodes)

    def getNode(self, city, heuristics, end):                   # Get a specific neighbour which has minimum cost
        nodes = list()
        min = 999
        for (b,dist) in self.graph_dict[city].items():
            if(b == end):
                return Node(city, b, dist, heuristics[b], dist+heuristics[b] )
            nodes.append(Node(city, b, dist, heuristics[b], dist+heuristics[b] ))
            if (dist+heuristics[b]) < min:
                min = dist+heuristics[b]
                minnode = Node(city, b, dist, heuristics[b], dist+heuristics[b] )
        return minnode

    def printgraph(self):                                       # Function to print each edge in the entire graph
        for a in list(self.graph_dict.keys()):
            for (b, dist) in self.graph_dict[a].items():
                print (self.graph_dict.setdefault(a,{})[b], end = " : ")
                print(a, end = " - ")
                print(b)


#  A* function
```

```
def A_Star(graph, heuristics, start, end):
    open_list = list()
    closed_list = list()
    path = list()                                          # Will store the path we are taking
    curr_node = graph.getNode(start,heuristics, end)                    # Starting node
    open_list.append(curr_node)
    totalcost = 0

    if(end not in graph.graph_dict):                            # Incase the goal state does not exist
        print("\n\n---------------------------\nGOAL STATE DOES NOT EXIST\n---------------------------\n\n")
        return  None

    while(curr_node.name != end):                               # Runs Until we cannot find the goal
state or
        totalcost += curr_node.g
        path.append(curr_node.name)
        curr_node = open_list.pop()
        closed_list.append(curr_node)
        curr_node = graph.getNode(curr_node.parent,heuristics, end)
        open_list.append(curr_node)
        if(curr_node.name == end):
            path.append(curr_node.name)
            break

    print("\nFINAL COST -> " + str(totalcost))
    return path

# Main function
```

```
# The main entry point for this module
def main():
    graph = Graph()

    # distance in Kilometer
    graph.connect('North Karachi', 'Sakhi Hassan', 3)
    graph.connect('North Karachi', 'Nagan Chowrangi', 2)
    graph.connect('North Karachi', 'Gulberg', 90)
    graph.connect('Sakhi Hassan', 'Five Star Chowrangi', 5)
    graph.connect('Five Star Chowrangi', 'Nagan Chowrangi', 17)
    graph.connect('Nagan Chowrangi', 'Star Gate', 150)
    graph.connect('Nagan Chowrangi', 'Malir Cantt', 192)
    graph.connect('Malir Cantt', 'Malir Halt', 14)
    graph.connect('Gulberg', 'Rashid Minhas Road', 45)
    graph.connect('Rashid Minhas Road','Shahrah-e-Faisal', 13)
    graph.connect('Shahrah-e-Faisal', 'Gulshan Iqbal',7)
    graph.connect('Gulshan Iqbal', 'Airport Road', 4)
    graph.connect('Airport Road','Malir Cantt', 19)
    graph.connect('Airport Road','Malir Halt', 3)
    graph.connect('Star Gate', 'Jinnah Airport', 6)
    graph.connect('Malir Halt', 'Jinnah Airport', 3)
    graph.connect('Gulberg', 'Jinnah Airport', 104)


    #  graph undirected
    graph.make_undirected()

    # Create heuristics value
    heuristics = {}
```

```
heuristics['North Karachi'] = 366
heuristics['Jinnah Airport'] = 40
heuristics['Airport Road'] = 160
heuristics['Gulshan Iqbal'] = 242
heuristics['Star Gate'] = 176
heuristics['Rashid Minhas Road'] = 244
heuristics['Shahrah-e-Faisal'] = 241
heuristics['Five Star Chowrangi'] = 380
heuristics['Malir Halt'] = 193
heuristics['Malir Cantt'] = 374
heuristics['Nagan Chowrangi'] = 70
heuristics['Gulberg'] = 329
heuristics['Sakhi Hassan'] = 85
heuristics['Karsaz'] = 99


# Print Graph Nodes

print("==============================================================")
print("            \n\t\t\tOPTIMAL PATH\n       ")
print("==============================================================")

print("Bykea Driver destintion Jinnah Airport")
print("--------------------------------\n\n")

print("Kilometer \t Area")
print("--------------------------------\n\n")
graph.printgraph()

# Run search algorithm
```

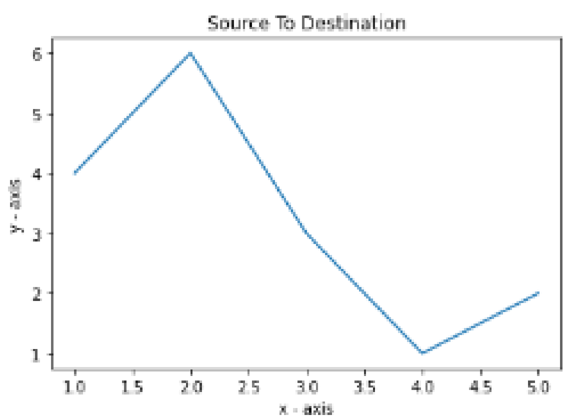
```
    print("--------------------------------\n\n")
    src=input("\n\n(Please Be Sure First Letter must Be Capitalize) \nSource: ")
    Dest=input("Destination: ")
    path = A_Star(graph, heuristics, src, Dest)
    print("\nPATH: " ,end = " ")
    print(path)

# run main method
if __name__ == "__main__": main()

print("Graph")
x = [1,2,3,4,5]
y = [4,6,3,1,2]
plt.plot(x, y)
plt.xlabel('x - axis')
plt.ylabel('y - axis')
plt.title('Source To Destination')
plt.show()
```

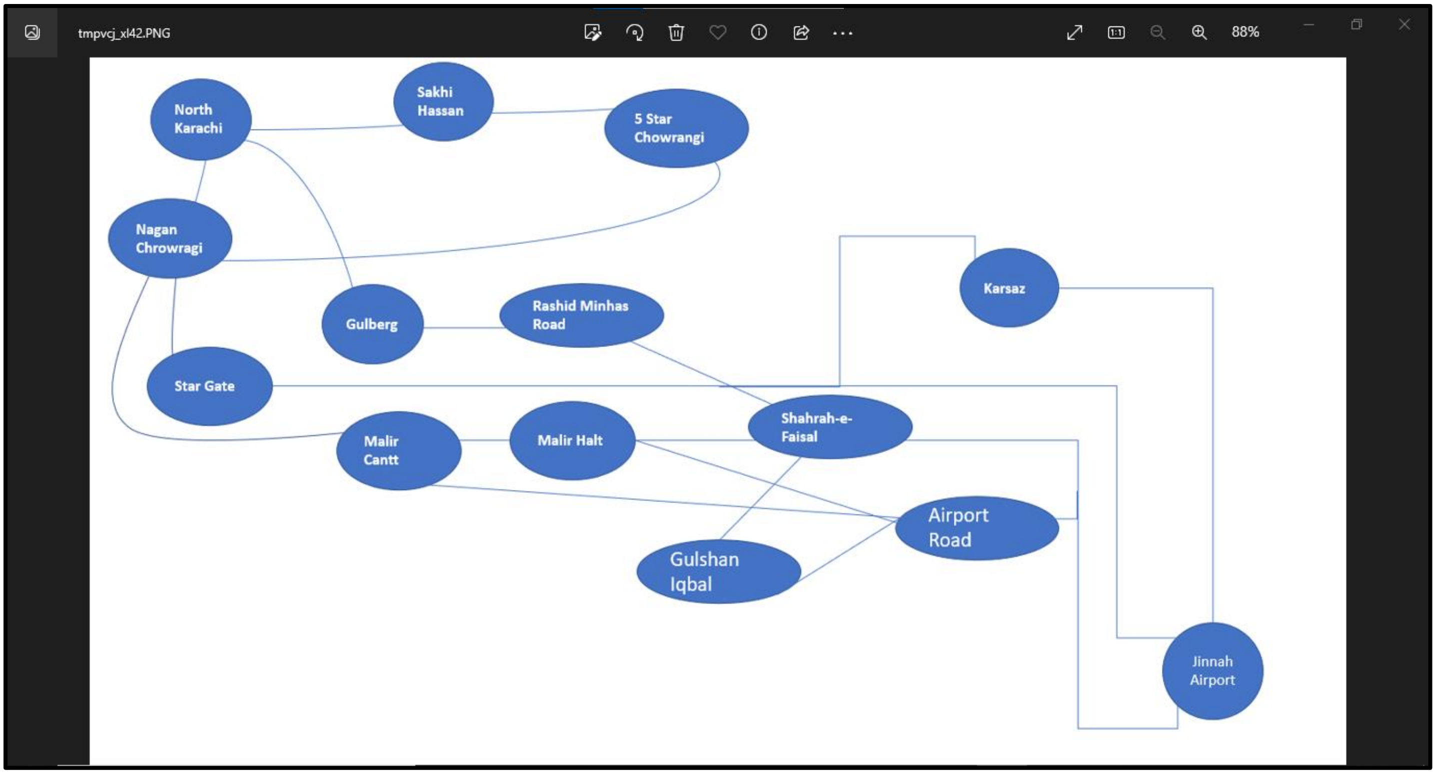
OUTPUT:

```
x = [1,2,3,4,5]
y = [4,6,3,1,2]
plt.plot(x, y)
plt.xlabel('x - axis')
plt.ylabel('y - axis')
plt.title('Source To Destination')
plt.show()
```

```
=================================================================

                        OPTIMAL PATH

=================================================================
Bykea Driver destintion Jinnah Airport
--------------------------------


Kilometer        Area
--------------------------------


3 : North Karachi - Sakhi Hassan
2 : North Karachi - Nagan Chowrangi
90 : North Karachi - Gulberg
5 : Sakhi Hassan - Five Star Chowrangi
3 : Sakhi Hassan - North Karachi
17 : Five Star Chowrangi - Nagan Chowrangi
5 : Five Star Chowrangi - Sakhi Hassan
150 : Nagan Chowrangi - Star Gate
192 : Nagan Chowrangi - Malir Cantt
2 : Nagan Chowrangi - North Karachi
17 : Nagan Chowrangi - Five Star Chowrangi
14 : Malir Cantt - Malir Halt
192 : Malir Cantt - Nagan Chowrangi
19 : Malir Cantt - Airport Road
45 : Gulberg - Rashid Minhas Road
104 : Gulberg - Jinnah Airport
90 : Gulberg - North Karachi
13 : Rashid Minhas Road - Shahrah-e-Faisal
45 : Rashid Minhas Road - Gulberg
7 : Shahrah-e-Faisal - Gulshan Iqbal
13 : Shahrah-e-Faisal - Rashid Minhas Road
4 : Gulshan Iqbal - Airport Road
```

```
13 : Rashid Minhas Road - Shahrah-e-Faisal
45 : Rashid Minhas Road - Gulberg
7 : Shahrah-e-Faisal - Gulshan Iqbal
13 : Shahrah-e-Faisal - Rashid Minhas Road
4 : Gulshan Iqbal - Airport Road
7 : Gulshan Iqbal - Shahrah-e-Faisal
19 : Airport Road - Malir Cantt
3 : Airport Road - Malir Halt
4 : Airport Road - Gulshan Iqbal
6 : Star Gate - Jinnah Airport
150 : Star Gate - Nagan Chowrangi
3 : Malir Halt - Jinnah Airport
14 : Malir Halt - Malir Cantt
3 : Malir Halt - Airport Road
104 : Jinnah Airport - Gulberg
6 : Jinnah Airport - Star Gate
3 : Jinnah Airport - Malir Halt
--------------------------------



(Please Be Sure First Letter must Be Capitalize)
Source: North Karachi
Destination: Jinnah Airport

FINAL COST -> 158

PATH:  ['North Karachi', 'Nagan Chowrangi', 'Star Gate', 'Jinnah Airport']
Graph
```




tmpvcj_xl42.PNG
88%
North Karachi
Sakhi Hassan
5 Star Chowrangi
Nagan Chowragi
Gulberg
Rashid Minhas Road
Karsaz
Star Gate
Shahrah-e-Faisal
Malir Cantt
Malir Halt
Airport Road
Gulshan Iqbal
Jinnah Airport