Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)

Subject: Artificial Intelligence (DJSDSC502)

AY: 2024-25

Experiment 6

(Optimization)

DHRUV SHAH 60009220132 D1-1

Aim: Find the shortest path between two places using A* Algorithm.

Theory:

A* is a searching algorithm that is used to find the shortest path between an initial and a final point.

It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It still remains a widely popular algorithm for graph traversal.

It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

Another aspect that makes A* so powerful is the use of weighted graphs in its implementation. A weighted graph uses numbers to represent the cost of taking each path or course of action. This means that the algorithms can take the path with the least cost, and find the best route in terms of distance and time.

Explanation

In the event that we have a grid with many obstacles and we want to get somewhere as rapidly as possible, the A* Search Algorithms are our savior. From a given starting cell, we can get to the target cell as quickly as possible. It is the sum of two variables' values that determines the node it picks at any point in time.

At each step, it picks the node with the smallest value of 'f' (the sum of 'g' and 'h') and processes that node/cell. 'g' and 'h' is defined as simply as possible below:

- 'g' is the distance it takes to get to a certain square on the grid from the starting point, following the path we generated to get there.
- 'h' is the heuristic, which is the estimation of the distance it takes to get to the finish line from that square on the grid.

Heuristics are basically educated guesses. It is crucial to understand that we do not know the distance to the finish point until we find the route since there are so many things that might get in the way (e.g., walls, water, etc.). In the coming sections, we will dive deeper into how to calculate the heuristics.

Algorithm

Initial condition - we create two lists - Open List and Closed List.

Now, the following steps need to be implemented -

- 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:

1. Find the node with the least f on the open list and name it "q".
2. Remove Q from the open list.
3. Produce q's eight descendants and set q as their parent.
4. 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 q.

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

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.

Lab Assignment to do:
Solve the following Shortest Path Algorithm:
1. Consider 40 -45 geo locations between Juhu beach till Gateway of India. 5-8 locations not in the route and choice of location should keep in mind two different possible path.
2. Find the shortest path between Juhu beach till Gateway of India.
3. Use Havesine formula for distance calculation.

Double-click (or enter) to edit

Import Necessary Libraries

```
import math from queue import
PriorityQueue import
matplotlib.pyplot as plt
```

Dictionary of {'city': (latitude, longitude)}

```
cities_coordinates = {
"Mumbai": (19.076, 72.8777),
    "Pune": (18.5204, 73.8567),
    "Vapi": (20.3719, 72.9049),
    "Nagpur": (21.1458, 79.0882),
    "Nashik": (19.9975, 73.7898),
    "Aurangabad": (19.8762, 75.3433),
    "Ahmedabad": (23.0225, 72.5714),
    "Surat": (21.1702, 72.8311),
    "Jaipur": (26.9124, 75.7873),
    "Jodhpur": (26.2389, 73.0243),
    "Kota": (25.2138, 75.8648),
    "Jhansi": (25.4484, 78.5685),
    "Bhopal": (23.2599, 77.4126),
    "Indore": (22.7196, 75.8577),
    "Faridabad": (28.4089, 77.3178),
    "Ghaziabad": (28.6692, 77.4538),
    "New Delhi": (28.6139, 77.209),
"Noida": (28.5355, 77.391),
    "Gurugram": (28.4595, 77.0266),
    "Manali": (32.2396, 77.1887),
    "Dharamshala": (32.219, 76.3234),
    "Shimla": (31.1048, 77.1734),
    "Spiti Valley": (32.246, 78.0081),
    "Chandigarh": (30.7333, 76.7794),
    "Jammu": (32.7266, 74.857),
    "Amritsar": (31.634, 74.8723),
    "Dalhousie": (32.5373, 75.9713),
    "Mussoorie": (30.4599, 78.066),
"Dehradun": (30.3165, 78.0322),
"Katra": (32.9856, 74.9389),
    "Srinagar": (34.0837, 74.7973),
    "Kargil": (34.5553, 76.135),
    "Leh": (34.1526, 77.577),
}
```

City Graph

```
city_graph = {
    "Mumbai": ["Pune", "Nashik", "Vapi", "Surat"],
    "Pune": ["Mumbai", "Nashik", "Aurangabad"],
    "Nashik": ["Mumbai", "Pune", "Aurangabad"],
    "Aurangabad": ["Nashik", "Indore"],
    "Vapi": ["Mumbai", "Surat"],
    "Surat": ["Mumbai", "Vapi", "Ahmedabad"],
    "Ahmedabad": ["Surat", "Indore"],
    "Indore": ["Ahmedabad", "Bhopal"],
    "Bhopal": ["Indore", "Jhansi"],

    "Jhansi": ["Bhopal", "Kota"],
    "Kota": ["Jhansi",  "Jaipur"],

    "New Delhi": ["Faridabad", "Ghaziabad", "Noida", "Jaipur", "Manali", "Amritsar"],
    "Noida": ["New Delhi", "Ghaziabad", "Kota"],
    "Ghaziabad": ["New Delhi", "Faridabad", "Noida"],
    "Faridabad": ["New Delhi", "Ghaziabad", "Gurugram"],
    "Gurugram": ["Faridabad", "New Delhi"],

    "Jaipur": [ "Jodhpur", "Gurugram"],
    "Jodhpur": ["Jaipur"],
```

```
    "Manali": ["Dharamshala", "Shimla"],
    "Dharamshala": ["Manali", "Shimla"],
    "Shimla": ["Dharamshala", "Manali"],
    "Srinagar": ["Jammu", "Leh"],
    "Jammu": ["Srinagar", "Amritsar", "Leh"],
    "Amritsar": ["Jammu", "Srinagar"],
    "Leh": ["Srinagar", "Kargil"],
    "Kargil": ["Leh"],
    "Mussoorie": ["Dehradun"],
    "Dehradun": ["Mussoorie", "Haridwar"],
    "Spiti Valley": ["Kinnaur"],
    "Kinnaur": ["Spiti Valley"],
}
```

Eucledean Distance

```
def euclidean_distance(coord1, coord2):     return math.sqrt((coord1[0] -
coord2[0])**2 + (coord1[1] - coord2[1])**2)


euclidean_distance((cities_coordinates["Mumbai"][0], cities_coordinates["Mumbai"][1]),
                               (cities_coordinates["Noida"][0], cities_coordinates["Noida"][1]))
```

```
10.481031301355797
```

Heuristic Function

```
def heuristic(node, goal):
    return euclidean_distance(cities_coordinates[node][0], cities_coordinates[node][1],
                              cities_coordinates[goal][0], cities_coordinates[goal][1])
```

Get Neighbours

```
def get_neighbors(city, city_graph):
return city_graph.get(city, [])


get_neighbors("Mumbai",city_graph)
```

```
['Pune', 'Nashik', 'Vapi', 'Surat']
```

Reconstruct Path

```
def reconstruct_path(came_from, current):
    path = [current]     while
current in came_from:
current = came_from[current]
path.append(current)
path.reverse()     return path


came_from = {
'B': 'A',
    'E': 'B',
    'G': 'E'
}

goal = 'G'
reconstruct_path(came_from,goal)
```

```
['A', 'B', 'E', 'G']
```

Propagate Improvement

```
def propagate_improvement(city, g_score, f_score, came_from, open_set, city_graph, cities_coordinates, goal):
    print()     for neighbor in get_neighbors(city,
city_graph):
```

```
        tentative_g_score = g_score[city] + euclidean_distance(cities_coordinates[city], cities_coordinates[neighbor])
print(neighbor,end = " ")
        if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
            came_from[neighbor] = city             g_score[neighbor] = tentative_g_score             f_score[neighbor] =
g_score[neighbor] + euclidean_distance(cities_coordinates[neighbor], cities_coordinates[goal])             if neighbor not in
[item[1] for item in open_set.queue]:
                open_set.put((f_score[neighbor], neighbor))             if neighbor in g_score:
propagate_improvement(neighbor, g_score, f_score, came_from, open_set, city_graph, cities_coordinates, goal)


city_graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F', 'G'],
    'F': ['C', 'E'],
    'G': ['E']
} cities_coordinates =
{
    'A': (0, 0),
    'B': (2, 2),
    'C': (2, -2),
    'D': (4, 3),
    'E': (4, 1),
    'F': (6, -2),
    'G': (6, 2)
} from math import
inf

g_score = {city: inf for city in city_graph} f_score = {city: inf for city in
city_graph} came_from = {} g_score['A'] = 0 f_score['A'] =
euclidean_distance(cities_coordinates['A'], cities_coordinates['G']) from queue
import PriorityQueue

open_set = PriorityQueue()
open_set.put((f_score['A'], 'A'))
goal = 'G'
propagate_improvement('A', g_score, f_score, came_from, open_set, city_graph, cities_coordinates, goal)
```

```
B A D B E B F C A F E G E C A F C E
```

```
print("g_score:", g_score) print("f_score:",
f_score) print("came_from:", came_from)
print("Open set contents:", list(open_set.queue))
```

```
g_score: {'A': 0, 'B': 2.8284271247461903, 'C': 2.8284271247461903, 'D': 5.06449510224598, 'E': 5.06449510224598, 'F': 6.82842712474
f_score: {'A': 6.324555320336759, 'B': 6.82842712474619, 'C': 8.485281374238571, 'D': 7.3005630797457695, 'E': 7.3005630797457695,
came_from: {'B': 'A', 'D': 'B', 'E': 'B', 'F': 'C', 'C': 'A', 'G': 'E'} Open set contents: [(6.324555320336759, 'A'),
(6.82842712474619, 'B'), (7.3005630797457695, 'D'), (7.3005630797457695, 'E'), (12.670
```

A Star

```
def astar(start, goal, city_graph, cities_coordinates):
    open_set = PriorityQueue()     open_set.put((0, start))     came_from = {}     g_score =
{start: 0}     f_score = {start: euclidean_distance(cities_coordinates[start],
cities_coordinates[goal])}

    while not open_set.empty():
current = open_set.get()[1]

        if current == goal:             return
reconstruct_path(came_from, current)

        for neighbor in get_neighbors(current, city_graph):             tentative_g_score = g_score[current] +
euclidean_distance(cities_coordinates[current], cities_coordinates[neighbor])

            if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current                 g_score[neighbor] = tentative_g_score
f_score[neighbor] = g_score[neighbor] + euclidean_distance(cities_coordinates[neighbor], cities_coordinates[goal])
open_set.put((f_score[neighbor], neighbor))
                if neighbor in g_score:                     propagate_improvement(neighbor, g_score, f_score, came_from,
open_set, city_graph, cities_coordinates, goal)     return None


astar("Mumbai", "New Delhi", city_graph, cities_coordinates)
```

```
Mumbai Nashik Mumbai Pune Aurangabad Nashik Indore Ahmedabad Surat Mumbai Vapi Mumbai Surat Ahmedabad Indore Bhopal Indore Jhansi Bh
['Mumbai',
 'Nashik',
 'Aurangabad',
 'Indore',
 'Bhopal',
 'Jhansi',
 'Kota',
 'Jaipur',
 'Gurugram',
 'New Delhi']
```

Calculate Path Length

```
def calculate_path_length(path, cities_coordinates):
    total_distance = 0     for i
in range(len(path) - 1):
        total_distance += euclidean_distance(cities_coordinates[path[i]], cities_coordinates[path[i+1]])
return total_distance


path = ['A', 'B', 'E', 'G']
calculate_path_length(path, cities_coordinates)
```

```
7.3005630797457695
```

Print Path function

```
def print_path(path, cities_coordinates):
    print("\nShortest path:")     for i, city in enumerate(path):
print(f"Step {i + 1}: {city}")     total_distance =
calculate_path_length(path, cities_coordinates)     print(f"\nTotal
distance: {total_distance:.2f} units")
```

Shortest Distance

```
def find_shortest_path(start, goal, city_graph, cities_coordinates):
    path = astar(start, goal, city_graph, cities_coordinates)
if path:
        print_path(path, cities_coordinates)
else:         print("No path found.")
```

Test run

```
start_city = "Mumbai"
end_city = "Leh"
find_shortest_path(start_city, end_city, city_graph, cities_coordinates)
```

```
Mumbai Nashik
Mumbai Pune Aurangabad
Nashik Indore
Ahmedabad
Surat
Mumbai Vapi
Mumbai Surat Ahmedabad Indore Bhopal
Indore Jhansi
Bhopal Kota
Jhansi Jaipur
Jodhpur
Jaipur Gurugram
Faridabad
```

```
New Delhi
Faridabad Ghaziabad
New Delhi Faridabad Noida
New Delhi Ghaziabad Kota Noida
New Delhi Ghaziabad Kota Jaipur Manali
Dharamshala
Manali Shimla
Dharamshala Manali Shimla
Dharamshala Manali Amritsar
Jammu
Srinagar
Jammu Leh
Srinagar Kargil
Leh Amritsar Leh
Srinagar Kargil
Leh Srinagar
Jammu Leh Ghaziabad
New Delhi Faridabad Noida Gurugram New Delhi
Faridabad Ghaziabad
New Delhi Faridabad Noida
New Delhi Ghaziabad Kota Noida
New Delhi Ghaziabad Kota Jaipur Manali
Dharamshala
Manali Shimla Shimla
Dharamshala Manali Amritsar
Jammu
Srinagar
Jammu Leh Amritsar Leh
Srinagar Kargil
Leh Srinagar
Jammu Leh Aurangabad
Nashik Indore
Ahmedabad
Surat
Mumbai Vapi
Mumbai Surat Ahmedabad Indore Bhopal
Indore Jhansi
Bhopal Kota
Jhansi Jaipur
Jodhpur
Jaipur Gurugram
Faridabad
New Delhi
F  id b d Gh  i b d
```

```
print("""
Conclusion: The Code for A* Algorithms has been implemented successfully.
It is an informed searching algorithm which tries to create a best path using heuristic function. Although this is efficient in time,
it may not always put out optimal path between two places.

""")
```

```
Conclusion: The Code for A* Algorithms has been implemented successfully.
It is an informed searching algorithm which tries to create a best path using heuristic function. Although this is efficient in time
it may not always put out optimal answers
```