Name: Shashwat Shah

SAP-ID: 60004220126

TY BTECH DIV B, Batch : C22

Aim: Identify and analyze informed search Algorithm to solve the problem.
Implement A* search algorithm to reach goal state.

Theory:

A*:

A* (pronounced "A-star") is a widely used and effective search algorithm in computer science and artificial intelligence. It is particularly employed in pathfinding and graph traversal problems, where you need to find the shortest path from a start node to a target node while considering the associated costs of moving through the graph or search space. A* is a heuristic search algorithm, which means it uses a combination of actual cost (the cost to reach a node from the start) and an estimated cost (a heuristic value) to make informed decisions during the search. This combination of real and estimated costs allows A* to efficiently explore the search space and find the optimal path while avoiding unnecessary exploration, making it highly efficient and effective.

The key to A*'s success is its ability to balance between completeness and efficiency. By using a heuristic, A* intelligently prioritizes the nodes to explore, favoring those that are likely to lead to the goal node and minimizing the number of nodes evaluated. This makes A* suitable for a wide range of applications, including GPS navigation, robotics, video game pathfinding, and more. The choice of heuristic can significantly impact A*'s performance, and when an admissible heuristic is used (one that never overestimates the cost to the goal), A* is guaranteed to find the optimal path, making it a powerful and versatile tool in solving complex search and optimization problems.

Code:

```
# A*
from collections import deque

class Graph:
    def __init__(self, adjac_lis):
        self.adjac_lis = adjac_lis

    def get_neighbors(self, v):
        return self.adjac_lis[v]

    def heuristic(self, n):
        H = {
```

```
            'S': 15,
            '1': 14,
            '2': 10,
            '3':  8,
            '4': 12,
            '5': 10,
            '6': 10,
            '7': 0
        }

        return H[n]

    def a_star(self, start, stop):

        open_list = set([start])
        closed_list = set([])

        distance = {} # Distance from Start.
        distance[start] = 0

        adjacent_nodes = {} # Adjacent Mapping of all Nodes
        adjacent_nodes[start] = start

        while len(open_list) > 0:
            n = None

            for v in open_list:
                if n == None or distance[v] + self.heuristic(v) < distance[n] + self.heuristic(n):
                    n = v

            if n == None:
                print('Path does not exist!')
                return None

            if n == stop: # If current node is stop, we restart
                reconst_path = []

                while adjacent_nodes[n] != n:
                    reconst_path.append(n)
                    n = adjacent_nodes[n]

                reconst_path.append(start)

                reconst_path.reverse()

                print('\nPath found: {}\n'.format(reconst_path))
                return reconst_path
```

```
            for (m, weight) in self.get_neighbors(n): # Neighbours of current
node

                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    adjacent_nodes[m] = n
                    distance[m] = distance[n] + weight

                else: # Check if its quicker to visit n then m
                    if distance[m] > distance[n] + weight:
                        distance[m] = distance[n] + weight
                        adjacent_nodes[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)


            open_list.remove(n) # Since all neighbours are inspected
            closed_list.add(n)
            print("OPEN LIST : ", end="")
            print(open_list)
            print("CLOSED LIST : ", end="")
            print(closed_list)
            print("- - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - -")

        print('Path does not exist!')
        return None


adjacent_list2 = {
    'S': [('1', 3), ('4', 4)],
    '1': [('S', 3), ('2', 4), ('4', 5)],
    '2': [('1', 4), ('3', 4), ('5', 5)],
    '3': [('2', 4)],
    '4': [('S', 4), ('1', 5), ('5', 2)],
    '5': [('4', 2), ('2', 5), ('6', 4)],
    '6': [('5', 4), ('7', 3)],
    '7': [('6', 3)],
}

g = Graph(adjacent_list2)
g.a_star('S', '7')
```

output:

```
OPEN LIST : {'4', '1'}
CLOSED LIST : {'S'}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
OPEN LIST : {'5', '1'}
CLOSED LIST : {'4', 'S'}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
OPEN LIST : {'1', '2', '6'}
CLOSED LIST : {'5', '4', 'S'}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
OPEN LIST : {'2', '6'}
CLOSED LIST : {'5', '4', 'S', '1'}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
OPEN LIST : {'3', '6'}
CLOSED LIST : {'1', '2', '4', '5', 'S'}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
OPEN LIST : {'6'}
CLOSED LIST : {'3', '1', '2', '4', '5', 'S'}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
OPEN LIST : {'7'}
CLOSED LIST : {'3', '1', '2', '4', '6', '5', 'S'}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Path found: ['S', '4', '5', '6', '7']

['S', '4', '5', '6', '7']
```

Conclusion:

Thus we implemented the A* algorithm.