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)

A.Y. 2022-2023

Subject: Artificial Intelligence **SAP ID: 60004220253 – Devansh Mehta**

Experiment No. 03

Aim: Implementation of A*

Theory:

A* (pronounced "A star") is a widely used graph traversal and pathfinding algorithm that finds the shortest path from a start node to an end node in a weighted graph. A* uses a heuristic to estimate the cost of reaching the goal fromthe current node, in addition to the actual cost from the start node. It combines these two costs to prioritise nodes for exploration. The algorithm maintains two sets of nodes: the open set (nodes to be evaluated) and the closed set (nodes that have already been evaluated). A* selects nodes from the open set based on a costfunction:

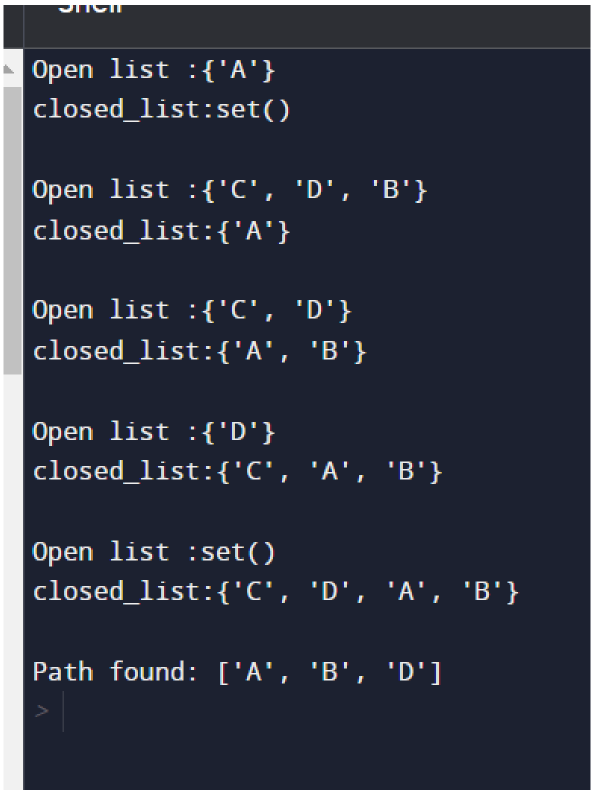
f(n) = g(n) + h(n), where *g*(*n*) is the cost of reaching node *n* from the start node,and *h*(*n*) is the heuristic estimate of the cost to reach the goal from *n*.

A* continues evaluating nodes until the goal node is reached or the open set is empty.

Code:

```
class Graph:
    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list
    def get_neighbors(self, v):
        return self.adjacency_list[v]
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }
        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        open_list = set([start_node])
        closed_list = set([])
        g = {}
        g[start_node] = 0
        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None
            print(f'Open list :{open_list} \nclosed_list:{closed_list}\n')
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v
            if n == None:
                print('Path does not exist!')
                return None

            if n == stop_node:
                open_list.remove(n)
                closed_list.add(n)
                print(f'Open list :{open_list} \nclosed_list:{closed_list}\n')
                reconst_path = []
                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]
                reconst_path.append(start_node)
                reconst_path.reverse()
                print('Path found: {}'.format(reconst_path))
                return reconst_path

            for (m, weight) in self.get_neighbors(n):
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n
                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            open_list.remove(n)
            closed_list.add(n)

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

adjacency_list = {
 'A': [('B', 1), ('C', 3), ('D', 7)],
 'B': [('D', 5)],
 'C': [('D', 12)]
}

graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

Output:



```
Open list :{'A'}
closed_list:set()

Open list :{'C', 'D', 'B'}
closed_list:{'A'}

Open list :{'C', 'D'}
closed_list:{'A', 'B'}

Open list :{'D'}
closed_list:{'C', 'A', 'B'}

Open list :set()
closed_list:{'C', 'D', 'A', 'B'}

Path found: ['A', 'B', 'D']
>
```

Conclusion:

A* algorithm is highly efficient and versatile in finding the shortest path in graphs. It guarantees an optimal solution, provided that the heuristic used is admissible (never overestimates the true cost to reach the goal) and consistent (satisfies the triangle inequality). The A* algorithm strikes a balance between breadth-first search (BFS) and Dijkstra's algorithm, as it explores the most promising paths first due to its intelligent use of heuristics. However, the performance of A* heavily depends on the quality of the heuristic function. A well-designed heuristic can significantly speed up the search process, while a poorly designed heuristic may cause the algorithm to explore unnecessary nodes,affecting its efficiency.

In conclusion, A* is a powerful algorithm for solving pathfinding problems and is widely used in various applications, including robotics, video games, and maprouting.