EXPERIMENT – 01

AIM:

To Implement A* Search Algorithm on any Problem.

DESCRIPTION:

A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

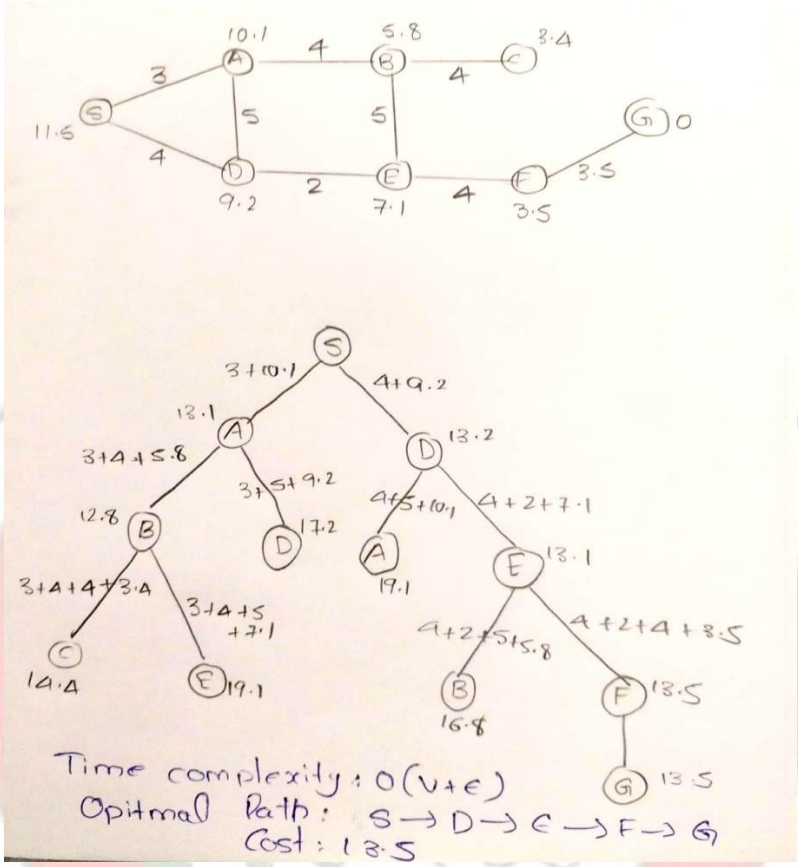
Informally speaking, A* Search algorithms, unlike other traversal techniques, it has "brains". What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections. And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.

What A* Search Algorithm does is that at each step it picks the node according to a value-'**f**' which is a parameter equal to the sum of two other parameters – '**g**' and '**h**'. At each step it picks the node/cell having the lowest '**f**', and process that node/cell.
We define '**g**' and '**h**' as simply as possible below

g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way.

EXAMPLE:



ALGORITHM:

1. Initialize the open list
2. Initialize the closed list put the starting node on the open list
   (you can leave its f at zero)
3. while the open list is not empty
   a) find the node with the least f on the open list, call it "q"
   b) pop q off the open list
   c) generate q's 8 successors and set their parents to q
   d) for each successor
      i) if successor is the goal, stop search
      ii) else, compute both g and h for successor
      successor.g = q.g + distance between successor and q
      successor.h = distance from goal to successor
      (This can be done using many ways, we will discuss three heuristics-
      Manhattan, Diagonal and Euclidean Heuristics)

successor.f = successor.g + successor.h

iii) if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor

iv) if a node with the same position as is in the CLOSED list which has a lower f than successor, skip this successor, add the node to the open list

end (for loop)

e) push q on the closed list

end (while loop)

CODE:

```
import heapq
class Node:
    def __init__(self, name, heuristic):
        self.name = name
        self.heuristic = heuristic
        self.neighbors = []
    def add_neighbor(self, neighbor, weight):
        self.neighbors.append((neighbor, weight))
def a_star_search(start, goal):
    open_list = []
    heapq.heappush(open_list, (0, start))
    came_from = {}
    g_score = {start: 0}
    f_score = {start: start.heuristic}

    while open_list:
        current_f, current = heapq.heappop(open_list)
        if current == goal:
            path = []
            while current in came_from:
                path.append(current.name)
                current = came_from[current]
            path.append(start.name)
            return path[::-1], g_score[goal]
```

```
        for neighbor, weight in current.neighbors:
            tentative_g_score = g_score[current] + weight
            if tentative_g_score < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = tentative_g_score + neighbor.heuristic
                heapq.heappush(open_list, (f_score[neighbor], neighbor))
    return None, float('inf')

nodes = {}
num_nodes = int(input("Enter the number of nodes: "))
for _ in range(num_nodes):
    name = input("Enter node name: ")
    heuristic = float(input(f"Enter heuristic value for {name}: "))
    nodes[name] = Node(name, heuristic)

num_edges = int(input("Enter the number of edges: "))
for _ in range(num_edges):
    node1 = input("Enter the start node of the edge: ")
    node2 = input("Enter the end node of the edge: ")
    weight = float(input(f"Enter the weight of the edge between {node1} and
{node2}: "))
    nodes[node1].add_neighbor(nodes[node2], weight)
    nodes[node2].add_neighbor(nodes[node1], weight)

start_node = nodes[input("Enter the start node: ")]
goal_node = nodes[input("Enter the goal node: ")]

path, cost = a_star_search(start_node, goal_node)
if path:
    print("Path found:", " -> ".join(path))
    print("Total cost:", cost)
else:
    print("No path found")
```

OUTPUT ANALYSIS:

Time Complexity

In general, its O(V+E) where, V=No. of Nodes, E=No of Vertices but in AIML, its O(bd) where b=Branch Factor (No. of Children of a particular node) and d=depth.

1. Worst-Case Time Complexity:

- The time complexity of A* depends on the heuristic. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) d: O(b^d), where b is the branching factor (the average number of successors per state).
- This occurs when the heuristic function h(n) is not very informative, making the algorithm behave similarly to a breadth-first search.

2. Best-Case Time Complexity:

- In the best case, when the heuristic h(n) is perfect (meaning it exactly predicts the cost to reach the goal from any node), A* runs in linear time O(d).

3. Average-Case Time Complexity:

- The average-case complexity is typically better than the worst case but worse than the best case. It depends on how well the heuristic approximates the actual cost to the goal. If the heuristic is admissible and consistent (or monotonic), A* tends to perform well, though it's still usually exponential in nature, closer to O(bd)O(b^d).

Space Complexity

The space complexity of the A* algorithm is determined by:

- A* needs to store all generated nodes in memory to keep track of the paths it has explored. This includes the open list (frontier) and the closed list (visited nodes).
- The open list, in particular, can grow exponentially, leading to a space complexity of O(b^d), where:
- b is the branching factor (the average number of successors per state).
- d is the depth of the optimal solution.

Summary

1. **Time Complexity**: O(b^d)
2. **Space Complexity**: O(b^d)

CHAITANYA BHARATHI
INSTITUTE OF TECHNOLOGY
స్వయం తేజస్వినా భవ
1979