Experiment No: 5 Date:

A* Search Algorithm

Aim: To Implement the A* Search Algorithm

Theory:

The A* search algorithm is a widely used technique in computer science for finding the shortest path between nodes in a graph. It combines the advantages of both Dijkstra's algorithm and Greedy Best-First Search by using a heuristic to guide its search.

Components of A* Search:

- Nodes (Vertices): These represent points in the graph.
- Edges: These represent the connections between nodes.
- Costs or Weights: Each edge has an associated cost or weight, which represents the distance or cost of traveling between two nodes.
- Heuristic Function (h): A* requires a heuristic function, denoted as $h(n)$, which estimates the cost from the current node to the goal node. This heuristic is problem-specific and should be admissible (never overestimates the true cost) and consistent (satisfies the triangle inequality).
- Evaluation Function (f): The evaluation function, denoted as $f(n)$, combines the actual cost from the start node to the current node ($g(n)$) and the heuristic cost from the current node to the goal node ($h(n)$). It's defined as $f(n)=g(n)+h(n)$.

Advantages of A* Search:

- Completeness: A* is complete, meaning it will always find a solution if one exists (provided the graph is finite).
- Optimality: If the heuristic is admissible, A* is optimal, guaranteeing the shortest path.
- Efficiency: A* typically explores fewer nodes than uninformed search algorithms like Breadth-First Search or Depth-First Search.

Disadvantages of A* Search:

- A disadvantage of the A* search algorithm is that it can be computationally expensive if the heuristic function is poorly chosen or if the graph has a high branching factor.

Algorithm:

```
Procedure A*()
  open ← List(start)
  f(start) ← h(start)
  parent(start) ← NIL
  closed ← {}
  while open is not EMPTY
    do
      Remove node n from open such that f(n) has the lowest value
      Add n to closed
      if GoalTest(n) = TRUE
          then return ReconstructPath(n)
      neighbours ← MoveGen(n)
      for each m ∈ neighbours
          do switch
            case mÏopen AND mÏclosed :     /* new node */
                Add m to open
                parent(m) ← n
                g(m)  ← g(n) + k(n, m)
                f(m) ← g(m) + h(m)

            case m ∈ open :
                if (g(n) + k(n, m)) < g(m)
                  then  parent(m) ← n
                        g(m) ← g(n) + k(n, m)
                        f(m) ← g(m) + h(m)

            case m ∈ closed :      /* like above case */
                if (g(n) + k(n, m)) < g(m)
                  then  parent(m) ← n
                      g(m) ← g(n) + k(n, m)
                      f(m) ← g(m) + h(m)
                      PropagateImprovement(m)
  return FAILURE

PropagateImprovement(m)
  neighbours ← MoveGen(m)
  for each s ∈ neighbours
    do newGvalue ← g(m) + k(m, s)
      if newGvalue < g(s)
        then  parent(s) ← m
            g(s) ← newGvalue
            if s ∈ closed
          then PropagateImprovement(s)
```

Example:

In a maze-solving scenario, the A* search algorithm efficiently finds the shortest path from a start point to a goal point while considering obstacles. It starts by evaluating adjacent cells based on their distance from the start and a heuristic estimate of their distance to the goal. At each step, it chooses the cell with the lowest combined cost and heuristic value. This process continues until the goal is reached or all possible paths are explored. A* is widely used in robotics for navigation in dynamic environments due to its ability to find optimal paths quickly while considering obstacles and constraints.

Program:

```
import heapq
class Graph:
    def __init__(self):
        self.nodes = set()
        self.edges = {}
        self.heuristic = {}

    def add_node(self, value, heuristic=0):
        self.nodes.add(value)
        self.heuristic[value] = heuristic

    def add_edge(self, from_node, to_node, cost):
        if from_node not in self.edges:
            self.edges[from_node] = []
        self.edges[from_node].append((to_node, cost))

    def get_neighbors(self, node):
        if node in self.edges:
            return self.edges[node]
        else:
            return []

    def a_star(self, start, goal):
        frontier = [(0, start)]
        came_from = {}
        cost_so_far = {start: 0}

        while frontier:
            current_cost, current_node = heapq.heappop(frontier)
            if current_node == goal:
                path = []
                while current_node in came_from:
                    path.append(current_node)
                    current_node = came_from[current_node]
                path.append(start)
                path.reverse()

                # Calculate total cost
                total_cost = 0
                for i in range(len(path) - 1):
                    total_cost += self.get_cost(path[i], path[i+1])
                return path, total_cost
            for neighbor, cost in self.get_neighbors(current_node):
                new_cost = cost_so_far[current_node] + cost
                if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                    cost_so_far[neighbor] = new_cost
                    priority = new_cost + self.heuristic[neighbor]
                    heapq.heappush(frontier, (priority, neighbor))
                    came_from[neighbor] = current_node
```

```
        return None, None

    def get_cost(self, from_node, to_node):
        for neighbor, cost in self.edges[from_node]:
            if neighbor == to_node:
                return cost
        return None


graph = Graph()
graph.add_node('S', heuristic=14)
graph.add_node('A', heuristic=11)
graph.add_node('B', heuristic=10)
graph.add_node('C', heuristic=8)
graph.add_node('D', heuristic=12)
graph.add_node('E', heuristic=5)
graph.add_node('F', heuristic=12)
graph.add_node('H', heuristic=8)
graph.add_node('I', heuristic=10)
graph.add_node('J', heuristic=8)
graph.add_node('K', heuristic=6)
graph.add_node('L', heuristic=10)
graph.add_node('M', heuristic=7)
graph.add_node('N', heuristic=4)
graph.add_node('O', heuristic=8)
graph.add_node('P', heuristic=5)
graph.add_node('Q', heuristic=1)
graph.add_node('R', heuristic=6)
graph.add_node('T', heuristic=2)
graph.add_node('G', heuristic=0)

graph.add_edge('S', 'D', 25)
graph.add_edge('D', 'A', 32)
graph.add_edge('D', 'F', 24)
graph.add_edge('A', 'B', 11)
graph.add_edge('A', 'H', 36)
graph.add_edge('B', 'C', 24)
graph.add_edge('B', 'K', 42)
graph.add_edge('C', 'E', 40)
graph.add_edge('E', 'K', 32)
graph.add_edge('K', 'H', 28)
graph.add_edge('K', 'N', 27)
graph.add_edge('K', 'Q', 62)
graph.add_edge('H', 'N', 44)
graph.add_edge('N', 'Q', 32)
graph.add_edge('N', 'G', 42)
graph.add_edge('T', 'G', 32)
graph.add_edge('R', 'T', 52)
graph.add_edge('O', 'R', 27)
graph.add_edge('L', 'O', 26)
graph.add_edge('C', 'D', 3)
```

```
graph.add_edge('I', 'L', 21)
graph.add_edge('I', 'M', 32)
graph.add_edge('J', 'M', 20)
graph.add_edge('M', 'P', 23)
graph.add_edge('H', 'J', 22)
graph.add_edge('D', 'I', 26)
graph.add_edge('F', 'L', 27)

start_node = 'S'
goal_node = 'G'
path, total_cost = graph.a_star(start_node, goal_node)
if path:
    print(" -> ".join(path))
    print("Total Cost:", total_cost)
else:
    print("No path found")
```

Output:

```
PS C:\Users\DIGGAJ\Desktop\Diggaj\College\GEC\COMP\Sem 6\AI\Practical> &
/Python312/python.exe "c:/Users/DIGGAJ/Desktop/Diggaj/College/GEC/COMP/S
S -> D -> A -> H -> N -> G
Total Cost: 179
```

Conclusion: The A* Search Algorithm was implemented and executed successfully.