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).

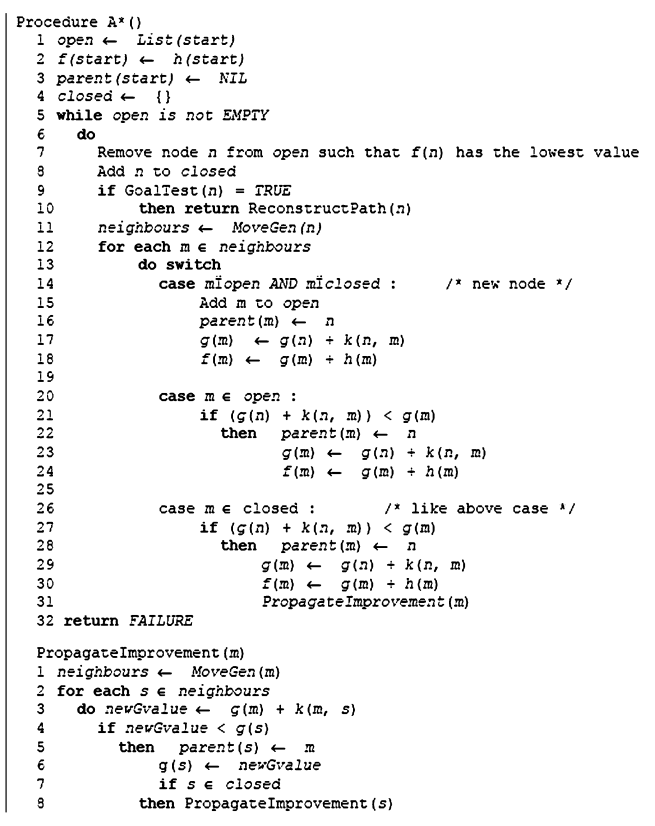
**Advantages of A\* Search:**

* 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:



**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:

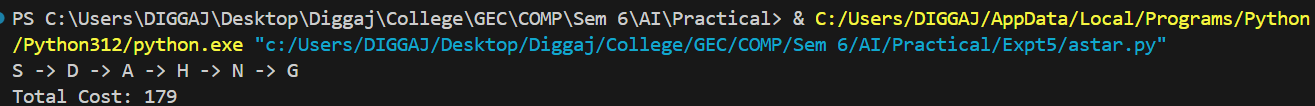
# print(" -> ".join(path))

# print("Total Cost:", total\_cost)

# else:

# print("No path found")

**Output**:



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