Experiment No.3
Study and Implementation of Informed search method: A* Search algorithm.
Date of Performance: 05/02/2025
Date of Submission: 17/02/2025

Aim: Study and Implementation of A* search algorithm.

Objective: To study the informed searching techniques and its implementation for problem solving.

Theory:

A* (pronounced as "A star") is a computer algorithm that is widely used in path finding and graph traversal. The algorithm efficiently plots a walkable path between multiple nodes, or points, on the graph. However, the A* algorithm introduces a heuristic into a regular graph-searching algorithm, essentially planning at each step so a more optimal decision is made.

A* is an extension of Dijkstra's algorithm with some characteristics of breadth-first search (BFS). Like Dijkstra, A* works by making a lowest-cost path tree from the start node to the target node. What makes A* different and better for many searches is that for each node, A* uses a function f(n)f(n)f(n) that gives an estimate of the total cost of a path using that node. Therefore, A* is a heuristic function, which differs from an algorithm in that a heuristic is more of an estimate and is not necessarily provably correct.
A* expands paths that are already less expensive by using this function:

$f(n)=g(n)+h(n)$, where

- f(n) = total estimated cost of path through node n
- g(n) = cost so far to reach node n
- h(n) = estimated cost from n to goal. This is the heuristic part of the cost function, so it is like a guess.

Pseudocode

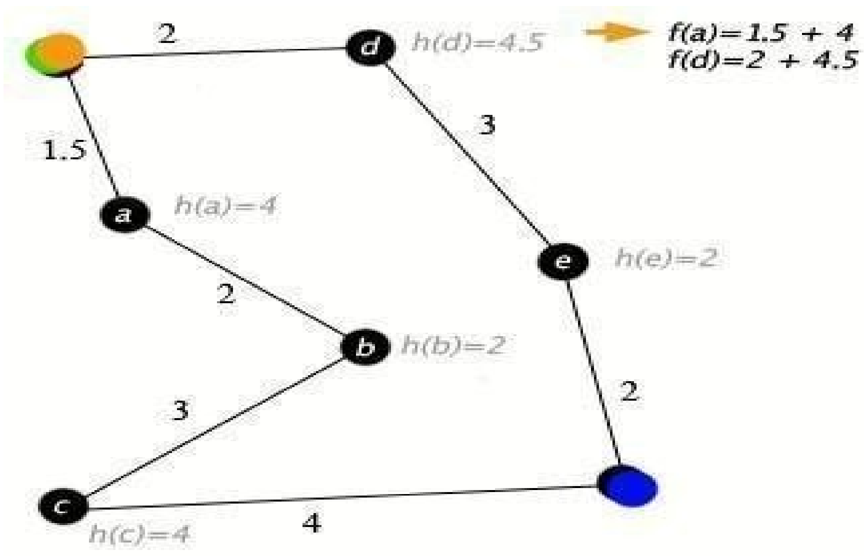
The following pseudocode describes the algorithm:

```
function reconstruct_path(cameFrom, current)
    total_path := {current} while current in
    cameFrom.Keys: current :=
    cameFrom[current]
    total_path.prepend(current) return
    total_path


// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from node n. function
A_Star(start, goal, h)
    // The set of discovered nodes that need to be (re-)expanded.
    // Initially, only the start node is known. openSet
    := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from
start to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
    gScore := map with default value of Infinity gScore[start] := 0


    // For node n, fScore[n] := gScore[n] + h(n).
    fScore := map with default value of
    Infinity fScore[start] := h(start)
```

while openSet is not empty current := the node in openSet having
the lowest fScore[] value if current = goal return
reconstruct_path(cameFrom, current)
openSet.Remove(current)
closedSet.Add(current) for
each neighbor of current if
neighbor in closedSet
continue
// d(current,neighbor) is the weight of the edge from current to neighbor //
tentative_gScore is the distance from start to the neighbor through current
tentative_gScore := gScore[current] + d(current, neighbor)
if tentative_gScore < gScore[neighbor]
// This path to neighbor is better than any previous one. Record it!
cameFrom[neighbor] := current gScore[neighbor]
:= tentative_gScore fScore[neighbor] :=
gScore[neighbor] + h(neighbor) if neighbor not in
openSet openSet.add(neighbor)

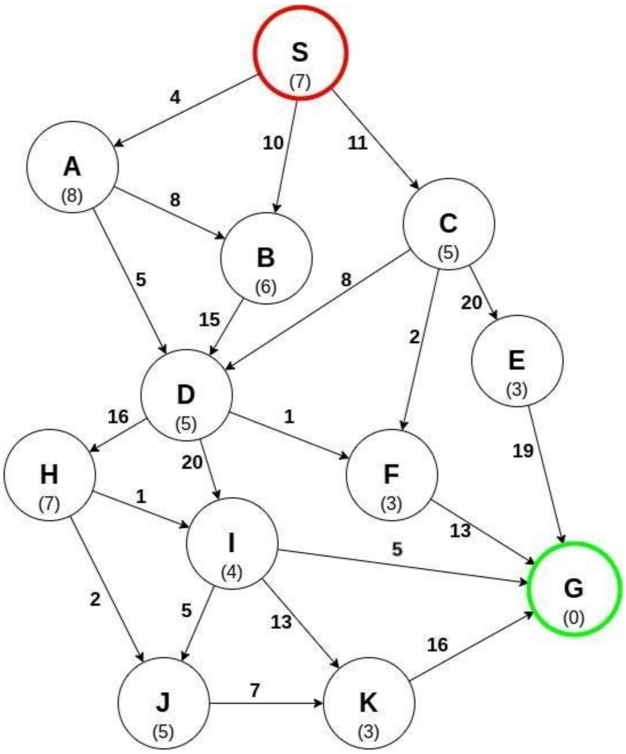
// Open set is empty but goal was never reached return
failure

An example of an A* algorithm in action where nodes are cities connected with roads and h(x) is the straight-line distance to target point:



Question 1:

Apply A* algorithm to find the path from node S to node G in graph given below. (Edge cost and hueristic values are mentioned in the graph itself.)



Solution:

Step 1:

Initialization Start at

S (7). openSet = {S}

gScore[S] = 0

fScore[S] = gScore[S] + h(S) = 0 + 7 = 7

Step 2: Expand Nodes

Expand S, add its neighbors:

A (cost 4), g(A) = 4, f(A) = 4 + 8 = 12

B (cost 10), g(B) = 10, f(B) = 10 + 6 = 16

C (cost 11), g(C) = 11, f(C) = 11 + 5 = 16 openSet = {A(12), B(16),

C(16)}

Expand A (lowest fScore = 12), add its neighbors:
D (cost 5), g(D) = 4 + 5 = 9, f(D) = 9 + 5 = 14 openSet = {D(14),

B(16), C(16)} Expand D (lowest fScore = 14), add its neighbors: F

(cost 1), g(F) = 9 + 1 = 10, f(F) = 10

+ 3 = 13

H (cost 16), g(H) = 9 + 16 = 25 (not optimal)

I (cost 20), g(I) = 9 + 20 = 29 (not optimal) openSet = {F(13), B(16),

C(16)}

Expand F (lowest fScore = 13), add its neighbor:

G (cost 13), g(G) = 10 + 13 = 23, f(G) = 23 + 0 = 23

openSet = {G(23), B(16), C(16)}

Expand G (goal reached).

Step 3: Reconstruct the Path

Backtrack from G → F → D → A → S

Optimal path: S → A → D → F → G

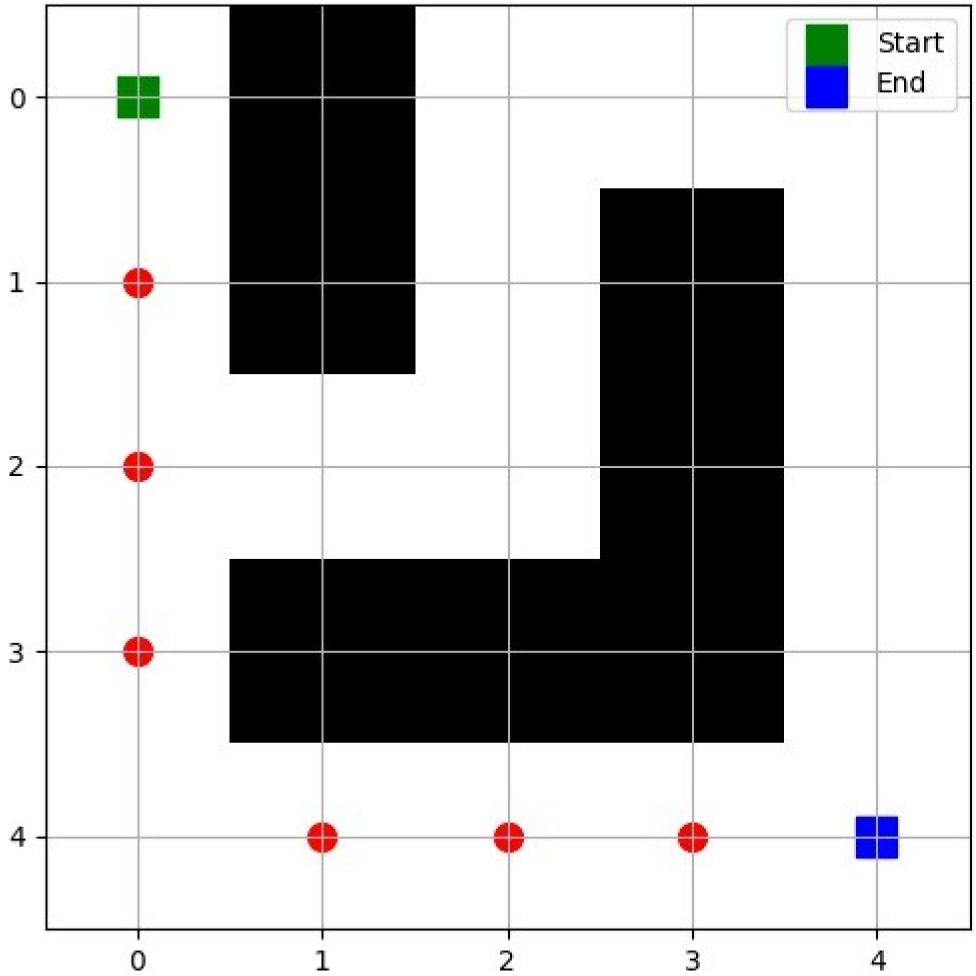
Total cost: 23
Code :

```
import heapq import
matplotlib.pyplot as plt
import numpy as np

class Node:
   def _init_(self, position, parent=None):
       self.position = position self.parent = parent self.g =
       float('inf') # Cost from start to this node self.h = 0 #
       Heuristic cost from this node to the goal
       self.f = float('inf') # Total cost

   def _lt_(self, other): return self.f
       < other.f

def heuristic(a, b): return abs(a[0] - b[0])
   + abs(a[1] - b[1])
```

```
def astar(grid, start, end, allow_diagonal=False):
    open_list = [] open_dict = {} # Dictionary for
    fast lookup closed_set = set()

    start_node = Node(start) start_node.g
    = 0
    start_node.h = heuristic(start, end) start_node.f
    = start_node.h

    heapq.heappush(open_list, start_node)
    open_dict[start] = start_node

    move_offsets = [(-1, 0), (1, 0), (0, -1), (0, 1)] if
    allow_diagonal: move_offsets += [(-1, -1), (-1, 1),
    (1, -1), (1, 1)]

    while open_list:
        current_node = heapq.heappop(open_list)
        del open_dict[current_node.position]

        if current_node.position == end:
            path = [] while
            current_node:
                path.append(current_node.position) current_node
                = current_node.parent
            return path[::-1] # Return reversed path

        closed_set.add(current_node.position)

        for dx, dy in move_offsets:
            neighbor_pos = (current_node.position[0] + dx, current_node.position[1] + dy)

            if (neighbor_pos[0] < 0 or neighbor_pos[0] >= len(grid) or
                neighbor_pos[1] < 0 or neighbor_pos[1] >= len(grid[0]) or
                grid[neighbor_pos[0]][neighbor_pos[1]] == 1 or # Obstacle
                neighbor_pos in closed_set):
                continue

            step_cost = 1.4 if allow_diagonal and dx != 0 and dy != 0 else 1
            new_g = current_node.g + step_cost
```

```
        if neighbor_pos in open_dict and open_dict[neighbor_pos].g <= new_g: continue

        neighbor_node = Node(neighbor_pos, current_node)
        neighbor_node.g = new_g neighbor_node.h =
        heuristic(neighbor_pos, end) neighbor_node.f =
        neighbor_node.g + neighbor_node.h

        heapq.heappush(open_list, neighbor_node)
        open_dict[neighbor_pos] = neighbor_node
    return None # No path found

def visualize(grid, path, start,

end):

    grid_np = np.array(grid) plt.figure(figsize=(6,6))
    plt.imshow(grid_np, cmap='Greys', origin='upper')

    for (x, y) in path:
        plt.scatter(y, x, c='red', marker='o', s=100)

    plt.scatter(start[1], start[0], c='green', marker='s', s=200, label='Start')
    plt.scatter(end[1], end[0], c='blue', marker='s', s=200, label='End')
    plt.legend() plt.grid(True) plt.show()

# Example usage
grid = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
] start = (0,
0) end = (4,
4)

path = astar(grid, start, end, allow_diagonal=True)
if path:
    print("Path:", path) visualize(grid,
    path, start, end)
else:
    print("No path found")
```

Output:



Conclusion:

The implemented A* algorithm efficiently finds the shortest path from Start (S) to Goal (G) in a grid-based environment by balancing actual travel costs (g(n)) and heuristic estimates (h(n)). Using a priority queue, it expands nodes with the lowest f-score (g(n) + h(n)) and reconstructs the path by backtracking once the goal is reached. The algorithm demonstrates optimality and efficiency when using an admissible heuristic, such as Manhattan distance, and successfully avoids obstacles while selecting the lowest-cost path. Although its performance relies on the heuristic's quality, it effectively solves shortest path problems with a time complexity of O(E log V) and space complexity of O(V).