# DEPARTMENT OF INFORMATION TECHNOLOGY

|  |  |
| --- | --- |
| **COURSE CODE: DJ19ITL504** | **DATE:15-10-24** |
| **COURSE NAME: Artificial Intelligence Laboratory** | **CLASS: TY-IT** |

**NAME: Anish Sharma ROLL NO: I011**

**EXPERIMENT NO.04**

**CO/LO:** Apply various AI approaches to knowledge intensive problem solving, reasoning, planning and uncertainty.

**AIM / OBJECTIVE:** Implement A\* search algorithm to reach goal state (Identify and analyze Informed Search Algorithm to solve the problem).

**DESCRIPTION OF EXPERIMENT:**

* Students should generate the state space for a suitable problem.
* The traversal path for A\* search should be displayed.
* Discuss the search strategy with respect to time, space complexities and completeness, optimality.

**EXPLANATION / SOLUTIONS (DESIGN):**

**Code:**

import heapq

# Define the grid with obstacles, 0 is walkable, 1 is an obstacle

grid = [

[0, 1, 0, 0, 0, 0],

[0, 1, 0, 1, 1, 0],

[0, 0, 0, 1, 0, 0],

[0, 1, 1, 1, 0, 1],

[0, 0, 0, 0, 0, 0]

]

# Define the start and goal positions

start = (0, 0) # Start node

goal = (4, 5) # Goal node

# Heuristic function: Manhattan Distance

def heuristic(node, goal):

return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

# A\* search algorithm

def astar(grid, start, goal):

rows, cols = len(grid), len(grid[0])

# Priority queue (min-heap), stores tuples of (cost, current\_node)

open\_list = []

heapq.heappush(open\_list, (0, start))

# To store the cost to reach each node from start

g\_cost = {start: 0}

# To store the parent of each node for path reconstruction

came\_from = {start: None}

while open\_list:

# Get the node with the lowest cost (f = g + h)

\_, current = heapq.heappop(open\_list)

# If we have reached the goal, reconstruct the path

if current == goal:

path = []

while current is not None:

path.append(current)

current = came\_from[current]

return path[::-1] # Return reversed path

# Explore neighbors (up, down, left, right)

for direction in [(0, 1), (1, 0), (0, -1), (-1, 0)]:

neighbor = (current[0] + direction[0], current[1] + direction[1])

# Check if neighbor is within bounds and walkable (not an obstacle)

if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols and grid[neighbor[0]][neighbor[1]] == 0:

# Calculate new g\_cost

new\_g\_cost = g\_cost[current] + 1

# If the neighbor has not been visited yet or we found a cheaper path to it

if neighbor not in g\_cost or new\_g\_cost < g\_cost[neighbor]:

g\_cost[neighbor] = new\_g\_cost

f\_cost = new\_g\_cost + heuristic(neighbor, goal)

heapq.heappush(open\_list, (f\_cost, neighbor))

came\_from[neighbor] = current

# Return None if no path is found

return None

# Run the A\* algorithm

path = astar(grid, start, goal)

# Print the output

if path:

print("Path found:", path)

else:

print("No path found")

**Output and traversal path:**

Path found: [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (4, 2), (4, 3), (4, 4), (4, 5)]

**CONCLUSION:**  In conclusion. We have learned to implement A\* Algorithm and learned about its limitation and advantages.