**ASSIGNMENT HELP**

**MANUAL**



SUBMITTED

TO

VISHWAKARMA INSTITUTE OF INFORMATION TECHNOLOGY, PUNE

FOR THE SKILL AND COMPETENCY EVALUATION OF

ARTIFICIAL INTELLIGENCE [CAUA31201]

IN

**CSE AI DEPARTMENT**

BY

**Vedant Rakesh Mukhekar**

**Class: T.Y. BTech Division: A Batch: A2**

**Batch Teacher**

**Dr. ANURADHA YENKIKAR.**

**INDEX**

|  |  |  |
| --- | --- | --- |
| **SR. NO.** | **CONTENTS** | **PAGE NO.** |
| **1** | **PROBLEM STATEMENT** | **4-5** |
| **2** | **LIBRARY USED** | **5-6** |
| **3** | **THEORY** | **6-7** |
| **4** | **METHODOLOGY** | **7-14** |
| **5** | **ADVANTAGES & DISADVANTAGES** | **14-15** |
| **6** | **WORKING** | **16-17** |
| **7** | **DIAGRAM** | **17-18** |
| **8** | **CONCLUSION** | **18-19** |

### ****Problem Statement****

The goal is to implement the **A\*** algorithm, a pathfinding and graph traversal algorithm, to find the shortest path between two points in a weighted grid or graph. This algorithm is widely used in applications such as robotics, navigation systems, and games. In this implementation, we will solve a pathfinding problem on a grid with obstacles, where the agent has to navigate from a start position to a goal position while minimizing the cost of movement.

### ****Libraries Used****

* **Python Standard Libraries**:
  + heapq: To implement the priority queue (open list) for selecting the node with the lowest cost.
  + math: For calculating heuristic distances if needed.

Optional libraries for visualization:

* matplotlib: For visualizing the grid and the path found by the algorithm.

### ****Theory****

The **A\*** algorithm is an informed search algorithm used to find the shortest path in a graph. It combines two metrics:

1. **g(n)**: The cost of the path from the start node to node n.
2. **h(n)**: The heuristic estimate of the cost from node n to the goal.

The algorithm minimizes the sum of both to get the total cost:

f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n)

* **g(n)** is the known cost from the start node to the current node.
* **h(n)** is an estimate of the remaining cost, often calculated using Manhattan or Euclidean distance.
* **f(n)** is the total estimated cost for the path through the node n.

The algorithm repeatedly selects the node with the lowest **f(n)** value from the open list (nodes to be evaluated), expands it, and adds its neighbors to the open list until the goal is reached.

### ****Methodology****

1. **Problem Setup**:
   * A grid or graph is defined where each cell/node represents a position, and edges represent movement between positions.
   * Some cells are marked as obstacles, which the agent cannot traverse.
2. **Heuristic Function**:
   * The Manhattan distance heuristic is commonly used in grid-based pathfinding when diagonal movement is not allowed. This heuristic estimates the distance between two points by summing the absolute differences of their coordinates.
3. **A\*** Algorithm Process:
   * The algorithm starts at the given start position, calculates **f(n) = g(n) + h(n)** for all neighboring nodes, and adds valid nodes to the open list.
   * The node with the lowest **f(n)** is selected, and the process continues until the goal node is reached or the open list is empty (indicating no path exists).
4. **Pathfinding**:
   * The algorithm keeps track of the parent node for each node to reconstruct the path from the goal to the start.

### ****Advantages & Disadvantages****

* **Advantages**:
  + **Optimal Path**: A\* guarantees the shortest path as long as the heuristic is admissible.
  + **Flexible**: Can be adapted to various types of grids or graphs by changing the heuristic.
  + **Informed Search**: It is faster than uninformed algorithms like Dijkstra's because it uses the heuristic to prioritize paths.
* **Disadvantages**:
  + **Memory Usage**: It requires storing all open and closed nodes, which can be memory-intensive for large graphs.
  + **Performance**: The choice of heuristic directly affects the performance of the algorithm. A poorly chosen heuristic can lead to unnecessary exploration.

### ****Working Example (Python Code)****

python

Copy code

import heapq

class Node:

def \_\_init\_\_(self, position, parent=None):

self.position = position # (x, y)

self.parent = parent

self.g = 0 # Cost from start to current node

self.h = 0 # Heuristic cost to goal

self.f = 0 # Total cost

def \_\_eq\_\_(self, other):

return self.position == other.position

def \_\_lt\_\_(self, other):

return self.f < other.f

# Heuristic function: Manhattan Distance

def heuristic(current, goal):

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

# A\* algorithm

def astar(grid, start, goal):

start\_node = Node(start)

goal\_node = Node(goal)

open\_list = []

closed\_list = set()

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

while open\_list:

current\_node = heapq.heappop(open\_list)

if current\_node == goal\_node:

path = []

while current\_node is not None:

path.append(current\_node.position)

current\_node = current\_node.parent

return path[::-1]

closed\_list.add(current\_node.position)

# Define possible movements (up, down, left, right)

neighbors = [(0, -1), (0, 1), (-1, 0), (1, 0)]

for offset in neighbors:

neighbor\_pos = (current\_node.position[0] + offset[0], current\_node.position[1] + offset[1])

if neighbor\_pos[0] < 0 or neighbor\_pos[0] >= len(grid) or neighbor\_pos[1] < 0 or neighbor\_pos[1] >= len(grid[0]):

continue

if grid[neighbor\_pos[0]][neighbor\_pos[1]] == 1: # Obstacle

continue

neighbor = Node(neighbor\_pos, current\_node)

if neighbor.position in closed\_list:

continue

neighbor.g = current\_node.g + 1

neighbor.h = heuristic(neighbor.position, goal\_node.position)

neighbor.f = neighbor.g + neighbor.h

if any(open\_node for open\_node in open\_list if neighbor == open\_node and neighbor.g > open\_node.g):

continue

heapq.heappush(open\_list, neighbor)

return None

# Example grid (0 - empty space, 1 - obstacle)

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]

]

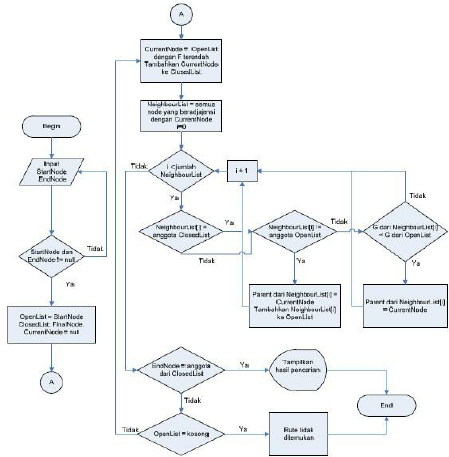
start = (0, 0)

goal = (4, 4)

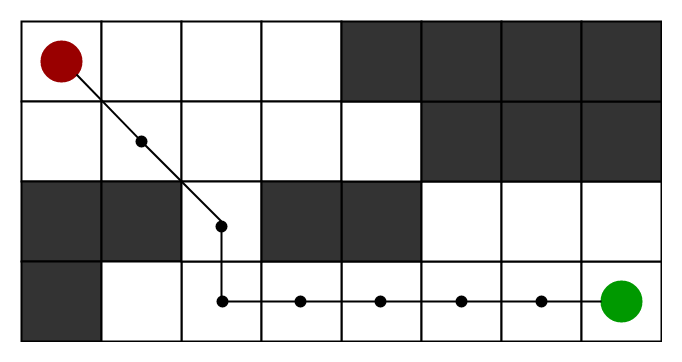
path = astar(grid, start, goal)

print("Path:", path)

### ****Diagram****



APLICATION OF THE PROGRAM



### ****Conclusion****

The **A\*** algorithm is a powerful and widely used pathfinding algorithm that efficiently finds the shortest path in grid-based or graph-based environments. Its combination of actual cost and heuristic makes it more efficient than uninformed search algorithms. In this implementation, the algorithm was used to solve a grid-based pathfinding problem, where it successfully navigated from the start to the goal while avoiding obstacles. This algorithm is widely applicable in domains such as robotics, game development, and route optimization.