**Path Finding Using A\* Algorithm**

**PROJECT REPORT**

**Session (2024-25)**

**Submitted by**

**Alok Kumar Singh**

**202410116100019**

**Ansh Raj**

**202410116100031**

**Aman Nayak**

**202410116100021**

**Submitted in partial fulfillment of the**

**Requirements for the Degree of**

**MASTER OF COMPUTER APPLICATION**

**Under the Supervision of**

**Mr. Apoorv Jain**

### Assistant Professor



**Submitted to**

**Department Of Computer Applications**

**KIET Group of Institutions, Ghaziabad**

**Uttar Pradesh-201206**

1. **Introduction**

**1.1 Background :**

1.1.1 Pathfinding in Robotics: Autonomous robots rely heavily on pathfinding to navigate environments, avoid obstacles, and reach desired destinations efficiently. This has applications in industrial automation, exploration, and search-and-rescue scenarios.

1.1.2 Pathfinding in Game Development: Games often require pathfinding for AI-controlled characters (NPCs) to move intelligently, react to player actions, and create realistic gameplay experiences. It's crucial for navigation, enemy AI, and strategic planning within games.

1.1.3 Route Planning and Logistics: Optimization of delivery routes, transportation networks, and logistics operations depends on efficient pathfinding algorithms to minimize travel time, fuel consumption, and overall costs.

1.1.4 Geographical Information Systems (GIS): Pathfinding plays a vital role in GIS applications, such as finding shortest routes on maps, determining travel directions, and analyzing spatial data for transportation planning and infrastructure development.

**1.2 A\* Algorithm :**

1.2.1 Efficiency and Optimality: A\* is favored for its ability to find the shortest path in many scenarios while exploring fewer nodes compared to uninformed search algorithms like breadth-first search. Its heuristic guidance helps prioritize promising paths.

1.2.1 Heuristic Function: The heuristic function estimates the cost from a given node to the goal. This estimation guides the search process, making it more efficient than algorithms that only consider the cost from the start node.

1.2.1 Admissibility and Consistency: For A\* to guarantee finding the shortest path, the heuristic function must be admissible (never overestimating the actual cost to the goal) and consistent (satisfying the triangle inequality).

**1.3 Objectives :**

1.3.1 Implementation: This project focuses on implementing the A\* algorithm in Python, demonstrating a practical understanding of its core principles and steps.

1.3.2 Visualization: A key objective is to visualize the search process, providing a clear and intuitive representation of how the algorithm explores the grid and identifies the optimal path. This enhances understanding and allows for analysis of the algorithm's behavior.

1.3.3 User Interaction: The project aims to create an interactive experience, allowing users to input grid parameters, obstacle configurations, and start/goal positions, making it adaptable to different scenarios.

**2. Methodology**

**2.1 Data Structure:**

* Grid Representation: The environment is modeled using a 2D grid, where each cell represents a node. This grid structure simplifies navigation and allows for easy representation of obstacles and traversable areas.
* Grid Initialization: The code allows users to specify the grid dimensions (rows and columns) and obstacle density. Obstacles are randomly placed on the grid based on the specified density, creating a dynamic environment.

**2.2 Node Representation :**

* Node Class: A Node class encapsulates the properties of each node in the grid:
* position: A tuple representing the (x, y) coordinates of the node on the grid.
* g: The cost of reaching the node from the start node.
* h: The heuristic estimate of the cost from the node to the goal node.
* f: The total cost, calculated as f = g + h.
* parent: A reference to the parent node, used for path reconstruction.

**2.3 Heuristic Function :**

* Manhattan Distance: The heuristic function used in this project is the Manhattan distance, which is calculated as the sum of the absolute differences in the x and y coordinates between two nodes.
* Admissibility: The Manhattan distance is admissible for grid-based environments with 4-directional movement, as it never overestimates the actual distance to the goal.

**2.4 Open and Closed Lists :**

* Open List (Priority Queue): The open list is implemented using a priority queue (heapq in Python), which ensures that the node with the lowest total cost (f) is always selected for exploration.
* Closed List (Set): The closed list is implemented using a set, which provides efficient checking for whether a node has already been visited.

**2.5 Algorithm Steps (Detailed) :**

1. Initialization :

* Create a Node object for the start position and the goal position.
* Initialize the open list with the start node.
* Initialize the closed list as an empty set.

1. Iteration:

* While the open list is not empty:
* Get the node with the lowest total cost (f) from the open list (using heapq.heappop). This is the current node.
* If the current node's position matches the goal position, the path has been found. Proceed to path reconstruction.
* Add the current node to the closed list.
* Generate neighboring nodes (adjacent cells) of the current node.
* For each neighbor:
* If the neighbor is within the grid boundaries, not an obstacle, and not in the closed list:
* Calculate the cost to reach the neighbor (g) by adding 1 to the current node's g value (assuming a cost of 1 per movement).
* Calculate the heuristic estimate (h) using the Manhattan distance between the neighbor and the goal node.
* Calculate the total cost (f) as f = g + h.
* If the neighbor is not in the open list, or if it is in the open list but with a higher f value:
* Create a new Node object for the neighbor with the calculated g, h, f, and set its parent to the current node.
* Add the neighbor to the open list (using heapq.heappush).

1. Path Reconstruction:

* If the goal node is found:
* Start with the goal node.
* Backtrack through the parent pointers of each node until the start node is reached.
* Store the positions of each node encountered during backtracking in a list.
* Reverse the list to obtain the path from the start to the goal.

**2.6 Visualization :**

* draw\_grid Function : This function uses matplotlib to create a visual representation of the grid, obstacles, start and goal points, and the path found by the A\* algorithm.
* Color Coding : The grid is displayed with different colors to represent:
* Obstacles (black or dark color)
* Start point (green)
* Goal point (red)
* Path (blue or another distinct color)
* Grid Display : The plt.imshow function is used to display the grid as an image.
* Markers : plt.scatter is used to mark the start and goal points.
* Legend and Title : A legend and title are added to the plot for clarity.

**3. Code Type**

**3.1 Programming Language :**

* Python: The project is implemented in Python due to its readability, extensive libraries for data science and algorithm development, and suitability for interactive experimentation.

**3.2 Libraries :**

* heapq: Provides functions for working with heaps, which are used to implement the priority queue for the open list.
* matplotlib: A powerful library for creating static, interactive, and animated visualizations in Python. It's used to visualize the grid and path.
* numpy: A fundamental library for numerical computing in Python, providing support for arrays and matrices. It's used to represent the grid and perform array operations.
* random: This module provides functions for generating random numbers, which are used to randomly place obstacles on the grid.

**3.3 Code Structure :**

* Functions : The code is organized into functions to improve modularity and readability :
* Node: A class representing a node in the grid.
* heuristic: Calculates the heuristic distance between two nodes.
* astar: Implements the main A\* algorithm logic.
* draw\_grid: Visualizes the grid and path.
* User Input : The code prompts the user to enter grid dimensions, obstacle density, start coordinates, and goal coordinates.
* Main Execution: After receiving user input, the code initializes the grid, runs the A\* algorithm, and displays the results using draw\_grid.

1. **Screen Shots :**

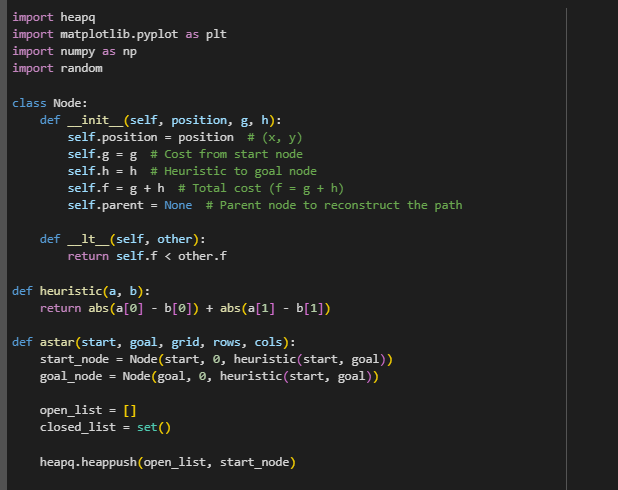


Figure 4.1

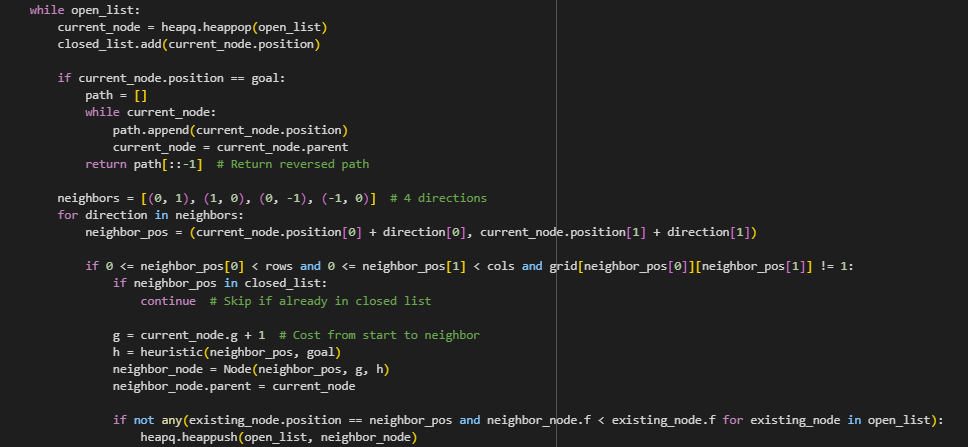


Figure 4.2

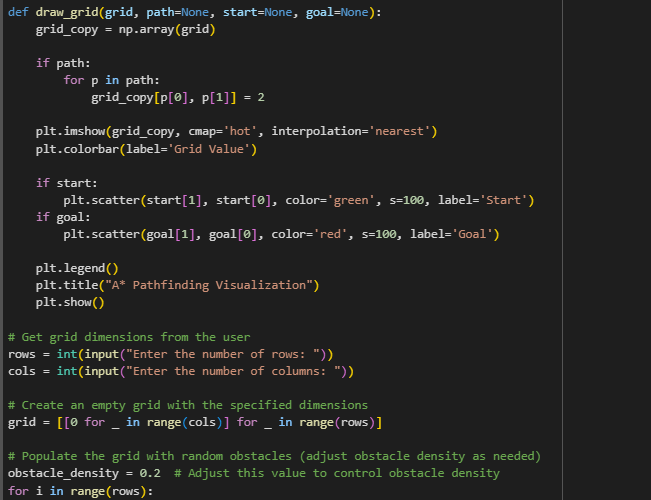


Figure 4.3

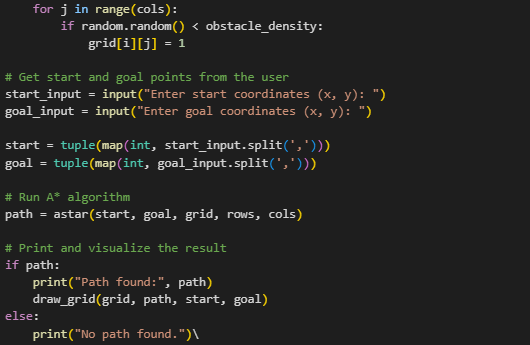


Figure 4.4

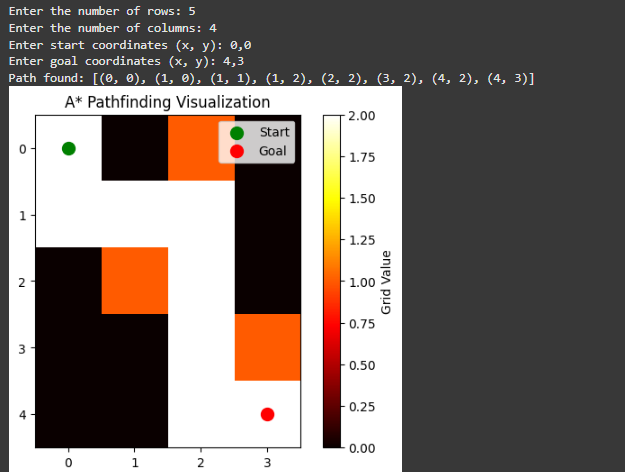


Figure 4.5