

Pathfinding with A\* Algorithm

Name: Shashank Raj

Roll No.: 2428CSEAI1767

Course: Introduction to AI

Date: 11 March 2025

**Introduction**

Pathfinding is a fundamental problem in computer science, used in applications like robotics, gaming, and navigation systems. The goal of pathfinding is to determine the shortest or most efficient path from a starting point to a destination while avoiding obstacles.

The *A (A-Star) Algorithm*\* is a widely used search algorithm for pathfinding. It combines the benefits of **Dijkstra’s Algorithm** (which guarantees the shortest path) and the **Greedy Best-First Search** (which uses heuristics for faster searching). A\* uses two key values:

1. **G-cost** – The actual distance from the start node to the current node.
2. **H-cost (heuristic)** – The estimated distance from the current node to the goal.

By adding these two values (**F = G + H**), A\* prioritizes paths that are both short and efficient, making it an optimal and fast algorithm for grid-based navigation problems.

This report explains the implementation of the A\* algorithm in Python for finding the shortest path in a grid-based environment with obstacles.

**Methodology**

To solve the pathfinding problem, we implemented the *A (A-Star) Algorithm*\* in Python. The approach follows these steps:

1. **Grid Representation**
   * The environment is represented as a **2D grid** where:
     + 0 represents walkable areas.
     + 1 represents obstacles that cannot be crossed.
2. **Heuristic Function (H-cost)**
   * We use the **Manhattan Distance** as the heuristic function: H(x,y)=∣x1−x2∣+∣y1−y2∣H(x, y) = |x\_1 - x\_2| + |y\_1 - y\_2|H(x,y)=∣x1​−x2​∣+∣y1​−y2​∣
   * This estimates the distance between a node and the goal, guiding the search efficiently.
3. **Priority Queue (Min-Heap)**
   * We use a **priority queue** (Min-Heap) to always explore the most promising node first, ensuring an efficient search.
4. **Path Exploration**
   * The algorithm starts from the **initial position** and explores possible moves (up, down, left, right).
   * It avoids obstacles and keeps track of the shortest path to each position using the **G-cost (actual movement cost)**.
   * The node with the lowest **F-cost (G + H)** is expanded first.
5. **Path Reconstruction**
   * Once the goal is reached, the algorithm backtracks using a dictionary (came\_from) to reconstruct the shortest path.
6. **Handling No Path Cases**
   * If no valid path is found, the algorithm returns None, indicating that the goal is unreachable.

This structured approach ensures that the shortest path is found optimally while maintaining efficiency in grid-based environments.

**Code**

import heapq  # Importing heapq for priority queue operations

# Function to calculate the heuristic (Manhattan Distance) between two points

def heuristic(a, b):

    return abs(a[0] - b[0]) + abs(a[1] - b[1])

# A\* Algorithm implementation

def astar(grid, start, goal):

    open\_set = [(0, start)]  # Priority queue initialized with start node

    came\_from, g\_score = {}, {start: 0}  # Tracking paths and cost from start

    while open\_set:

        \_, current = heapq.heappop(open\_set)  # Get node with lowest f-score

        if current == goal:  # If goal is reached, reconstruct path

            path = []

            while current in came\_from:

                path.append(current)

                current = came\_from[current]

            return path[::-1] + [start]  # Return reversed path including start

        # Possible movement directions (Right, Down, Left, Up)

        for dx, dy in [(0,1), (1,0), (0,-1), (-1,0)]:

            neighbor = (current[0] + dx, current[1] + dy)

            # Check if neighbor is within grid bounds and is walkable (0 = open, 1 = obstacle)

            if neighbor in g\_score or not (0 <= neighbor[0] < len(grid) and 0 <= neighbor[1] < len(grid[0]) and grid[neighbor[0]][neighbor[1]] == 0):

                continue  # Skip if already visited or not walkable

            came\_from[neighbor] = current  # Store the path

            g\_score[neighbor] = g\_score[current] + 1  # Update cost from start

            # Push neighbor into priority queue with its f-score (g + h)

            heapq.heappush(open\_set, (g\_score[neighbor] + heuristic(neighbor, goal), neighbor))

    return None  # No path found

# Grid representation (0 = walkable, 1 = obstacle)

grid = [

    [0, 1, 0, 0, 0],

    [0, 1, 0, 1, 0],

    [0, 0, 0, 1, 0],

    [0, 1, 1, 1, 0],

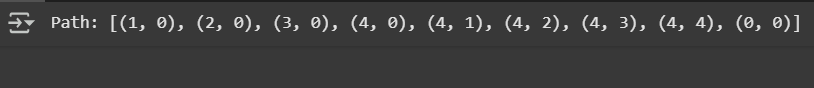
    [0, 0, 0, 0, 0]

]

# Running the A\* algorithm from (0,0) to (4,4) and printing the path

print("Path:", astar(grid, (0, 0), (4, 4)))

**Output**

****

**References**

 **Internet**

 **AI agents : ChatGPT,Gemini**

 **Libraries Used:**

* Python Standard Library (heapq for priority queue management).