Pathfinding with A* Algorithm

Prepared by: Vanshika Arora

Univ. R.No.202401100400208

Date: 10/March/2025

Purpose: This report details the implementation of pathfinding with A* Algorithm using Python.

Introduction

The *A (A-star) algorithm** is a popular and efficient pathfinding algorithm used to find the shortest path between two points in a grid, often avoiding obstacles. It is widely used in robotics, gaming, and AI systems due to its ability to balance between **optimality** and **efficiency**.

The A* algorithm works by combining **Dijkstra's algorithm** for pathfinding and a heuristic to guide the search towards the goal. It uses a function f(n) = g(n) + h(n) where:

- **g(n)** is the cost to reach the current node from the start node.
- **h(n)** is the heuristic estimate of the cost from the current node to the goal.

The main goal of this project is to implement the A* algorithm for pathfinding in a 2D grid, where the user can specify obstacles, a start point, and a goal point. The algorithm will then compute the shortest possible path while avoiding obstacles, providing a visual representation of the pathfinding process.

Methodology

The approach focused on:

1. **Input Handling**: Users define grid size, obstacles, start, and goal positions.
2. *A* Algorithm*: The core of the system, using Manhattan distance as a heuristic to efficiently find the shortest path.
3. **Path Reconstruction**: Backtracking from the goal to the start to determine the shortest path.
4. **Visualization**: Displaying the grid with obstacles and the computed path.

Implementation Steps

1. **Grid Setup and Input**:
   - User provides grid dimensions and obstacle positions.
   - Start and goal positions are also defined by the user.
2. *A* Algorithm Execution*:
   - The algorithm calculates f, g, and h values for each node.
   - Nodes are expanded based on the lowest f value, avoiding obstacles and out-of-bounds areas.
3. **Pathfinding**:
   - The algorithm expands nodes until the goal is found or the Open List is empty (indicating no path).

    - If the goal is reached, the path is reconstructed by backtracking from the goal node.
4. **Grid Visualization**:
    - The final grid is displayed with P for the path, X for obstacles, and . for empty spaces.
5. **Edge Case Handling**:
    - The system checks for invalid grid sizes, blocked start/goal positions, and ensures no path is found if the Open List is empty.

CODE TYPED

```
import heapq
# Directions (Right, Down, Left, Up)
DIRECTIONS = [(0, 1), (1, 0), (0, -1), (-1, 0)]

# Heuristic function (Manhattan Distance)
def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

# A* Algorithm
def a_star(grid, start, goal):
    open_list = []
    heapq.heappush(open_list, (0 + heuristic(start, goal), 0, start))  # (f, g, node)
    came_from = {}  # To reconstruct the path
    g_score = {start: 0}
    f_score = {start: heuristic(start, goal)}

    while open_list:
        _, g, current = heapq.heappop(open_list)

        if current == goal:
            # Reconstruct path
            path = []
            while current in came_from:
```

```
            path.append(current)

            current = came_from[current]

        path.append(start)

        return path[::-1]  # Return path from start to goal


    for direction in DIRECTIONS:

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


        # Check bounds and if the neighbor is not an obstacle

        if 0 <= neighbor[0] < len(grid) and 0 <= neighbor[1] < len(grid[0]) and
grid[neighbor[0]][neighbor[1]] != 1:

            tentative_g_score = g + 1  # Cost to reach the neighbor


            # If this path is better, update the scores

            if neighbor not in g_score or tentative_g_score < g_score[neighbor]:

                came_from[neighbor] = current

                g_score[neighbor] = tentative_g_score

                f_score[neighbor] = tentative_g_score + heuristic(neighbor, goal)

                heapq.heappush(open_list, (f_score[neighbor], tentative_g_score, neighbor))


    return None  # If no path exists


# Function to display the grid and path

def display_grid(grid, path):

    for i in range(len(grid)):
```

```
        for j in range(len(grid[i])):
            if (i, j) in path:
                print('P', end=' ')  # Mark path with 'P'
            elif grid[i][j] == 1:
                print('X', end=' ')  # Mark obstacles with 'X'
            else:
                print('.', end=' ')  # Empty spaces
        print()

# Function to get the grid input from the user (1-based indexing)
def get_grid_input():
    rows = int(input("Enter number of rows in the grid (1-based): "))
    cols = int(input("Enter number of columns in the grid (1-based): "))

    grid = [[0 for _ in range(cols)] for _ in range(rows)]  # 0 represents empty spaces

    print("Enter the positions of obstacles (row, column) as comma-separated values.")
    print("Enter 'done' when finished.")
    while True:
        obstacle_input = input("Enter obstacle position (row,col) or 'done': ")
        if obstacle_input.lower() == 'done':
            break
        try:
            row, col = map(int, obstacle_input.split(','))
            if 1 <= row <= rows and 1 <= col <= cols:
```

```
            grid[row - 1][col - 1] = 1  # Mark obstacle
          else:
            print("Invalid position. Out of bounds.")
        except ValueError:
          print("Invalid input. Please enter in 'row,col' format.")

    while True:
        try:
          start_row, start_col = map(int, input("Enter start position (row,col) as 1-based: ").split(','))
          if 1 <= start_row <= rows and 1 <= start_col <= cols and grid[start_row - 1][start_col - 1] != 1:
            break
          else:
            print("Invalid start position. Ensure it's within bounds and not an obstacle.")
        except ValueError:
          print("Invalid input. Please enter in 'row,col' format.")

    while True:
        try:
          goal_row, goal_col = map(int, input("Enter goal position (row,col) as 1-based: ").split(','))
          if 1 <= goal_row <= rows and 1 <= goal_col <= cols and grid[goal_row - 1][goal_col - 1] != 1:
            break
          else:
            print("Invalid goal position. Ensure it's within bounds and not an obstacle.")
```

```
        except ValueError:
            print("Invalid input. Please enter in 'row,col' format.")

    return grid, (start_row - 1, start_col - 1), (goal_row - 1, goal_col - 1)

# Main code to run A* algorithm
grid, start, goal = get_grid_input()

# Run A* algorithm
path = a_star(grid, start, goal)

# Display the grid and the path if one was found
if path:
    print("\nPath found:")
    display_grid(grid, path)
else:
    print("\nNo path found.")
```

SCREENSHOTS OUTPUT

```
Enter number of rows in the grid: 3
Enter number of columns in the grid: 3
Enter obstacle position (row,col) or 'done': 2,2
Enter obstacle position (row,col) or 'done': done
Enter start position (row,col): 1,1
Enter goal position (row,col): 3,3

Path found:
P . .
X P .
. . P
```

```
Enter number of rows in the grid (1-based): 3
Enter number of columns in the grid (1-based): 3
Enter the positions of obstacles (row, column) as comma-separated values.
Enter 'done' when finished.
Enter obstacle position (row,col) or 'done': 1,1
Enter obstacle position (row,col) or 'done': 2,
Invalid input. Please enter in 'row,col' format.
Enter obstacle position (row,col) or 'done': 2,3
Enter obstacle position (row,col) or 'done': 3,3
Enter obstacle position (row,col) or 'done': done
Enter start position (row,col) as 1-based: 2,1
Enter goal position (row,col) as 1-based: 3,2

Path found:
X . .
P P X
. P X
```

CONCLUSION

The A* Pathfinding algorithm successfully computes the shortest path in a grid while avoiding obstacles using the Manhattan distance heuristic. This implementation demonstrates the algorithm's efficiency and accuracy in grid-based scenarios, making it suitable for applications like robotics and game development.

Future Enhancements

1. **Diagonal Movement**: Support for diagonal movement to reduce path length.
2. **Dynamic Obstacles**: Handle obstacles that appear or disappear during runtime.
3. **Optimizations**: Improve performance for larger grids with better data structures.
4. **User Interface**: Develop a GUI for better user interaction and visualization of pathfinding.

References

1. *A Algorithm - Wikipedia**: https://en.wikipedia.org/wiki/A*_search_algorithm
2. **Artificial Intelligence: A Modern Approach** by Stuart Russell and Peter Norvig
3. *GeeksforGeeks - A Search Algorithm**: https://www.geeksforgeeks.org/a-search-algorithm/