Cover

| ARTIFICIAL INTELLIGENCE REPORT |  |
| --- | --- |
| A\* PATH FINDING SCRIPT | |
| Name - Aviral  Branch - CSE AI  University Roll no. - 202401100300080 | |

Intro and objectives

| A\* Pathfinding script | |
| --- | --- |

| 01 |
| --- |

### Introduction

#### Purpose

Demonstrate the functionality of the A\* algorithm in finding the shortest path.

#### Scope

Implementing the A\* algorithm in Python using a grid-based system

#### Importance

The A\* algorithm ensures that the path found is the shortest and most efficient, making it ideal for real-time applications.

|  | | |
| --- | --- | --- |

Methodology

# Methodology

#### 1. Grid Definition

* The grid is defined using a 2D list where:
  + 0 represents walkable spaces.
  + 1 represents obstacles that cannot be crossed.

#### 2. Node Class

Each cell in the grid is treated as a node. Each node contains:

* **Position:** The cell's coordinates in the form (x, y).
* **g-score:** The cost of moving from the starting point to the current node.
* **h-score (heuristic):** The estimated cost to reach the goal using Manhattan distance.
* **f-score:** The total estimated cost calculated as f = g + h.
* **Parent Node:** Tracks the node's predecessor to reconstruct the path.

#### 3. Heuristic Function

The heuristic function calculates the Manhattan distance between two points, which is defined as:

|x1 - x2| + |y1 - y2|

This heuristic is ideal for grid-based movement with only horizontal and vertical steps allowed.

#### 4. Algorithm Process

* The starting node is initialized with zero cost and pushed into the priority queue.
* The algorithm explores neighboring cells in four directions: Up, Down, Left, and Right.
* Each neighbor's cost is calculated, and if a node with a lower cost already exists in the queue, it is skipped.
* The algorithm terminates when the endpoint is reached or if no valid path exists.

#### 5. User Interaction

The user is prompted to:

* View the grid layout.
* Enter valid starting and ending coordinates (walkable points only).

CODE

# CODE

"""

pathFinding with A\* Algorithm

This script implements the A\* (A-star) algorithm for finding the shortest path

between two points on a grid. It uses a priority queue (heap) for efficient node management.

Author: [Your Name]

Date: [Current Date]

"""

*import* heapq *# For priority queue functionality*

*# Node class to store information about each cell in the grid*

class Node:

def **\_\_init\_\_**(*self*, *position*, *g*=0, *h*=0):

*self*.position = *position* *# (x, y) coordinates of the node*

*self*.g = *g* *# Cost from start node*

*self*.h = *h* *# Heuristic (estimated cost to goal)*

*self*.f = *g* + *h* *# Total cost (g + h)*

*self*.parent = None *# Reference to the previous node for backtracking the path*

*# For priority queue comparison (lower 'f' values get higher priority)*

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

*return* *self*.f < *other*.f

*# Heuristic function using Manhattan distance (best for grid-based movement)*

def **heuristic**(*a*, *b*):

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

*# A\* Algorithm implementation*

def **a\_star**(*grid*, *start*, *end*):

open\_list = [] *# Priority queue for nodes to explore*

closed\_set = set() *# Set for visited nodes*

*# Initialize the starting node with zero cost and add it to the open list*

start\_node = Node(*start*, 0, heuristic(*start*, *end*))

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

*# Main loop to explore nodes until the goal is reached or no path exists*

*while* open\_list:

current\_node = heapq.heappop(open\_list) *# Node with lowest f value*

*# Goal check - path found*

*if* current\_node.position == *end*:

path = []

*while* current\_node:

path.append(current\_node.position)

current\_node = current\_node.parent

*return* path[::-1] *# Return path in correct order*

*# Mark the current node as visited*

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

*# Explore neighboring nodes (up, down, left, right)*

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

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

*# Boundary and obstacle check*

*if* (0 <= neighbor\_pos[0] < len(*grid*) and

0 <= neighbor\_pos[1] < len(*grid*[0]) and

*grid*[neighbor\_pos[0]][neighbor\_pos[1]] == 0 and

neighbor\_pos not in closed\_set):

*# Calculate the cost for the neighbor*

g\_score = current\_node.g + 1

neighbor\_node = Node(neighbor\_pos, g\_score, heuristic(neighbor\_pos, *end*))

neighbor\_node.parent = current\_node *# Track the path*

*# Avoid adding duplicate nodes with higher costs*

*if* any(node.position == neighbor\_pos *and* node.f <= neighbor\_node.f *for* node *in* open\_list):

*continue*

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

*# If no path is found*

*return* None

*# Example grid (0 = walkable, 1 = obstacle)*

grid = [

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

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

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

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

[0, 0, 0, 1, 0]

]

*# Define start and end points for pathfinding*

def **get\_valid\_coordinates**(*prompt*, *grid*):

*while* True:

*try*:

x, y = map(int, input(*prompt*).split())

*if* 0 <= x < len(*grid*) and 0 <= y < len(*grid*[0]):

*if* *grid*[x][y] == 1:

print("That position contains an obstacle. Please choose a walkable position.")

*continue*

*return* (x, y)

*else*:

print(f"Coordinates must be between (0,0) and ({len(*grid*)-1},{len(*grid*[0])-1})")

*except* ValueError:

print("Please enter two numbers separated by a space (e.g., '2 3')")

*# Get start and end points from user input*

print("\nGrid layout (0 = walkable, 1 = obstacle):")

*for* row *in* grid:

print(row)

print("\nEnter coordinates as 'x y' (separated by space)")

start = get\_valid\_coordinates("Enter start position: ", grid)

end = get\_valid\_coordinates("Enter end position: ", grid)

*# Execute A\* algorithm and display results*

path = a\_star(grid, start, end)

*if* path:

print("Path found:", path)

*else*:

print("No path found")

Screenshots of output

# Screenshots of output

#### **Sample Grid Layout:**

#### 

#### 

#### **Input Example:**

#### 

#### **Output Example:**

#### 

#### **No Path Example:**

#### 

Conclusion

# Conclusion

The A\* algorithm is an efficient and reliable method for pathfinding. This implementation successfully demonstrates its functionality using a Python script. The project efficiently handles user-defined inputs, checks for valid coordinates, and accurately computes the optimal path.

Potential improvements could include:

* Adding diagonal movement support.
* Visual representation of the grid and the path.
* Dynamic grid resizing and customization.

This project effectively illustrates the fundamental concepts of the A\* algorithm, making it an excellent starting point for learning about pathfinding techniques in computer science.