

|  |
| --- |
| **RAJALAKSHMI INSTITUTE OF TECHNOLOGY** |
| (An Autonomous Institution, Affiliated to Anna University, Chennai) |

**DEPARTMENT OF CSE (ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)**

**ACADEMIC YEAR 2025 - 2026**

**SEMESTER III**

**ARTIFICIAL INTELLIGENCE LABORATORY**

**MINI PROJECT REPORT**

|  |  |
| --- | --- |
| **REGISTER NUMBER** | 2117240030039 |
| **NAME** | HARINI K |
| **PROJECT TITLE** | MAZE SOLVING USING A\* ALGORITHM |
| **DATE OF SUBMISSION** | 04.11.2025 |
| **FACULTY IN-CHARGE** | **Mrs. M. Divya** |

**Signature of Faculty In-charge**

**INTRODUCTION**

* **Artificial Intelligence (AI)** focuses on enabling machines to make intelligent decisions. One of the key applications of AI is **pathfinding**, which deals with finding an optimal route from a start point to a goal within a given environment.
* In this project, a maze-solving problem is addressed using the **A\*** (A-star) search algorithm, which combines the strengths of Dijkstra’s algorithm and heuristic search to efficiently find the shortest path.
* The project demonstrates how AI can be used for decision-making and route optimization in constrained environments.

**PROBLEM STATEMENT**

* The task is to find the **shortest possible path** from a start cell to a goal cell within a maze containing obstacles. The maze can have multiple routes, and the algorithm must determine the most efficient one while avoiding walls.
* The problem highlights the need for intelligent pathfinding that minimizes computational cost while guaranteeing an optimal result.

**GOAL**

* To design and implement an algorithm capable of **solving any given maze** by finding the shortest path between start and goal nodes.
* To visualize the exploration and pathfinding process using Python.
* To analyze the **efficiency** of A\* compared to other classical search algorithms like BFS and Dijkstra.

**THEORETICAL BACKGROUND**

* + Maze solving is a **search problem** where each cell is treated as a **state**.
  + Common search algorithms include:
    - **Depth-First Search (DFS)** – explores deep but not always optimal.
    - **Breadth-First Search (BFS)** – guarantees shortest path but uses more memory.
    - **Dijkstra’s Algorithm** – works for weighted graphs but is slower.
    - **A\*** Algorithm – combines path cost and heuristic guidance.
  + **A\*** uses the function:

where:

* + - g(n) = cost from start to current node.
    - h(n) = estimated cost (heuristic) to reach the goal.
  + **Heuristic used:** Manhattan Distance — suitable for 2D grid mazes.
  + **Justification:** A\* is both **optimal** and **complete**, offering best performance among classical methods.

**ALGORITHM EXPLANATION WITH EXAMPLE**

* Initialize **open list** (priority queue) and **closed list**.
* Add the **start node** to the open list with f-score = 0.
* Pick the node with the **lowest f(n)** from the open list.
* Expand neighbors (Up, Down, Left, Right) — skip walls or invalid cells.
* Calculate:

for each neighbor.

* Update if a **better path** is found.
* Move current node to the **closed list**.
* Continue until the **goal node** is reached.
* **Trace back** the path from goal to start.

**Example:**

* In a 5×5 maze, if start = (0,0) and goal = (4,4),  
  A\* will explore nearby cells and find the shortest path while skipping walls.

**IMPLEMENTATION AND CODE**

import heapq

import matplotlib.pyplot as plt

import matplotlib.animation as animation

import numpy as np

import random

# ---------- Heuristic Function ----------

def heuristic(a, b):

"""Manhattan distance heuristic"""

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

# ---------- A\* Algorithm ----------

def astar\_visual(maze, start, goal):

rows, cols = len(maze), len(maze[0])

open\_list = []

heapq.heappush(open\_list, (0, start))

came\_from = {}

g\_score = {start: 0}

f\_score = {start: heuristic(start, goal)}

visited = []

while open\_list:

current = heapq.heappop(open\_list)[1]

visited.append(current)

if current == goal:

# Reconstruct final path

path = []

while current in came\_from:

path.append(current)

current = came\_from[current]

path.append(start)

path.reverse()

return path, visited

# Explore 4-directional neighbors

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

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

if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols:

if maze[neighbor[0]][neighbor[1]] == 1:

continue # Skip walls

tentative\_g = g\_score[current] + 1

if neighbor not in g\_score or tentative\_g < g\_score[neighbor]:

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g

f\_score[neighbor] = tentative\_g + heuristic(neighbor, goal)

heapq.heappush(open\_list, (f\_score[neighbor], neighbor))

return None, visited # No path found

# ---------- Random Maze Generator ----------

def generate\_maze(rows, cols, wall\_prob=0.3):

"""Generate a random maze with 0s (paths) and 1s (walls)"""

maze = [[1 if random.random() < wall\_prob else 0 for \_ in range(cols)] for \_ in range(rows)]

maze[0][0] = 0 # Start

maze[rows - 1][cols - 1] = 0 # Goal

return maze

# ---------- Auto-Solvable Maze ----------

def generate\_solvable\_maze(rows, cols, wall\_prob=0.3):

"""Keep generating random mazes until one is solvable"""

print("Generating solvable maze...")

attempts = 0

while True:

maze = generate\_maze(rows, cols, wall\_prob)

path, visited = astar\_visual(maze, (0, 0), (rows - 1, cols - 1))

attempts += 1

if path: # Path found

print(f"✅ Solvable maze generated in {attempts} attempt(s)")

return maze, path, visited

# ---------- Maze Setup ----------

rows, cols = 12, 12

maze, path, visited = generate\_solvable\_maze(rows, cols, wall\_prob=0.28)

start = (0, 0)

goal = (rows - 1, cols - 1)

# ---------- Visualization Setup ----------

color\_map = {

0: [1, 1, 1], # White (open cell)

1: [0, 0, 0], # Black (wall)

}

fig, ax = plt.subplots()

img = np.array([[color\_map[val] for val in row] for row in maze], dtype=float)

im = ax.imshow(img)

def update(frame):

"""Update frame for animation"""

temp = np.array([[color\_map[val] for val in row] for row in maze], dtype=float)

# Mark visited cells

for (x, y) in visited[:frame]:

temp[x][y] = [0.3, 0.5, 1.0] # Blue (visited)

# Mark final path

if path:

for (x, y) in path:

temp[x][y] = [1.0, 1.0, 0.3] # Yellow (final path)

# Mark start & goal

sx, sy = start

gx, gy = goal

temp[sx][sy] = [0.0, 1.0, 0.0] # Green (start)

temp[gx][gy] = [1.0, 0.0, 0.0] # Red (goal)

im.set\_array(temp)

return [im]

ani = animation.FuncAnimation(fig, update, frames=len(visited) + 20,

interval=180, repeat=False)

plt.title("A\* Maze Solving Visualization (Random + Always Solvable)")

plt.axis("off")

plt.show()

**OUTPUT**

* The program begins generating random mazes.
* The message **“✅ Solvable maze generated”** confirms a valid maze was created where a path exists from start to goal.
* This ensures the upcoming visualization will successfully display a solvable maze.

A black background with white text

AI-generated content may be incorrect.

* The maze grid is displayed using colors to represent different states:
* **Green** → Start cell
* **Red** → Goal cell
* **Black** → Walls or obstacles
* **White** → Open paths
* **Blue** → Explored cells during search
* **Yellow** → Final shortest path found by A\*
* The yellow path clearly shows the **optimal route** from start to goal through open cells, avoiding obstacles.

A colorful squares and a yellow line

AI-generated content may be incorrect.

**RESULTS AND FUTURE ENHANCEMENT**

* **Results**
  + The A\* algorithm successfully finds the **shortest path** between the start and goal in the generated maze.
  + The algorithm intelligently avoids walls and explores only the necessary paths.
  + The **visual output** clearly shows the exploration process (blue cells) and the final shortest path (yellow cells).
  + A\* performs efficiently even in medium-sized mazes (10×10, 12×12 grids).
  + The **runtime is low** (under 0.2 seconds for smaller mazes).
  + The algorithm guarantees an **optimal solution** if the heuristic is admissible.
  + Compared to other classical algorithms (DFS, BFS, Dijkstra), A\* provides a **balance of speed and accuracy**.
* **Future enhancements**
  + Add **diagonal movements** to make pathfinding more flexible and realistic.
  + Extend the system to handle **3D or larger mazes**.
  + Integrate **dynamic obstacle detection**, where the maze changes during execution.
  + Combine A\* with **Reinforcement Learning (Q-Learning)** for adaptive and self-improving navigation.
  + Develop a **robot navigation simulator** using the same logic for real-world applications.
  + Implement **weighted paths** to simulate different terrain costs (e.g., easy vs. difficult areas).

|  |  |
| --- | --- |
| **Git Hub Link of the project and report** | **https://github.com/harinik1510/HariniK** |

**REFERENCES**

* **Stuart Russell & Peter Norvig (2020).** *Artificial Intelligence: A Modern Approach* (4th Edition).  
   https://aima.cs.berkeley.edu/
* *TutorialsPoint – A Algorithm in AI*\*  
  https://www.tutorialspoint.com/artificial\_intelligence/artificial\_intelligence\_a\_star\_algorithm.htm
* *GeeksforGeeks – A Search Algorithm in Python*\*  
  https://www.geeksforgeeks.org/a-search-algorithm/
* **Towards Data Science – Understanding A\* Pathfinding Algorithm**  
  https://towardsdatascience.com/a-star-a-search-algorithm-eb495fb156bb
* **Python Official Documentation – Matplotlib Animation**  
   https://matplotlib.org/stable/api/animation\_api.html
* **Python Official Documentation – Heap Queue (heapq) Module**  
  https://docs.python.org/3/library/heapq.html