Date of Submission July 15, 2025.

Prim's Algorithm for Maze Generation and A^* Algorithm for Maze Solving

AMRIT KANDEL[1], (THA078BEI004), PRASISH TIMALSINA[2], (THA078BEI026)
[1]Department of Electronics and Computer Engineering, Thapathali Campus (e-mail: amrit.tha078bei004@tcioe.edu.np)
[2]Department of Electronics and Computer Engineering, Thapathali Campus (e-mail: prasish.tha078bei026@tcioe.edu.np)

ABSTRACT In this lab, we experimented with generating and solving a maze using two of the most well-known algorithms: Prim's Algorithm and A* (A-star) Algorithm. We generated an ideal maze using Prim's Algorithm, i.e., a maze that has a unique path connecting every pair of points. After generating the maze, we applied the A* algorithm to find the shortest path from a start point to an end point. A* uses an effective path with a cost function and a heuristic. We implemented everything in Python and generated visual outputs in order to view the generated maze and the solved path.

INDEX TERMS Prim's Algorithm, A* Algorithm, maze solving

I. INTRODUCTION

A. MINIMUM SPANNING TREE

A minimum spanning tree (MST) is defined as a spanning tree that has the minimum weight among all the possible spanning trees. The minimum spanning tree has all the properties of a spanning tree with an added constraint of having the minimum possible weights among all possible spanning trees. Like a spanning tree, there can also be many possible MSTs for a graph.

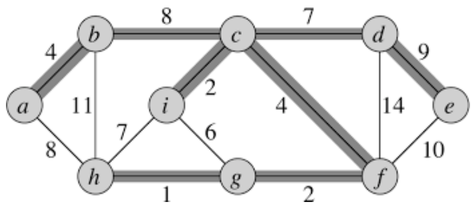

FIGURE 1. Example shouwing a minimum sapnning tree

In the above figure, the highlighted path shows the minimum spanning tree as it can connect all the nodes of the graph in minimum cost. There are several algorithms to find the minimum spanning tree from a given graph out of which Prim's Algorithm is one.

B. PRIM'S ALGORITHM

Prim's algorithm is a Greedy algorithm like Kruskal's algorithm. This algorithm always starts with a single node and moves through several adjacent nodes, in order to explore all of the connected edges along the way. Prim's algorithm finds the MST by first including a random vertex to the MST. The algorithm then finds the vertex with the lowest edge weight from the current MST, and includes that to the MST. Prim's algorithm keeps doing this until all nodes are included in the MST. The algorithm is given in 1.

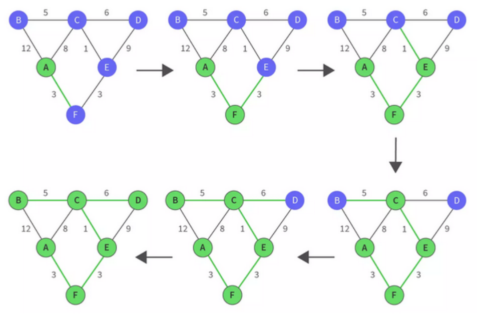

FIGURE 2. Example showing steps of Prim's Algorithm

TABLE 1. Prims Algorithm

	Steps of Prim's Algorithm
1.	Determine an arbitrary vertex as the starting vertex of the MST. We pick 0 in the below diagram.
2.	Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
3.	Find edges connecting any tree vertex with the fringe vertices.
4.	Find the minimum among these edges.
5.	Add the chosen edge to the MST. Since we consider only the edges that connect fringe vertices with the rest, we never get a cycle.
6.	Return the MST and exit

C. A* ALGORITHM

The A* (A-star) search algorithm is a popular and efficient pathfinding and graph traversal technique used in various fields such as robotics, artificial intelligence, and game development. It is an informed search algorithm, meaning it uses knowledge beyond just the current state to guide its exploration.

A* combines the strengths of Dijkstra's Algorithm and Greedy Best-First Search. It finds the shortest path from a start node to a goal node by considering both the cost to reach a node and an estimated cost (heuristic) from that node to the goal. Formally, A* uses the evaluation function:

$$f(n) = g(n) + h(n)$$

where:

- $f(n)$ is the total estimated cost of the cheapest solution through node n,
- $g(n)$ is the cost from the start node to node n,
- $h(n)$ is the estimated cost from node n to the goal (heuristic).

The heuristic function $h(n)$ plays a critical role in the performance of the A* algorithm. If it is admissible, A* is guaranteed to find the optimal solution.

II. EXPERIMENTATION DETAILS

The experiment was divided into two main phases: maze generation using Prim's Algorithm and maze solving using the A* algorithm.

A. MAZE GENERATION USING PRIM'S ALGORITHM

We used Prim's algorithm to generate a random perfect maze. The maze is represented as a grid where every cell is a node. All cells were initially considered walls. The algorithm starts with an arbitrarily selected cell and adds it to the maze. At each step, it selects the closest unvisited neighbor cell with the lowest cost and connects it so that no cycles are formed. The algorithm continues until every cell is included in the spanning tree structure.

B. MAZE SOLVING USING A* ALGORITHM

After generating the maze, we implemented the A* search algorithm to find the shortest path from the start cell to the goal cell. The A* algorithm uses a priority queue to explore paths that minimize the evaluation function:

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the lowest cost to travel from the beginning to node n, and $h(n)$ is the heuristic from node n towards the goal. We used Manhattan distance as the heuristic function, which is admissible and suitable for grid-based mazes.

The algorithm maintains visited nodes and has a trace dictionary for the tracing of the best path. Once the target is realized, the path is traced back and shown via colored markers on the grid.

III. RESULTS

After running the code, we obtained two notable outputs. First, we generated an optimal maze using Prim's Algorithm. The maze was constructed step by step by connecting neighboring cells randomly with the assurance that no loops would be formed. This assured us that the maze would have only a single lonely path between any two cells.

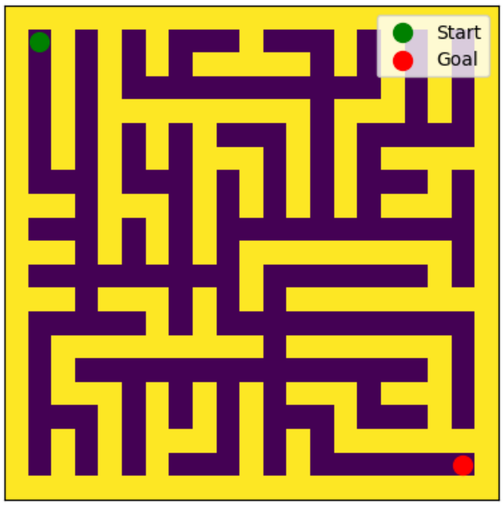

FIGURE 3. Maze using Prim's Algorithm

We utilized the A* algorithm to determine the solution to the maze by getting the shortest distance from the initial cell (usually the top-left cell) to the end cell (usually the bottom-right cell). The algorithm was capable of calculating the right path, which we graphically displayed on the maze.

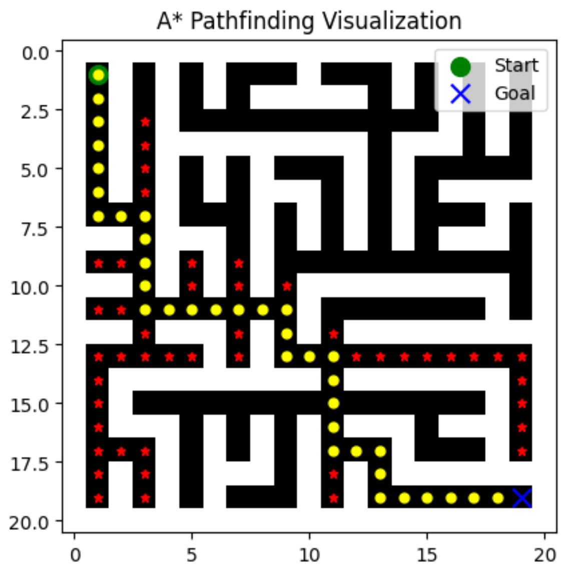

FIGURE 4. Path and visited cell using A^* algorithm

IV. DISCUSSION

Prim’s Algorithm was very efficient in generating a random maze. It built the maze by expanding step by step from a starting cell into neighboring cells without building cycles. This gave us a maze in which there was just one route between any two cells. The A* algorithm then used a combination of the actual cost of the route and an estimated cost (a heuristic) to find the shortest route through the maze. We used the Manhattan distance as our heuristic because it is appropriate for grid maps. The algorithm was effective and accurate in computing the best path, and the visual output showed the path from start to goal clearly. This experiment demonstrated how path-finding and maze generation are done using graph algorithms and how such techniques are utilized in artificial intelligence, games, and robotics.

V. CONCLUSION

In summary, we managed to create a perfect maze with the assistance of Prim’s Algorithm and solved it using the A* algorithm. The generated maze was original and had one solution path. The shortest solution was effectively identified by the A* algorithm using a heuristic-based approach.

APPENDIX A

SOURCE CODE

The source code is available in the following link: https://github.com/Amritkandel49/AI_labwork/blob/main/lab7/prims_algorithm.ipynb

```
import numpy as np
import matplotlib.pyplot as plt
import random
import heapq

def generate_maze(rows, cols):
# Ensure odd dimensions
    if rows % 2 == 0: rows += 1
    if cols % 2 == 0: cols += 1

    maze = np.ones((rows, cols), dtype=int)
    start_r = random.randrange(1, rows, 2)
    start_c = random.randrange(1, cols, 2)
    maze[start_r, start_c] = 0

# Frontier walls list: (r1, c1, r2, c2)
    walls = []
    for dr, dc in [(-2,0),(2,0),(0,-2),(0,2)]:
        r2, c2 = start_r + dr, start_c + dc

        if 0 <= r2 < rows and 0 <= c2 < cols:
            walls.append((start_r, start_c, r2, c2
                ))

    while walls:
        r1, c1, r2, c2 = walls.pop(np.random.
            randint(len(walls)))
        if maze[r2, c2] == 1:
            # Count adjacent passages around (r2,
                c2)
            neigh_passages = 0
            for dr, dc in [(-2,0),(2,0),(0,-2)
                ,(0,2)]:
                rr, cc = r2 + dr, c2 + dc
                if 0 <= rr < rows and 0 <= cc <
                    cols and maze[rr, cc] == 0:
                    neigh_passages = 0
                    neigh_passages += 1
            if neigh_passages == 1:
            # Carve wall and cell
                mid_r, mid_c = (r1 + r2)//2, (c1 +
                     c2)//2
                maze[mid_r, mid_c] = 0
                maze[r2, c2] = 0
# Add new frontier walls
                for dr, dc in [(-2,0),(2,0),(0,-2)
                    ,(0,2)]:
                    nr, nc = r2 + dr, c2 + dc
                    if 0 <= nr < rows and 0 <= nc
                        < cols and maze[nr, nc] ==
                         1:
                        walls.append((r2, c2, nr,
                            nc))
    return maze

maze = generate_maze(21, 21)

def place_start_goal(maze):
    rows, cols = maze.shape
    start = (1, 1)
    goal = (rows - 2, cols - 2)

    # Ensure start and goal are not walls
    # maze[start] = 2  # Start
    # maze[goal] = 3   # Goal

    return maze

maze = place_start_goal(maze)
print(maze)



grid = np.array(maze)
print(grid)
plt.imshow(grid, cmap=’viridis’)
start = np.argwhere(grid == 2)
goal = np.argwhere(grid == 3)

for y, x in [(1,1)]:
    plt.scatter(x, y, c=’g’, s=100, marker=’o’,
        label=’Start’)

for y, x in [(19, 19)]:
    plt.scatter(x, y, c=’r’, s=100, marker=’o’,
        label=’Goal’)

plt.xticks([])
plt.yticks([])
plt.legend()
plt.show()


ROWS = grid.shape[0] - 1
COLS = grid.shape[1] - 1

def get_neighbours(r, c):
    neighbors = []
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)
        ]
    for dr, dc in directions:
        new_r, new_c = r + dr, c + dc
        if 0 <= new_r <= ROWS and 0 <= new_c <=
            COLS and grid[new_r][new_c] != 1:  #
            Exclude walls
            neighbors.append((new_r, new_c))
    return neighbors
```

```
print("Neighbors of (19, 19):", get_neighbours(19,
     19))

def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])


import matplotlib.animation as animation
import matplotlib.colors as mcolors

#add animation in current function
def astar(maze, start, goal):
    rows, cols = len(maze), len(maze[0])
    pq = []
    heapq.heappush(pq, (0 + heuristic(start, goal)
        , 0, start, [start]))  # (f(n), g(n),
        position, path)

    visited = set()

    while pq:
        f, g, current, path = heapq.heappop(pq)

        if current == goal:
            print(path)
            return (path, visited)

        if current in visited:
            continue

        visited.add(current)

        x, y = current
        for dx, dy in [(-1,0), (1,0), (0,-1),
             (0,1)]:  # Up, Down, Left, Right
            nx, ny = x + dx, y + dy
            if 0 <= nx < rows and 0 <= ny < cols
                and maze[nx][ny] == 0:
                neighbor = (nx, ny)
                if neighbor not in visited:
                    g = g + 1
                    f = g + heuristic(neighbor,
                        goal)
                    heapq.heappush(pq, (f, g,
                        neighbor, path + [neighbor
                        ]))

    return None

start = (1, 1)
goal = (19,19)
path, visited = astar(maze, start, goal)
print(path)
# print("Path found:" if 0

def plot_path(maze, path):
    plt.imshow(maze, cmap='gray', origin='upper')
    if path:
        x, y = zip(*path)
        plt.plot(y, x, marker='o', color='red',
            linewidth=2, markersize=5)
    plt.scatter(start[1], start[0], marker='o',
        color='green', s=100, label='Start')
    plt.scatter(goal[1], goal[0], marker='x',
        color='blue', s=100, label='Goal')
    plt.legend()
    plt.title('A* Pathfinding Visualization')
    plt.show()
plot_path(maze, path)

def plot_path(maze, visited, path):
    plt.imshow(maze, cmap='gray', origin='upper')
    for x, y in visited:
        plt.plot(y, x, marker='*', color='red',
            linewidth=2, markersize=5)
        if (x, y) in path:
            plt.plot(y, x, marker='o', color='
                yellow', linewidth=2, markersize
                =5)

    plt.scatter(start[1], start[0], marker='o',
        color='green', s=100, label='Start')
    plt.scatter(goal[1], goal[0], marker='x',
        color='blue', s=100, label='Goal')
    plt.legend()
    plt.title('A* Pathfinding Visualization')
    plt.show()
plot_path(maze, visited, path)
print(visited)
```