Maze Generation using Prim's Algorithm and Solving using A* Algorithm with Visualization

Anveshan Timsina*, Sandesh Dhital†, and Utsab Dahal‡
Department of Electronics and Computer Engineering, Thapathali Campus
Institute of Engineering, Tribhuvan University, Kathmandu, Nepal
Email: *anveshan.078bei005@tcioe.edu.np, †sandesh.078bei034@tcioe.edu.np, ‡utsab.078bei046@tcioe.edu.np

***Abstract*—This report explores the use of the A* search algorithm in solving and visualizing grid-based mazes generated through Prim's algorithm. A* is an informed search algorithm that integrates both path cost and heuristic estimates to identify the shortest and most efficient path from a start point to a goal. In this experiment, Prim's algorithm has been first adapted to generate perfect mazes with guaranteed connectivity and no cycles. A* is then implemented to navigate these mazes, using the Manhattan distance as a heuristic due to the restriction to four-directional movement. The entire search process, including visited cells and final path, is visualized using matplotlib animations. The results validate the efficiency and effectiveness of A* in solving complex mazes with both optimality and computational efficiency.**

***Index Terms*—Prim's Algorithm, Minimum Spanning Tree, A* Algorithm, Heuristics, Maze Visualization, Matplotlib, Numpy, Python.**

I. Introduction

A. Background

In artificial intelligence, the development of efficient path-finding algorithms has been a central area of research, particularly in fields like robotics, game development, and autonomous navigation. Among the many algorithms devised for this purpose, the A* (A-star) algorithm stands out for its ability to combine the benefits of uniform-cost search with heuristic-driven strategies. Introduced in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael, A* was designed as an extension of Dijkstra's algorithm, incorporating heuristic estimations to guide the search more intelligently toward the goal. It was a landmark advancement in search techniques, providing a framework that could guarantee both optimality and efficiency under specific conditions.

It works by evaluating paths based on the sum of two functions; the cost to reach a node from the start (g(n)) and the estimated cost from that node to the goal (h(n)). The combination, f(n) = g(n) + h(n), allows A* to prioritize nodes that are likely to lead to a shorter overall path. The key to its effectiveness lies in the choice of the heuristic function. When the heuristic is admissible (never overestimates the actual cost), A* guarantees finding the shortest path. If the heuristic is also consistent, the algorithm becomes even more efficient by avoiding redundant searches.

In comparison to breadth-first search (BFS) and depth-first search (DFS), A* demonstrates clear advantages, particularly in terms of path optimality and search efficiency. BFS is exhaustive and guarantees the shortest path in an unweighted graph, but it can be memory-intensive and slow in large or complex environments. DFS, on the other hand, is memory-efficient but may not find the shortest path and can get stuck exploring deep paths that lead nowhere. A* strategically balances these extremes. By leveraging both the actual cost and heuristic estimate, it narrows down the search to the most promising paths, reducing unnecessary exploration and computational overhead.

B. Objectives

This lab experiment aims to:

- Implement A* algorithm on a grid-based maze (generated using Prim's algorithm) to solve it and determine the path length.
- visualize the search process using animations and highlight the path from start (initial state) to the goal (goal state).

II. Experimentation

A. Prim's Algorithm

Prim's algorithm is a classic greedy algorithm used to find the minimum spanning tree (MST) of a connected, undirected, weighted graph. A spanning tree is a subset of the graph that connects all vertices with the minimum number of edges and without forming any cycles. The goal of the minimum spanning tree is to ensure that the total weight of the selected edges is as small as possible while still maintaining connectivity across the entire graph.

1: Choose an arbitrary vertex s as the starting vertex
2: Initialize a set MST $= \{s\}$
3: Create a priority queue (or min-heap) of all edges (s, v) where $v \notin$ MST
4: **while** MST does not include all vertices **do**
5: Find the edge (u, v) with the smallest weight such that $u \in$ MST and $v \notin$ MST
6: Add vertex v to MST
7: Add all edges (v, x) to the priority queue where $x \notin$ MST
8: **end while**
9: **return** the set of edges included in MST

Prim's algorithm, though traditionally used for constructing a Minimum Spanning Tree (MST) of a weighted graph, can

be adapted effectively for maze generation. The key idea is to treat the maze as an undirected graph where each cell represents a node, and each possible passage between neighboring cells is an edge. This method leads to the creation of a perfect maze in which there is exactly one unique path between any two cells, with no loops or inaccessible regions.

```
def generate_maze(rows, cols):

    if rows % 2 == 0: rows += 1
    if cols % 2 == 0: cols += 1

    maze = np.ones((rows, cols), dtype=np.int8)

    start_r = random.randrange(1, rows, 2)
    start_c = random.randrange(1, cols, 2)
    maze[start_r, start_c] = 0  # Passage
    wall_list = []

    def add_frontiers(r, c):
        for dr, dc in [(-2,0), (2,0), (0,-2),
            (0,2)]:
            nr, nc = r + dr, c + dc
            if 0 < nr < rows and 0 < nc < cols and
                maze[nr, nc] == 1:
                wall_list.append((r, c, nr, nc))

    add_frontiers(start_r, start_c)

    while wall_list:
        idx = random.randint(0, len(wall_list)-1)
        r1, c1, r2, c2 = wall_list.pop(idx)

        if maze[r2, c2] == 1:
            adjacent_passages = 0
            for dr, dc in [(-2,0), (2,0), (0,-2),
                (0,2)]:
                ar, ac = r2 + dr, c2 + dc
                if 0 <= ar < rows and 0 <= ac <
                    cols and maze[ar, ac] == 0:
                    adjacent_passages += 1

            if adjacent_passages == 1:
                mid_r, mid_c = (r1 + r2) // 2, (c1
                    + c2) // 2
                maze[mid_r, mid_c] = 0
                maze[r2, c2] = 0
                add_frontiers(r2, c2)

    return maze

maze = generate_maze(21, 21)
```

Here, Prim's algorithm serves as a randomized maze generator by starting from an initial cell and gradually connecting it to unvisited neighboring cells using randomly selected, minimal-cost edges (edge cost for each adjacent cell being 1 in this implementation).

B. A Algorithm*

The A* (A-star) algorithm is an informed search algorithm widely used in path-finding and graph traversal problems. Unlike uninformed search strategies such as BFS or DFS, which have no prior knowledge of the goal location, A* incorporates heuristic information to guide its search. This makes it more efficient in finding optimal paths, especially in large or complex environments. A* intelligently balances the actual cost to reach a node and an estimate of the remaining cost to reach the goal, enabling it to prioritize more promising paths and reduce unnecessary exploration.

A heuristic is an estimate (based on domain knowledge) used by search algorithms to predict the cost from a given node to the goal. It helps guide the search process more efficiently by prioritizing paths that appear more promising. While a heuristic is not guaranteed to be accurate but provides a reasonable guess to inform the algorithm's decisions. For example, in grid-based path-finding, a common heuristic is the Manhattan distance, which calculates the sum of the horizontal and vertical distances between two points. For a node at position (x_1, y_1) and a goal at (x_2, y_2), the Manhattan distance is given by;

$$h(n) = |x_1 - x_2| + |y_1 - y_2|$$

It is widely used in grid-based pathfinding (such as maze solving) where movement is restricted to four directions: up, down, left, and right. When the heuristic never overestimates the true cost to the goal (i.e., it is admissible), A* is guaranteed to find the shortest path. If the heuristic also satisfies the triangle inequality (i.e., it is consistent or monotonic), A* avoids revisiting nodes and becomes more efficient.

1: Initialize the open list (priority queue) with the start node
2: Initialize the closed list (visited set) as empty
3: Assign $g(\text{start}) = 0$ and calculate $f(\text{start}) = g + h$
4: **while** open list is not empty **do**
5: Remove the node n with the lowest $f(n)$ from the open list
6: **if** n is the goal node **then**
7: Reconstruct and return the path
8: **end if**
9: Add n to the closed list
10: **for** each neighbor of n **do**
11: **if** neighbor is in the closed list **then**
12: **continue** to next neighbor
13: **end if**
14: Calculate tentative $g(\text{neighbor}) = g(n) + \text{cost}(n, \text{neighbor})$
15: **if** neighbor not in open list **or** tentative g is lower **then**
16: Update $g(\text{neighbor})$
17: Calculate $f(\text{neighbor}) = g + h$
18: Set the parent of neighbor to n
19: Add or update neighbor in the open list
20: **end if**
21: **end for**
22: **end while**
23: **return** failure (no path exists)

```
def a_star_animated_logging(maze, start, goal):
    rows, cols = maze.shape
    new_maze = maze.copy()
    visited = set()
    parent = {}

    heap = []
    heapq.heappush(heap, (0 +
        manhattan_distance(start, goal), 0, start))
```

```
    frames = [new_maze.copy()]
    visited_count = 0
    path_length = 0

    t0 = time.time()
    while heap:
        f, g, (r, c) = heapq.heappop(heap)
        if (r, c) == goal:
            cur = goal
            while cur != start:
                if new_maze[cur] not in [2, 3]:
                    new_maze[cur] = 5  # Path=Blue
                    frames.append(new_maze.copy())
                cur = parent[cur]
                path_length += 1
            t1 = time.time()
            exec_time = t1 - t0
            print(f"Visited nodes: {visited_count}")
            print(f"Path length: {path_length}")
            print(f"Execution time: {exec_time:.5f}
                seconds")
            return frames, True

        if (r, c) in visited:
            continue
        visited.add((r, c))
        visited_count += 1

        if new_maze[r, c] not in [2, 3]:
            new_maze[r, c] = 4  # Visited=Green
            frames.append(new_maze.copy())

        for nr, nc in get_neighbors(maze, r, c):
            if (nr, nc) not in visited:
                parent[(nr, nc)] = (r, c)
                g_new = g + 1
                f_new = g_new +
                    manhattan_distance((nr, nc),
                    goal)
                heapq.heappush(heap, (f_new, g_new,
                    (nr, nc)))

    t1 = time.time()
    exec_time = t1 - t0
    print("No path found.")
    print(f"Visited nodes: {visited_count}")
    print(f"Execution time: {exec_time:.5f}
        seconds")
    return frames, False

maze = generate_maze(21, 21)
maze_with_sg = place_start_goal(maze)
start = (1, 1)
goal = (maze.shape[0]-2, maze.shape[1]-2)

frames, found =
    a_star_animated_logging(maze_with_sg, start,
    goal)

astar_cmap = mcolors.ListedColormap(['white',
    'black', 'orange', 'red', 'green', 'blue'])
astar_bounds = [0,1,2,3,4,5,6]
astar_norm = mcolors.BoundaryNorm(astar_bounds,
    astar_cmap.N)

fig, ax = plt.subplots(figsize=(10, 10))
im = ax.imshow(frames[0], cmap=astar_cmap,
    norm=astar_norm)
plt.axis('off')
plt.title("A* Search Animation")

def animate(i):
    im.set_data(frames[i])
    return [im]

ani = animation.FuncAnimation(fig, animate,
    frames=len(frames), interval=30, blit=True)
plt.close(fig)
HTML(ani.to_jshtml())
```

III. Result

A randomly generated 21 by 21 example maze has been shown in Figure 1. This was solved using A* algorithm. The start node was represented by orange, goal by red, visited path by green and final solution by blue. For further insight on the working of A* algorithm, the length of the path with number of visited nodes and total execution time was determined (Table I).

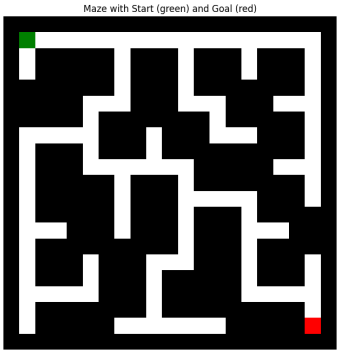

Fig. 1: Unsolved Maze Visualized

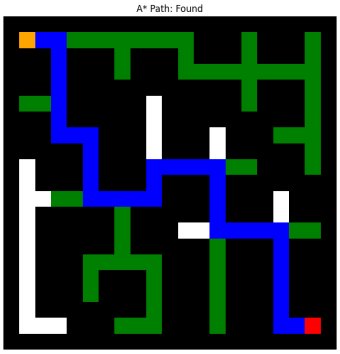

Fig. 2: Maze Solved using A* Visualized

TABLE I: A* Algorithm Performance Metrics

Metric	A* Algorithm
Path length	52
Total visited nodes	86
Execution time (s)	0.00102

IV. Discussion

The implementation of the A* algorithm on mazes generated via Prim's algorithm demonstrates the strengths of informed search strategies in structured environments. Prim's

algorithm ensured that each generated maze had a unique path between any two cells, creating ideal test cases for evaluating path-finding methods. By applying A* to these mazes, the algorithm was able to leverage the admissible and consistent Manhattan distance heuristic to significantly reduce unnecessary exploration. Compared to uninformed approaches like BFS or DFS, which either explore all paths uniformly or dive deep into specific branches, A* intelligently focuses its efforts on the most promising paths by combining actual and estimated costs.

The visualizations created through matplotlib provided intuitive insights into the step-by-step search process. The use of distinct color coding for visited cells, start and goal states, and final path allowed for real-time tracking of the algorithm's decision-making process. Performance metrics such as execution time, number of visited nodes, and path length were also logged to evaluate the algorithm quantitatively. These metrics confirmed that A* was not only able to find the shortest path but did so with efficient resource usage.

V. Conclusion

This lab session successfully demonstrates the power and practicality of the A* algorithm for solving and visualizing maze navigation problems. By combining the systematic maze generation capabilities of Prim's algorithm with the intelligent search capabilities of A*, we achieved a robust framework for path-finding in structured grid environments. A* proved to be an optimal and efficient solution, outperforming traditional uninformed methods by incorporating heuristic guidance into its search process. The Manhattan distance heuristic served as a suitable choice for four-directional movement scenarios, maintaining both admissibility and consistency. Through visualization, the exploration patterns of the algorithm became more intuitive. The project enhanced understanding of search mechanics, legal state transitions, and animation in Python.