Project I Report for COM S 4/5720 Spring 2025: A* Algorithm

Jesus Soto Gonzalez

***Abstract*—This project involves developing a path planning algorithm for solving grid-based navigation tasks. Each task consists of an agent navigating from a specified start position to a goal position while avoiding obstacles. I implemented the A* search algorithm to efficiently compute optimal paths in these environments, utilizing a Manhattan distance heuristic due to the uniform movement cost of 1.**

My implementation dynamically expands the most promising nodes using a min-heap priority queue, ensuring efficient exploration and optimal path selection. The algorithm was tested on 100 different task configurations, evaluating its performance in varied grid environments with different obstacle densities. Results indicate that A* efficiently finds all valid paths. This report details the algorithm's design, mathematical formulation, and implementation process.

I. Introduction

Pathfinding is a fundamental component of robotics, autonomous navigation, and various AI-driven systems, enabling efficient movement through complex environments. As discussed in lecture, the A* algorithm is widely used because it efficiently computes optimal paths by combining actual and estimated costs. Its balance between completeness, optimality, and efficiency makes it preferable over uninformed search methods such as BFS and Dijkstra.

This report presents my optimized A* implementation for grid-based navigation, focusing on theoretical foundations, heuristic selection, algorithm design, and performance evaluation. Additionally, I compare A* with BFS and Dijkstra to justify its selection.

II. Theoretical Background

The A* search algorithm finds the shortest path from a start node s to a goal node g by minimizing the estimated total cost function:

$$f(n) = g(n) + h(n) \tag{1}$$

where:

- $g(n)$ represents the exact cost from the start node s to the current node n.
- $h(n)$ is the heuristic estimate of the cost from node n to the goal g.

A. Comparison with BFS and Dijkstra

Several algorithms exist for pathfinding, but A* offers significant advantages:

- **BFS** expands all nodes at the current depth before moving deeper but does not account for path costs. This makes it inefficient for weighted grids since all paths are treated equally.
- **Dijkstra's Algorithm** finds the shortest path by expanding all nodes in increasing order of distance. While optimal, it explores all possible paths indiscriminately, making it computationally expensive.
- **A* Search** balances efficiency and optimality by incorporating heuristics. It expands only the most promising nodes using $f(n) = g(n) + h(n)$, significantly reducing unnecessary computations while guaranteeing the shortest path.

Thus, A* is preferable over BFS and Dijkstra for this project because it efficiently finds the shortest path while limiting unnecessary explorations.

This A* implementation uses a min-heap priority queue to always expand the most promising node first. See Listing 1.

```python
open_set = []
heapq.heappush(open_set, (0, start))
```

Listing 1. Min-Heap Priority Queue for A*

B. Manhattan Distance Heuristic

Since all movement costs are equal to 1, the Manhattan distance heuristic is used:

$$h(n) = |x_n - x_g| + |y_n - y_g| \tag{2}$$

This heuristic provides an efficient estimate of the shortest possible path between a node n and the goal g by summing the horizontal and vertical distances. Given that all movement directions; horizontal, vertical, and diagonal, have a uniform cost of 1, the Manhattan heuristic ensures that the estimated cost closely matches the actual traversal effort in the grid.

A* relies on a heuristic that maintains admissibility and consistency while guiding the search toward the goal. The Manhattan heuristic is well-suited for this purpose due to the following properties:

- **Efficiency**: A well-matched heuristic reduces unnecessary node expansions. By directly reflecting the uniform movement costs, the Manhattan heuristic allows A* to focus on exploring the most promising paths while minimizing computational overhead.
- **Alignment with Movement Costs**: Since all movement directions have the same cost, the heuristic provides a direct and accurate estimation of the remaining distance, eliminating any distortions that might arise from heuristics designed for variable-cost movement.

The heuristic function is shown in Listing 2:

```python
def heuristic(a, b):
    dx, dy = abs(a[0] - b[0]), abs(a[1] - b[1])
    return dx + dy  # Manhattan distance heuristic
```

Listing 2. Manhattan Distance Heuristic Function

III. Completeness and Optimality

A* is **complete**, meaning it is guaranteed to find a solution if one exists. According to **Theorem 1**, since the Manhattan heuristic is **admissible**, A* will always expand the necessary nodes to reach the goal. This ensures that all valid paths are systematically explored before termination. Listing 3 illustrates how the main loop processes nodes in order of increasing $f(n)$, ensuring that all reachable nodes are considered.

```
while open_set:
    _, current = heapq.heappop(open_set)
    open_set_lookup.remove(current)

    if current == end:
        path = deque()
        while current in came_from:
            path.appendleft(current)
            current = came_from[current]
        path.appendleft(start)
        return np.array(path)
```

Listing 3. Main Loop of A* Algorithm

A* is also **optimal** because it guarantees finding the shortest path when using an **admissible** heuristic. According to **Definition 1**, a heuristic function is admissible if:

$$h(n) \leq h^*(n), \quad \forall n \tag{3}$$

where $h^*(n)$ represents the true cost from node n to the goal. Since the Manhattan heuristic satisfies this condition by never overestimating the actual cost, it ensures that A* always expands the most promising nodes first, leading to the shortest path selection.

The use of a **priority queue** guarantees that nodes are expanded in increasing order of $f(n)$, ensuring efficiency in node selection. By **Theorem 3**, A* expands the fewest possible nodes among all admissible algorithms, making it optimally efficient in terms of node expansion.

According to **Definition 2**, the Manhattan heuristic is **consistent**, meaning that the estimated cost to reach the goal never decreases along the path. By **Theorem 2**, a consistent heuristic ensures that A* finds the optimal path while expanding each node at most once, eliminating unnecessary re-exploration.

Additionally, **Lemma 1** states that if a heuristic is consistent, the first time A* expands a node, it has already found the optimal path to that node. This property prevents redundant node visits, further reducing computational overhead.

In summary, A* achieves both completeness and optimality by systematically expanding nodes using an admissible and consistent heuristic while leveraging a priority queue to maintain efficiency.

IV. Implementation

A. Key Components

A* relies on the following data structures to efficiently track path costs and ensure optimal expansion order:

- **Priority Queue (Min-Heap)**: Stores nodes to be explored, always expanding the one with the lowest $f(n)$ value first.
- **Cost Dictionaries**:
  - $g(n)$ stores the cost from the start node to n.
  - $f(n) = g(n) + h(n)$ is the estimated total cost of reaching the goal.
- **Came-From Dictionary**: Stores parent nodes for reconstructing the shortest path once the goal is reached.

B. State Classification

Each node in the search space falls into one of three categories:

- **Unexplored Set** U: Nodes that have not been added to the priority queue.
- **Frontier Set** F: Nodes currently in the priority queue, pending expansion.
- **Explored Set** E: Nodes that have been fully expanded and processed.

The **Frontier Set** represents the boundary between explored and unexplored nodes, allowing A* to make informed decisions about which node to expand next.

C. Handling of Obstacles

In grid-based path planning, obstacles represent impassable cells that must be avoided. During the **frontier expansion** step, each neighboring node is checked to ensure it is within grid boundaries and is not an obstacle. If a node is in an obstacle cell, it is ignored and not added to the priority queue. This ensures that A* always finds a valid path while avoiding collisions.

The following pseudocode describes the process of handling obstacles:

Algorithm 1 Obstacle Handling in A*

```
1:  for each neighbor $n'$ of the current node $n$ do
2:      if $n'$ is out of bounds OR is an obstacle then
3:          Skip to the next neighbor
4:      end if
5:      $g_{\text{new}} \leftarrow g(n)+$ movement cost
6:      if $g_{\text{new}} < g(n')$ then (Better path found)
7:          $g(n') \leftarrow g_{\text{new}}$ (Update cost)
8:          $f(n') \leftarrow g(n') + h(n')$ (Compute priority)
9:          Set parent of $n'$ for path reconstruction
10:         if $n'$ is not in the priority queue then
11:             Add $n'$ to the priority queue
12:         end if
13:     end if
14: end for
```

This process ensures that obstacles are effectively avoided while the algorithm continues to explore only valid paths.

D. Pseudocode Implementation

The following pseudocode provides a simplified explanation of the implemented A* search algorithm optimized for grid-based pathfinding.

Algorithm 2 A* Pathfinding Algorithm

```
1:  **Input:** Grid, start position, goal position
2:  Initialize priority queue with start node
3:  Initialize cost dictionaries and tracking structures
4:  while priority queue is not empty do
5:      Extract node with lowest $f(n)$ value (frontier)
6:      if goal is reached then return path
7:      end if
8:      for each neighbor $n'$ of current node $n$ do
9:          Check if $n'$ is an obstacle or out of bounds
10:         if $n'$ is valid (not an obstacle) then
11:             Compute tentative $g(n')$
12:             if tentative cost improves path then
13:                 Update $g(n')$, $f(n')$, and parent tracking
14:                 Add $n'$ to priority queue if not present
15:             end if
16:         end if
17:     end for
18: end while
19: Output: Optimal path or failure
```

E. Implementation Walkthrough

The A* algorithm follows a structured process to compute the optimal path while avoiding obstacles.

1) ***Initialization:*** The algorithm initializes a priority queue (`open_set`) as a min-heap to store nodes prioritized by their estimated cost $f(n)$. It also tracks explored nodes (`open_set_lookup`), parent relationships (`came_from`), and cost values ($g(n)$ and $f(n)$). The start node is assigned a cost of zero, while all others are set to infinity.

2) ***Node Expansion:*** Nodes are processed in order of increasing $f(n)$. The node with the lowest cost is removed from the priority queue and marked as explored. If the goal is reached, the algorithm reconstructs the path using `came_from`.

3) ***Evaluating Neighbors:*** For each movement direction (horizontal, vertical, and diagonal), a neighbor is evaluated. The algorithm ensures the neighbor:

- Is within grid boundaries.
- Is not an obstacle.

If both conditions hold, a tentative cost g_{new} is calculated.

4) ***Cost Update and Frontier Expansion:*** If g_{new} improves the current cost for the neighbor, the values of $g(n)$ and $f(n)$ are updated. The parent node is also recorded. If the neighbor is not already in the priority queue, it is added for future exploration.

5) ***Path Reconstruction:*** Once the goal is reached, the shortest path is reconstructed by backtracking through `came_from`. If the priority queue is exhausted without reaching the goal, the algorithm returns failure.

6) ***Efficiency Considerations:***

- The **priority queue** ensures optimal node selection.
- The **octile distance heuristic** improves accuracy for diagonal movement.
- The **cost updates** prevent redundant node exploration.

This structured approach enables A* to efficiently compute the shortest path in grid-based environments.

V. Results and Analysis

The implemented A* algorithm was tested using a standardized set of 100 grid-based navigation tasks provided in the project dataset. Each task required the agent to navigate from a predefined start position to a specified goal position while avoiding obstacles.

A. Testing Framework

The dataset contained 100 different task configurations. Each configuration specified:

- A grid representation, where obstacles were marked as impassable cells.
- A start and goal position within the grid.
- A validity check function to ensure the correctness of the generated path.

The implementation was tested using the ‘main.py‘ script, which:

- Loaded the task configurations and initialized the grid environments.
- Applied the A* path planning algorithm to each configuration.
- Validated whether the computed path successfully connected the start and goal positions while avoiding obstacles.

B. Results

The path planner was executed on all 100 tasks, and in all cases, the algorithm successfully computed a valid path to the goal.

VI. Conclusion

This report presented my implementation of the A* algorithm for grid-based pathfinding, demonstrating its ability to efficiently compute optimal paths in structured environments. The use of the octile distance heuristic enabled the algorithm to handle diagonal movement effectively, while the priority queue ensured an optimal expansion strategy. Through systematic testing on 100 grid-based navigation tasks, the implementation consistently found valid, shortest paths while maintaining computational efficiency.

The results confirm that A* is a highly effective algorithm for solving navigation problems in static environments, making it a valuable tool for different searching applications. Its ability to balance completeness, optimality, and efficiency makes it superior to uninformed search methods and traditional uniform-cost approaches like Dijkstra’s algorithm.

References

1 P. Hart, N. Nilsson, and B. Raphael, ”A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100-107, 1968.

2 B. Weng, ”Notes on A-Star Searching,” Department of Computer Science, Iowa State University, March 1, 2025.