ME5413_Homework3

Planning

Praveen Kumar Krishnapur
Matric Number: A0304688U

April 3, 2025

Abstract

This report explores grid-based path planning algorithms applied to a floor map of VivoCity Level 2, with the objective of computing efficient routes between specified locations. An A* algorithm was implemented using both Manhattan and Euclidean heuristics to evaluate path quality and computation time. The impact of obstacle inflation was addressed to account for the agent's physical footprint. Results showed that the Manhattan heuristic provided slightly faster runtimes and shorter paths compared to the Euclidean heuristic. In the second task, the Travelling Shopper Problem was modeled using the distance matrix from A* planning. Two solution methods—Brute Force and Nearest Neighbour—were implemented and compared. While the Brute Force method yielded the optimal path with a shorter total travel distance, the Nearest Neighbour method demonstrated better computational scalability. The study highlights the trade-offs between optimality and efficiency in path planning and route optimization strategies.

Contents

1 Task 1: Global Planning

1.1 Introduction

Tasks 1 and 2 make use of a map of VivoCity Level 2. The left image is the original floor plan, while the right one is a grayscale version intended for planning algorithms. The full map measures 1000 $\times$ 1000 pixels (grid cells), with each cell corresponding to a 0.2 m $\times$ 0.2 m square area. Each grid cell can be either **Free** (value 255) or **Occupied** (value 0). A circular footprint of at least 0.3 m radius is assumed for the navigating agent.

Five key locations are defined on the map: `start`, `snacks`, `store`, `movie`, and `food`. The objective is to generate a path or trajectory between each pair of start and end points, display the resulting plan on the map, and compute the total travel distance in meters. Based on the computed distances, the most efficient route that visits all four locations and returns to the starting point (`start`) must be determined.

A* planning algorithm was implemented.

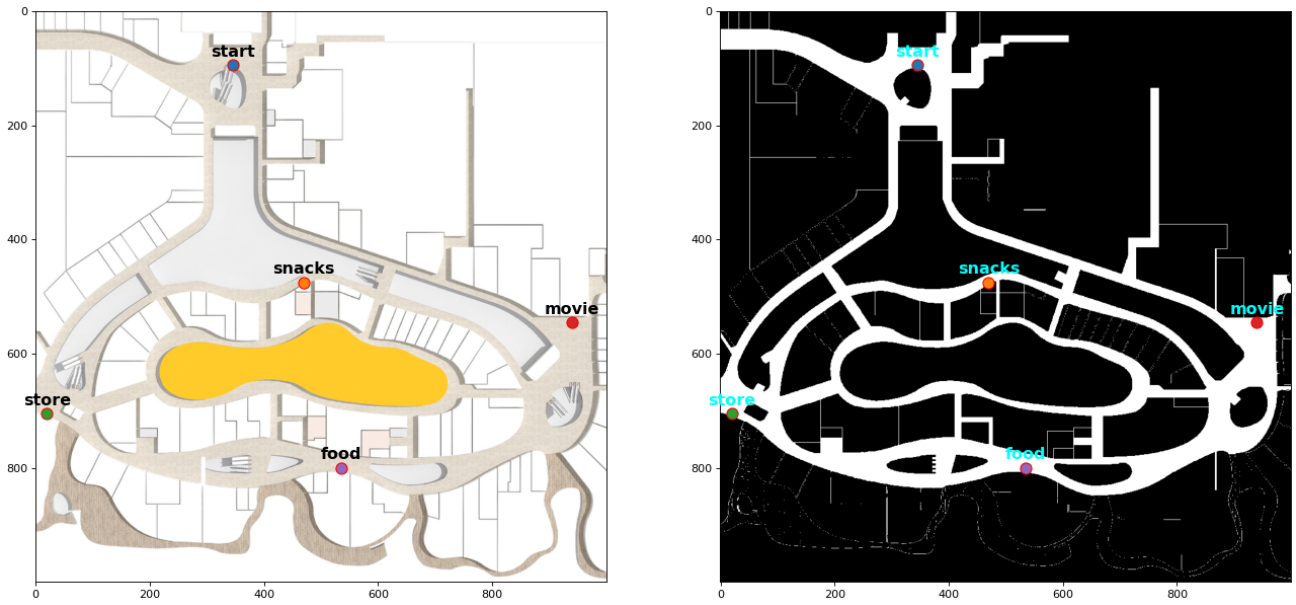

Figure 1: Map of VivoCity Level 2. The left image shows the original floor plan, and the right image is the grayscale version used for planning.

1.2 A* Algorithm

The A* algorithm is a graph-based search algorithm that finds the shortest path from a start node to a goal node using both the actual cost from the start and a heuristic estimate to the goal. In each iteration, the node with the lowest estimated total cost is expanded. This cost is the sum of the actual path cost so far ($g(n)$) and a heuristic estimate to the goal ($h(n)$), making A* both complete and optimal when an admissible heuristic is used.

The algorithm maintains two main sets: a priority queue of candidate nodes to explore and a visited list of already evaluated nodes. For each node, all its 8-connected neighbors are checked. Diagonal movements are assigned a higher cost than horizontal or vertical steps. Obstacles are inflated to account for the agent's physical size.

The pseudocode below summarizes the key steps of the implementation.

Algorithm 1: A* Algorithm for Grid-Based Path Planning

Input: Grid map, start position, goal position
Output: Shortest path and total distance

Inflate obstacles in the grid map
Initialize priority queue with start node
Initialize visited set as empty
while *priority queue is not empty* **do**
    Select node with minimum $f(n) = g(n) + h(n)$
    if *current node is goal* **then**
        Mark success and exit loop
    Move node from queue to visited
    for *each neighbor of current node* **do**
        if *neighbor is obstacle or in visited* **then**
            continue
        Compute step cost (0.2 or 0.282)
        Compute new cost $g(n)$
        if *neighbor not in queue or new cost is lower* **then**
            Add or update neighbor in queue
if *goal was reached* **then**
    Reconstruct path from goal to start using parent links
    return *total distance*
else
    return ∞

Since the circular footprint has a radius of 0.3 m and the map resolution is 0.2 m per grid cell, the effective footprint spans approximately 1.5 grid cells. To account for this discrepancy and prevent collisions with obstacles, the original `grid_map` is transformed into an `inflated_map`. This inflation process marks additional cells as occupied if they are within the radius of the footprintotprint of any actual obstacle. The figure below highlights the pixels that were originally free in the `grid_map` but were reclassified in the `inflated_map`.

Figure 2: Pixels that were free in the original map but marked as occupied in the inflated map to accommodate the agent's footprint.

1.3 Task 1 - Results and Discussions

When the heuristic function was taken as the Manhattan distance, given by

$$h(n) = |x_{\text{goal}} - x_n| + |y_{\text{goal}} - y_n|,$$

the algorithm was able to find the path between every pair of points in the list: `Start`, `Snacks`, `Store`, `Movie`, and `Food`. The table below summarizes the total distance (in meters) from each point to every other point.

From / To	Start	Snacks	Store	Movie	Food
Start	0.00	143.79	159.00	208.78	232.42
Snacks	152.87	0.00	123.71	141.13	135.25
Store	164.70	118.58	0.00	250.54	114.73
Movie	182.45	107.55	218.76	0.00	117.53
Food	234.39	134.43	132.33	117.72	0.00

Table 1: Total distances (in meters) between each location using A* with Manhattan distance heuristic.

All the paths calculated using the Manhattan heuristic function are shown in Figure 3. The time taken to calculate each path is summarized in Table 2.

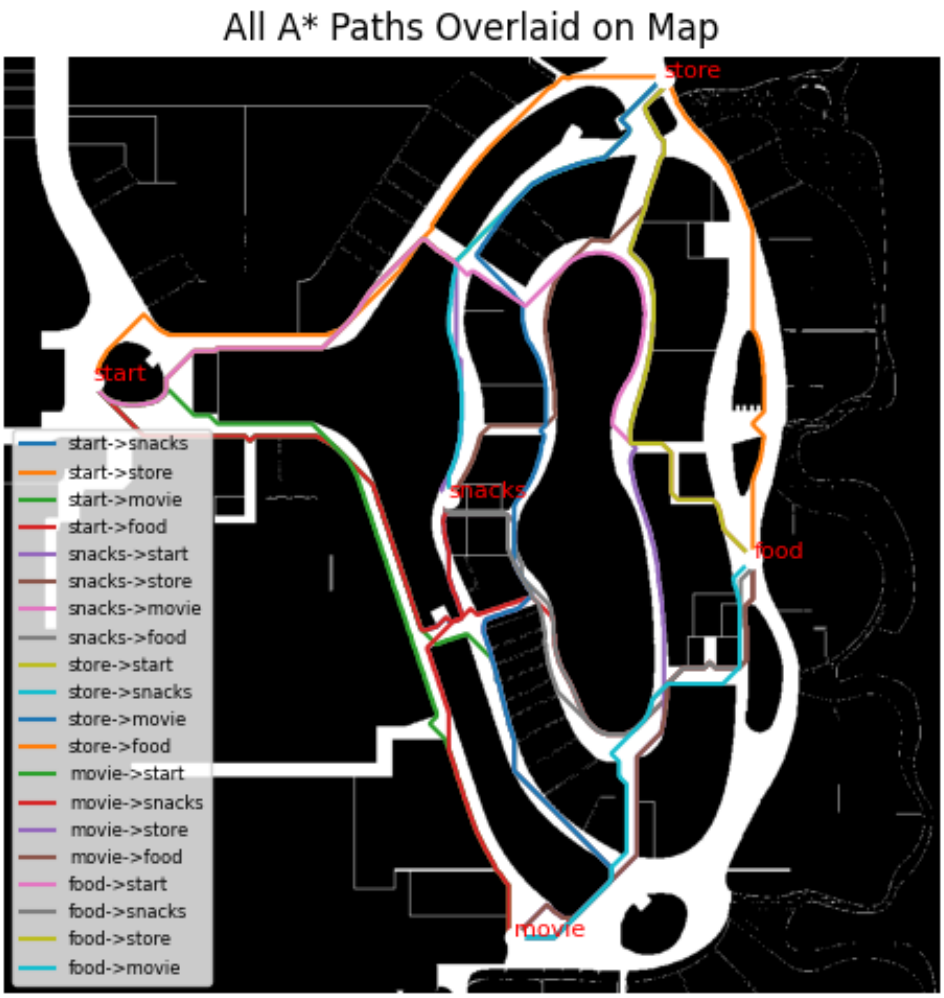

Figure 3: Paths computed between all pairs of locations using the A* algorithm with Manhattan distance as the heuristic.

From / To	Start	Snacks	Store	Movie	Food
Start	0.00	3.43	4.44	3.81	5.86
Snacks	3.16	0.00	2.53	2.93	3.06
Store	3.93	2.49	0.00	4.13	2.57
Movie	2.90	2.55	3.02	0.00	2.59
Food	6.16	4.61	2.59	2.67	0.00

Table 2: Computation time (in seconds) for each path using A* with Manhattan heuristic.

Different heuristic functions produce different outputs when used with the A* algorithm. The results below correspond to using the *Euclidean distance* heuristic instead of the Manhattan distance. The computed distance costs are slightly higher when using the Euclidean heuristic. Similarly, the time taken to compute the paths is also marginally greater.

From / To	Start	Snacks	Store	Movie	Food
Start	0.00	144.34	163.62	181.56	238.13
Snacks	154.78	0.00	122.89	125.84	134.38
Store	167.92	117.13	0.00	255.76	114.97
Movie	183.25	109.32	239.16	0.00	205.93
Food	237.18	134.43	124.39	118.12	0.00

Table 3: Total distances (in meters) using A* with Euclidean heuristic.

From / To	Start	Snacks	Store	Movie	Food
Start	0.00	4.35	2.69	3.10	6.85
Snacks	3.13	0.00	2.50	2.91	3.72
Store	2.84	2.47	0.00	4.36	2.44
Movie	2.95	2.50	2.92	0.00	6.08
Food	5.20	6.92	2.64	2.55	0.00

Table 4: Computation time (in seconds) using A* with Euclidean heuristic.

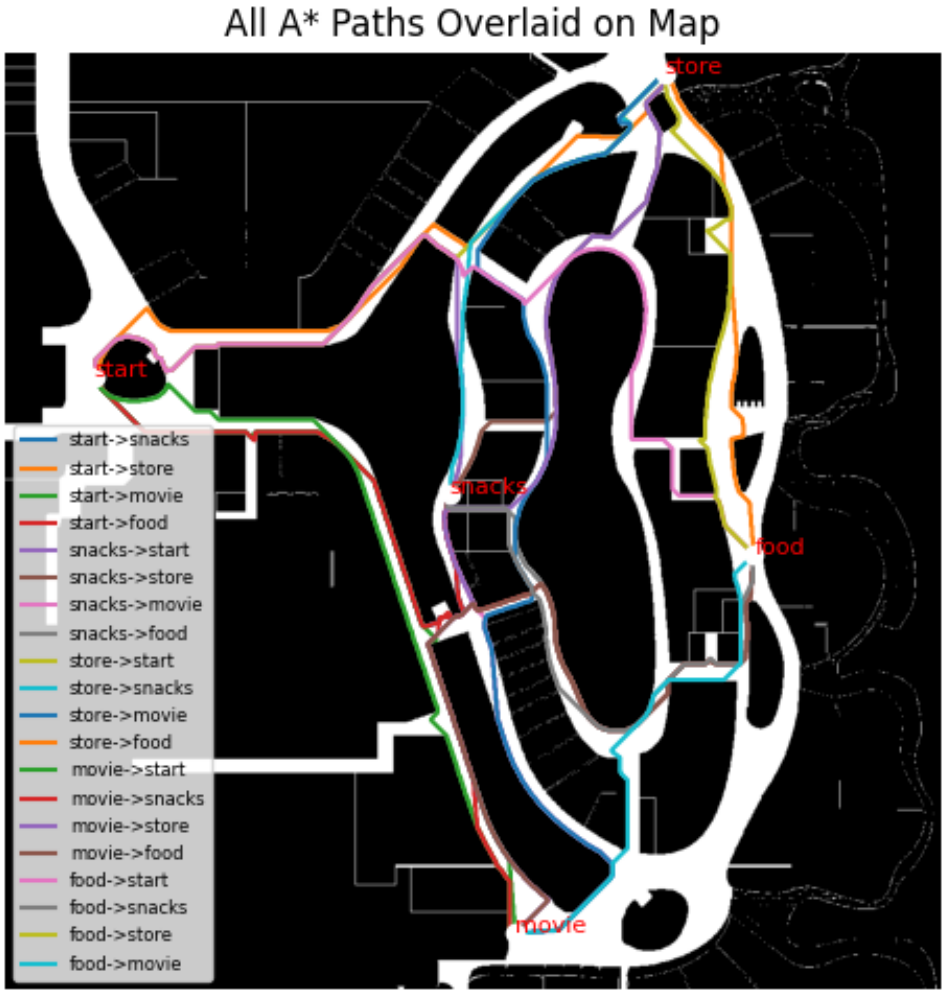

Figure 4: Paths computed between all pairs of locations using A* with Euclidean distance as the heuristic.

2 Task 2: The Travelling Shopper Problem

2.1 Introduction

To compute the shortest round-trip route that visits all four locations and returns to the starting point, the distance table obtained from the A* algorithm using the Manhattan heuristic can be used as input.

This problem is a variant of the Travelling Salesman Problem (TSP), where the objective is to minimize the total travel cost.

Two methods are considered for solving this problem:

Brute Force: This approach generates all possible permutations of the four locations to determine every possible visiting sequence. For each permutation, the total travel cost is computed by summing the distances between consecutive locations and including the return trip to the starting point. The permutation with the lowest total cost is selected as the optimal path. Although this guarantees the optimal solution, it becomes computationally expensive as the number of locations increases.

Nearest Neighbour: This heuristic method starts from the initial location and repeatedly visits the nearest unvisited location based on the distance matrix. The process continues until all locations have been visited, after which it returns to the start. This approach is faster and easier to implement, but it may not always yield the globally optimal solution.

2.2 Task 2: Results and Discussions

Using the Brute Force method, the results are as follows:

- **Best route:** start $\rightarrow$ store $\rightarrow$ food $\rightarrow$ movie $\rightarrow$ snacks $\rightarrow$ start
- **Total travel distance:** 660.80 meters

Using the Nearest Neighbour method:

- **Nearest Neighbour route:** start $\rightarrow$ snacks $\rightarrow$ store $\rightarrow$ food $\rightarrow$ movie $\rightarrow$ start
- **Total travel distance:** 683.56 meters

In this case, the Brute Force method yields a shorter total travel distance compared to the Nearest Neighbour method. This is expected, as the Brute Force approach exhaustively evaluates all possible permutations to guarantee the optimal route. However, the computational cost associated with Brute Force grows factorially with the number of locations.

The time complexity of the Brute Force approach is $\mathcal{O}(n!)$, where n is the number of locations to visit. For instance, with 4 locations (excluding the start), there are $4! = 24$ possible sequences to evaluate. While this is manageable for small n, the number of permutations increases rapidly. For example, 10 locations would result in $10! = 3,628,800$ permutations.

On the other hand, the Nearest Neighbour method has a time complexity of $\mathcal{O}(n^2)$, as it performs n lookups to find the nearest unvisited node at each step. This makes it far more scalable and efficient for larger problems, though at the cost of potential suboptimality in the solution.

Therefore, while Brute Force provides better accuracy in small-scale problems, Nearest Neighbour offers a practical trade-off between computational time and solution quality in larger settings.

3 Conclusion

This report examined the application of A* path planning in a structured indoor environment using a pixel-based grid map of VivoCity Level 2. The implementation of the A* algorithm with both Manhattan and Euclidean heuristics demonstrated the influence of heuristic choice on path optimality and runtime. The Manhattan heuristic consistently yielded lower path costs and faster computation times due to its closer alignment with grid-based movement.

To ensure safe navigation, obstacle inflation was incorporated to reflect the agent's footprint. Additionally, a comparative study of two approaches for solving the Travelling Shopper Problem—Brute Force and Nearest Neighbour—was conducted. The Brute Force method achieved the shortest route but at a factorial time complexity, making it impractical for larger datasets. In contrast, the Nearest Neighbour approach offered significantly better computational performance at the cost of slight suboptimality.

Overall, the experiments emphasize the importance of heuristic design, map preprocessing, and algorithmic scalability when addressing real-world planning problems.

References

[1] Red Blob Games. *Introduction to A* Pathfinding*. https://www.redblobgames.com/pathfinding/a-star/introduction.html

[2] Atsushi Sakai. *A* Algorithm in Python - PythonRobotics*. https://github.com/AtsushiSakai/PythonRobotics/blob/master/PathPlanning/AStar/a_star.py

[3] DataCamp. *A* Algorithm Tutorial*. https://www.datacamp.com/tutorial/a-star-algorithm