An Autonomous Agent for Package Delivery in a Grid City

Mannan Arora, 24BAS10066

September 24, 2025

1 Introduction

The goal of this project was to design and build a smart delivery agent that can navigate a city to deliver a package. The city isn't simple; it has different types of terrain, roadblocks, and even moving obstacles like other vehicles.

Our agent needed to be "rational," which is a fancy way of saying it should make smart choices. It has to find the most efficient path, not just any path, while dealing with constraints like a limited amount of fuel.

To do this, we implemented several different pathfinding algorithms, from simple ones to more complex ones, and then ran experiments to see which one performed the best and why. This report explains how we modeled our city, how the agent works, and what we found from our tests.

2 The Environment Model

Before the agent can deliver anything, we first need to build the world it lives in. We decided to model our city as a simple 2D grid, like a checkerboard. Each square on the board represents a location in the city.

2.1 The City Grid

The grid is just a list of lists in our code. Each cell has a character that tells us what's there:

- **'S'** and **'G'**: The starting point and the goal (the delivery location).
- **'X'**: A static obstacle, like a building or a permanent roadblock. The agent cannot move into these cells.
- **Numbers ('1', '2', '4', etc.)**: These represent different types of terrain. A '1' might be a smooth paved road, while a '4' could be a bumpy gravel path that costs more fuel and time to cross.

2.2 Dynamic Obstacles and Traffic

A real city isn't static. Things move and change. We added two features to model this:

1. **Moving Vehicles**: These are dynamic obstacles that block a specific cell at a specific time. In our map files, we can define a schedule like 'D,2,4,3,8', which means the cell at (row 2, column 4) is blocked at time steps 3 and 8. The agent has access to this schedule and can plan around it.

2. **Changing Traffic**: To simulate rush hour, we added traffic multipliers. A line like 'T,6,1,5,4.0' means that at time step 5, the cell at (row 6, column 1) has a traffic multiplier of 4.0. The cost to enter that cell at that time is multiplied by four.

3 The Agent's Design

Our agent is designed to be efficient. Its main goal is to get from 'S' to 'G' using the path with the lowest possible total cost, which represents a combination of time and fuel.

The main constraint on the agent is its **fuel tank**. Every move costs fuel, and more difficult terrain costs more fuel. If a potential path costs more fuel than the agent has, it cannot take that path.

To find the best path, the agent was equipped with several algorithms. We split them into three categories:

- **Uninformed Search**: These are simple algorithms that don't have any extra information about the map. They just explore systematically. We used Breadth-First Search (BFS) and Uniform-Cost Search (UCS).
- **Informed Search**: This is a "smarter" type of search. It uses a heuristic (a smart guess) to guide its exploration in the right direction. We used the A* (A-star) algorithm.
- **Local Search**: This is a different approach that starts with a complete (but maybe bad) path and tries to improve it over time. We implemented Simulated Annealing.

4 Heuristic for A* Search

The A* algorithm is special because it uses a "heuristic function" to estimate how far a cell is from the goal. This helps it decide which paths are more promising to explore.

For our grid world, we chose the **Manhattan Distance**. This is a very common and effective heuristic for grids where you can only move up, down, left, or right.

The formula is simple. For two points (x_1, y_1) and (x_2, y_2), the distance is:

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

This is like calculating the number of blocks you'd have to walk in a city grid to get from one point to another.

We chose this because it is *admissible*, which means it never overestimates the actual cost. This is important because it guarantees that A* will find the true cheapest path if one exists.

5 Experimental Results

We tested our algorithms on four different maps: a small maze, a medium map with a chokepoint, a large map designed to be a "trap" for certain algorithms, and a dynamic map with moving vehicles and traffic.

The results are summarized in the tables below. We measured three things:

- **Path Cost**: The total fuel/time cost of the final path. Lower is better.
- **Nodes Expanded**: How many grid cells the algorithm had to look at. Lower is better.
- **Time**: How long the algorithm took to run, in seconds. Lower is better.

Table 1: Results for the Small Map

Algorithm	Path Cost	Nodes Expanded	Time (s)
A* Search	6.0	8	0.0001
Uniform-Cost	6.0	11	0.0001
Simulated Annealing	6.0	1001	0.0150
BFS (steps only)	6	11	0.0001

Table 2: Results for the Large "Terrain Trap" Map

Algorithm	Path Cost	Nodes Expanded	Time (s)
A* Search	44.0	68	0.0004
Uniform-Cost	44.0	135	0.0005
Simulated Annealing	48.0	1001	0.0210
BFS (steps only)	50	138	0.0002

6 Analysis

From the results, a clear pattern emerged.

A* Search was consistently the best algorithm. On every map, it found the true cheapest path (the same cost as Uniform-Cost Search), but it did so by expanding significantly fewer nodes. On the large "Terrain Trap" map, UCS had to explore almost double the number of cells as A*. This is because the heuristic guided A* away from the long, cheap path that led in the wrong direction and kept it focused on the goal.

Uniform-Cost Search (UCS) is reliable and always finds the best path, but it is inefficient. It wastes a lot of time exploring paths that are cheap but lead far away from the goal, which makes it slow on big maps.

Simulated Annealing, as a local search, was very fast but not always optimal. On the large map, it got stuck with a path that was more expensive than the one found by A* and UCS. This shows its main trade-off: you sacrifice the guarantee of a perfect solution for speed, which can be useful in some situations but not ideal here.

BFS is not suitable for this problem. It found the path with the fewest steps, but this path was very expensive because it ignored the high-cost terrain.

Finally, our **Replanning Simulation** was a success. We simulated a new wall appearing in the middle of the map. The agent, by re-running A* from its current position on each step, was able to detect the change and immediately calculate a new, optimal path around the blockage. This proves that A* is an excellent choice for dynamic environments where things can change unexpectedly.

7 Conclusion

In this project, we successfully designed an autonomous agent capable of navigating a complex grid environment with various costs and obstacles.

Our experiments clearly show that the **A* search algorithm is the superior choice** for this task. It provides the perfect balance of optimality and efficiency. It is guaranteed to find the most efficient path (like UCS) but does so much more quickly and intelligently by using the Manhattan Distance heuristic to guide its search.

For future work, the agent could be improved by giving it more complex behaviors, such as the ability to manage multiple package deliveries in one trip, or by testing different heuristics to see how they perform on different types of maps.