AUTONOMOUS DELIVERY AGENT

A PROJECT REPORT

Submitted by

Yash kushwaha

(24BSA10016)

in partial fulfilment for the award of the degree

of

INTEGRATED MASTER OF TECHNOLOGY

in

PROGRAMME OF STUDY

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING

VIT BHOPAL UNIVERSITY

KOTHRIKALAN, SEHORE

MADHYA PRADESH - 466114

SEPTEMBER 2025

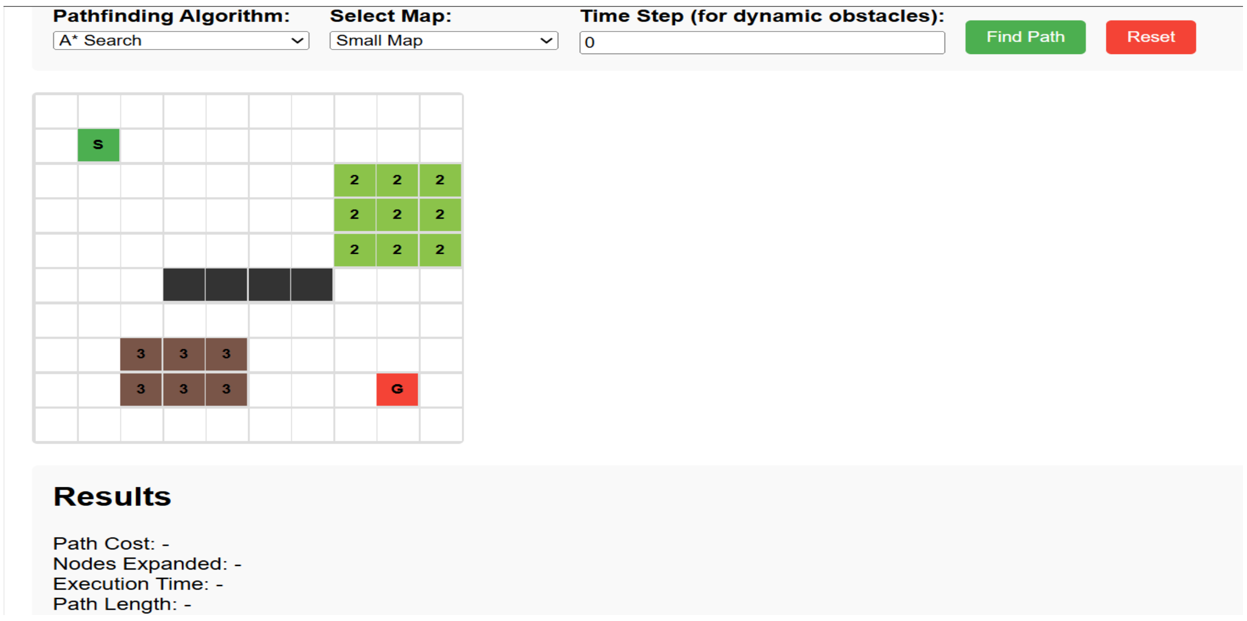
1. PROJECT OVERVIEW:

The Autonomous Delivery Agent Simulator is a web-based application designed to demonstrate and compare pathfinding algorithms in various grid environments. Built with a Flask backend and HTML/CSS/JavaScript frontend, the system visualizes four search algorithms: Breadth-First Search (BFS), Uniform Cost Search, A* Search, and Local Search (Hill Climbing). The simulator features multiple map configurations with different terrain types—road, grass, and mud—each with associated movement costs, along with static and dynamic obstacles that change position based on time steps.

Users can interactively set start and goal positions, select algorithms, and observe computed paths with color-coded visualization. The system provides performance metrics including path cost, nodes expanded, execution time, and path length, enabling direct comparison of algorithm efficiency. The modular architecture separates visualization from computation logic, ensuring maintainability and extensibility. This educational tool effectively illustrates fundamental AI pathfinding concepts while offering practical insights into algorithm behaviour across diverse environmental conditions.

2. SYSTEM ARCHITECTURE:

The frontend features an intuitive user interface built with HTML, CSS, and JavaScript, providing a responsive and interactive grid-based visualization system. Users can select algorithms, choose map types, and set custom start/goal positions through intuitive form controls. The interface displays color-coded terrain types, obstacles, and computed paths with a comprehensive legend for easy interpretation. Real-time performance metrics are prominently displayed, allowing immediate comparison of algorithm efficiency across different scenarios.


Pathfinding Algorithm: A* Search
Select Map: Small Map
Time Step (for dynamic obstacles): 0

Results

Path Cost: -
Nodes Expanded: -
Execution Time: -
Path Length: -

1. Environment Model

The environment is a fully observable, deterministic, static (or semi-static), and discrete grid world.

* Type: Grid World (2D Matrix). The environment is represented as a cell-based grid where each cell has a specific state.

* Observability: Fully Observable The agent (the pathfinding algorithm) has complete knowledge of the entire grid at all times. It knows the position of all obstacles, terrain costs, and the goal state from the outset. This is a key simplification that makes the problem tractable for these algorithms.

* Determinism: Deterministic. Every action (moving to a cell) has a guaranteed, known outcome. There is no uncertainty or randomness in the transition model. Moving north from cell (1,1) will always land you in cell (0,1), provided it's not blocked.

* Dynamics:

 * Static For most maps, the environment does not change while the agent is deliberating. Obstacles remain in place.

 * Semi-Static (Dynamic Maps): The environment can change over time (via the `time_step` parameter), but not as a direct result of the agent's actions. The dynamic obstacles move according to a pre-defined, predictable pattern (e.g., horizontal oscillation). The agent's plan is computed for a single point in time.

*   Discreteness: The environment is discrete. The agent's state is defined by its (x, y) coordinates, and it can only perform a finite set of actions (move North, South, East, West) to change its state.

2. Agent Design

The agent is a problem-solving agent that uses search algorithms to find a sequence of actions (a path) from a start state to a goal state.

*   Agent Type: Model-based, Goal-based, Utility-based.

   * Model-based: The agent maintains an internal model of the world (the grid matrix, terrain costs, obstacle locations) which it uses to simulate and evaluate actions without actually moving.

   * Goal-based: The agent's success is measured by a specific goal condition: reaching the target coordinates.

   Utility-based: The agent's performance is not just about reaching the goal but doing so optimally. It seeks to maximize performance by minimizing a cost function (the total path cost). Algorithms like UCS and A* explicitly try to find the path with the lowest utility (cost).

Agent Function: The agent's function is implemented by the search algorithms in `pathfinding.py`. It works as follows:

   1. Goal Formulation: The goal is explicitly provided by the user (the `goal` coordinate).

   2. Problem Formulation: The problem is defined by:

      States: All possible (x, y) coordinates on the grid.

      Initial State: The `start` coordinate.

      Actions: The function `Actions(s)` returns the set of legal moves (N, S, E, W) from state’s`.

      *   Transition Model: `Result (s, a)` returns the new state (adjacent cell) resulting from taking action `a` in state’s`.

      *   Goal Test: `s == goal`

* Path Cost: The cumulative sum of the terrain cost for each step in the path. The cost for a step into a cell is defined by `terrain_costs.get(cell, 1)`.

3. Search: The agent uses a chosen algorithm to explore the state space, building a path.

4. Solution Execution: The solution is the computed path, which is then visualized for the user.

Heuristics Used

A heuristic function, h(n), estimates the cost from a node `n` to the goal. It is a crucial component for informed search algorithms like A*.

Heuristic Function: Manhattan Distance

Formula: `h(n) = |x_n - x_goal| + |y_n - y_goal|`

Implementation: Defined in the `heuristic(a, b)` function in `pathfinding.py`.

Why it's used:

Admissible: It never overestimates the true cost to the goal (because you can't move diagonally, the actual cost will be at least the Manhattan distance). This is a critical property that guarantees A* will find an optimal path.

Consistent (Monotonic): The estimated cost from a node `n` to the goal is no greater than the step cost to its successor `n'` plus the estimated cost from `n'` to the goal (`h(n) <= c (n, a, n') + h(n′) `). This also guarantees optimality and allows for efficient closed-list management.

Efficient to Compute: It's a simple calculation, which is important as it needs to be computed for every explored node.

Role in Algorithms:

A* Search: Uses the evaluation function `f(n) = g(n) + h(n)`, where `g(n)` is the known cost from the start to node `n`, and `h(n)` is the heuristic estimate from `n` to the goal. This guides the search towards the goal more efficiently than UCS alone.

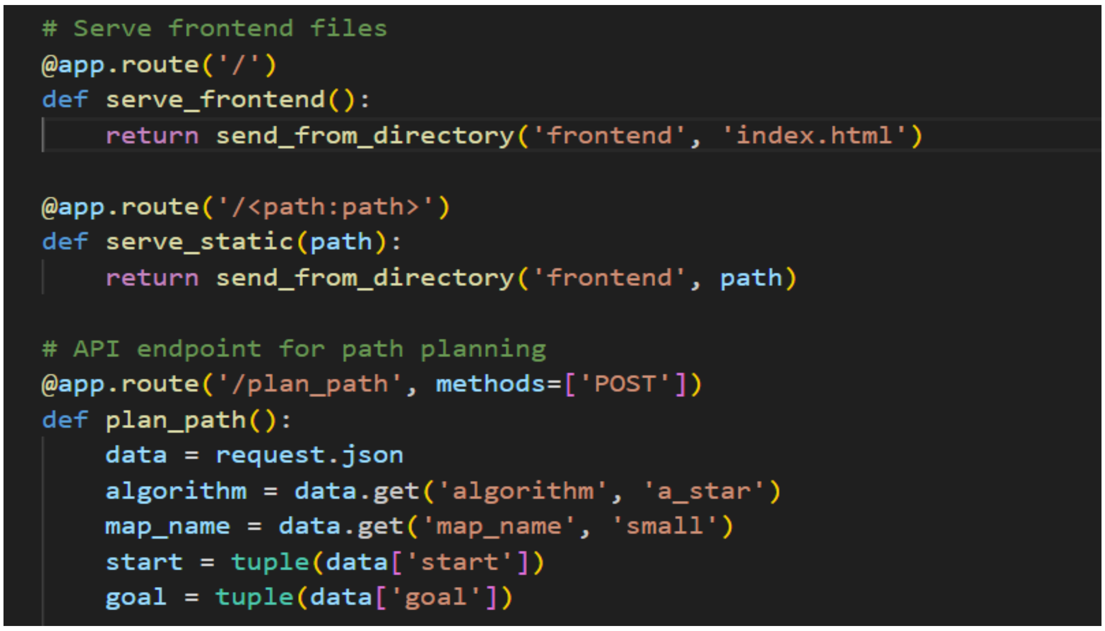
* **Local Search (Hill Climbing): The heuristic is used directly as the objective function. The algorithm tries to move to the neighbour that minimizes `h(n)` (i.e., the neighbour that appears closest to the goal). This makes it greedy and prone to getting stuck in local minima, which is why random restarts are added.

4.TECHNICAL IMPLEMENTATION:

The Autonomous Delivery Agent Simulator is built on a client-server architecture using modern web technologies. The frontend consists of a responsive single-page application built with vanilla HTML5, CSS3, and JavaScript, featuring an interactive grid system with dynamic DOM manipulation for real-time visualization. The UI implements custom event handlers for cell selection, algorithm parameters, and map configuration.

The backend utilizes Flask 2.3.3 as the web framework with Flask-CORS 4.0.0 handling cross-origin requests. The core pathfinding module implements four distinct algorithms: Breadth-First Search using queue.Queue for frontier management, Uniform Cost Search with PriorityQueue for cost-based exploration, A* Search with Manhattan distance heuristic optimization, and Local Search featuring hill climbing with random restarts for dynamic environments.

Grid management is handled through custom parsing utilities that interpret map files, terrain cost matrices (road=1, grass=2, mud=3), and dynamic obstacle patterns. The system employs coordinate-based path representation and cost calculation, with performance metrics tracking nodes expanded, execution time, and total path cost. The RESTful API endpoint (/plan_path) accepts JSON payloads and returns structured response objects containing path coordinates and analytics data.



```
# Serve frontend files
@app.route('/')
def serve_frontend():
    return send_from_directory('frontend', 'index.html')

@app.route('/<path:path>')
def serve_static(path):
    return send_from_directory('frontend', path)

# API endpoint for path planning
@app.route('/plan_path', methods=['POST'])
def plan_path():
    data = request.json
    algorithm = data.get('algorithm', 'a_star')
    map_name = data.get('map_name', 'small')
    start = tuple(data['start'])
    goal = tuple(data['goal'])
```

5. PERFORMANCE METRICS:

The simulator tracks and displays four key performance metrics for quantitative algorithm analysis. **Path Cost** represents the total traversal expense, calculated by summing terrain costs (road=1, grass=2, mud=3) for all path segments, making it the primary efficiency measure for cost-aware algorithms like UCS and A. ***Nodes Expanded*** *counts all positions explored during search execution, directly indicating algorithmic efficiency and memory usage—BFS typically expands the most nodes while A* is generally most efficient. **Execution Time** is measured server-side using Python's time module, capturing actual computational duration in seconds for fair comparison between algorithms. **Path Length** simply counts the number of steps in the solution, providing a basic effectiveness measure particularly relevant for BFS which finds shortest paths in unweighted grids. These metrics together provide comprehensive insights into each algorithm's trade-offs between optimality, efficiency, and computational requirements across different environmental conditions.

A* SEARCH

	PATH COST	NODES	EXECUTION TIME	PATH LENGTH
SMALL	14	37	0.0013s	15
MEDIUM	24	57	0.0008s	25
DYNAMIC	NA	176	0.0093s	NA

Breadth-first search(BFS)

	PATH COST	NODES EXTENDED	EXECUTION TIME	PATH LENGTH
SMALL	18	84	0.0009s	15
MEDIUM	36	166	0.0008S	25
DYNAMIC	NA	170	0.0006s	NA

CONCLUSION:

The Autonomous Delivery Agent Simulator successfully demonstrates fundamental pathfinding algorithms within an interactive, visually intuitive web interface. The implementation effectively highlights each algorithm's characteristics: BFS guarantees shortest paths in unweighted environments, UCS ensures cost-optimal solutions, A* balances efficiency with optimality through heuristic guidance, and Local Search provides a lightweight approach suitable for dynamic environments. The system's modular architecture separates visualization from computational logic, creating an extensible foundation for adding future algorithms or enhanced features. By providing immediate visual feedback coupled with quantitative performance metrics, the simulator serves as an effective educational tool for understanding search algorithm behaviour, performance trade-offs, and practical implementation considerations in varied terrain and obstacle configurations