Assignment 1 – Problem Solving

Sunain Mushtaq

S224089221

SIT215 – Computational Intelligence

Submission Date : 12/04/2025

Tasks	Completed (✓)	Notes (Optional)
Task 1: Environment creation and problem formation Task (P/C)	✓	
2: Basic navigation implementation (P/C)	✓	
Task 3: Enhance environment and heuristics comparison (D)	✓	
Task 4: Performance enhancement with an alternative Algorithm(HD)	✓	
Task 5: Graphical user interface (Bonus Task)	✓	GUI and demo video complete

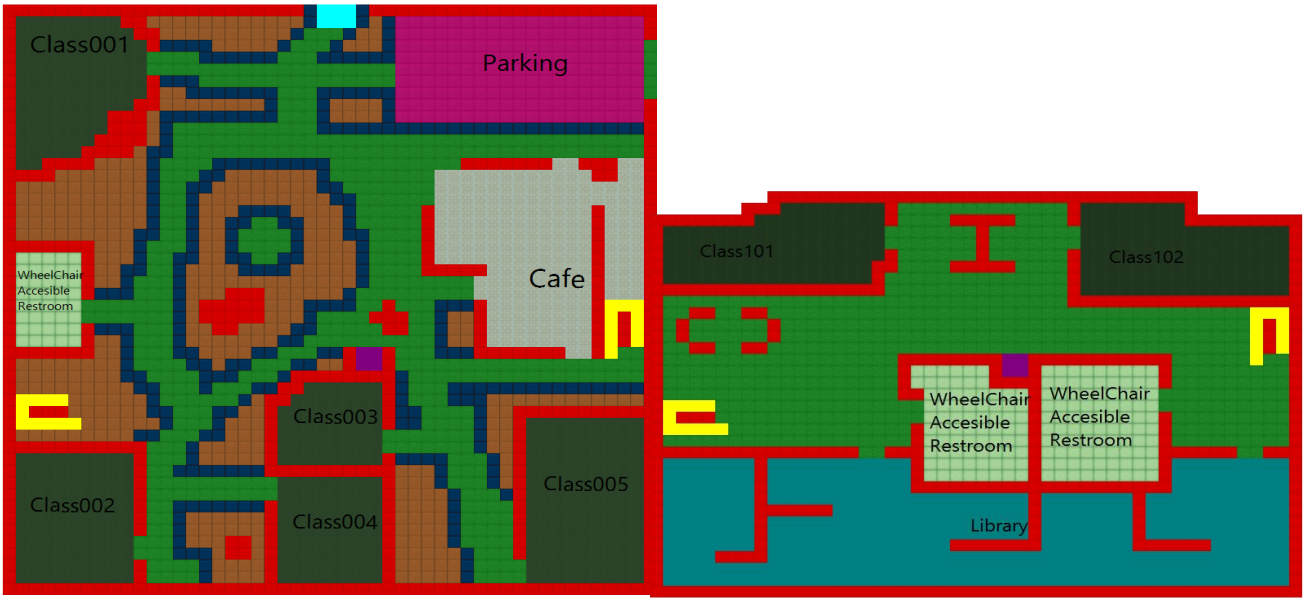
PROBLEM FORMATION

Navigation systems such as Google Maps provide basic pathfinding functionalities; however, they often lack vital support for accessibility-aware routing. For wheelchair users, obstacles such as **slopes**, **kerbs**, and varying **surface friction** can significantly hinder mobility and safety. This project aims to bridge that gap by designing a **wheelchair-accessible navigation system**, where environmental properties are encoded into the navigation logic and directly influence the feasibility and cost of travel paths.

Environment Overview

To simulate a realistic yet controlled environment, a custom map was created using **Tiled**, a 2D tile map editor. The map used in this project is:

- **Map Size:** 50 tiles wide × 50 tiles tall
- **Tile Size:** 16 × 16 pixels
- **Levels:** Two floors (Level 0 and Level 1)



Level 0 level1

The environment includes more than **150,000 interconnected path segments**, simulating a mid-scale campus-like structure. Each tile represents a region with specific **accessibility metadata** including **friction, elevation, and zone type**. Key regions include:

- Outdoor walking areas
- Multiple classrooms (Class001–005, 101–102)
- Campus library
- Cafe
- Ramps and lifts for elevation handling
- Designated accessible restrooms and parking zones

Tile-Based Representation and Properties

The environment is enriched using metadata assigned to individual tiles. The attributes reflect accessibility challenges that a wheelchair user might encounter. Here's a summary of tile types:

Zone	Location	Surface Friction	z-level	Use Case
Green	Classrooms, Cafes, Open areas	0.2	0	Smooth and accessible areas
Red	Walls/Barriers	10	0	Inaccessible
Orange	Uneven pavements	1.2	0	Mildly difficult surfaces
Blue	Kerbs	3	0.1	Small elevation barriers
Yellow	Ramps	1	Procedurally Generated	Smooth transition zones
Purple	Lifts	0.3	0	Used to simulate inter-floor navigation
Cyan	Main Entrance/Campus Entry	0.4	0	Access starting point
Magenta	Accessible Parking	0.6	0	Designated parking for wheelchair users
Lime	Wheelchair Accessible Restrooms	0.7	0	Bathroom facility
Teal	Campus Library (Upper floor access)	0.85	1	High-importance zone, elevated

Accessibility Simulation Logic

Accessibility features are modelled by combining **surface friction** (resistance to movement) and **elevation** (barriers like kerbs or ramps). The following rules were applied in code:

- **Friction values** influence the path cost: higher friction = harder to traverse.
- **Elevation modelling:**
    - Every **kerb** adds a **z-level of 0.1**.
    - **Ramps** increase height by **1** unit and have very high friction.
    - During map initialization, stair tiles dynamically accumulate elevation using the rule:
      each additional ramp step = +0.1 height
      This was implemented in code to simulate vertical climbing difficulty.

These properties influence how the agent evaluates a path, and which routes are deemed feasible or costly, making the problem highly aligned with **real-world wheelchair navigation constraints**.

Approach

Problem Modelling and Environment Structure

This project addresses the real-world problem of wheelchair navigation by formulating it as a **grid-based search problem**. The environment is loaded from a .tmx file created using Tiled, which includes metadata for each tile — such as zone, elevation (z-level), surface friction, and location.

Upon loading, the .tmx file is parsed and converted into a custom Map object. This object maintains:

- A dictionary of Level objects, each representing a different floor of the environment (e.g., level 0 and level 1).
- Each Level object contains a 2D **grid of Tile objects**, which represent the nodes in the search space.

Each Tile object contains attributes that define its characteristics and accessibility:

- **friction** – used to model movement cost
- **pos_x, pos_y** – spatial location within the grid
- **zone, location, height,** and **z-level** – environmental metadata affecting traversal feasibility

These properties allow the problem to be approached from an AI perspective: **nodes** represent the states, **edges** represent possible transitions (actions), and **pathfinding** is achieved by exploring the state space.

Problem and Agent Initialization

Within the Problem class:

- A Map object is initialized from the .tmx map file.
- Two agents (**a_agent** and **d_agent**) are created — one for A* and one for Dijkstra's algorithm.
- An Analyzer object is initialized to compute and evaluate paths using both algorithms.

Here, the **source tile** represents the agent's **initial state**, and the **destination tile** is the **goal state**. The Analyzer determines the sequence of actions required to transition from the source to the destination while minimizing total cost.

Environment Connectivity

The environment consists of **two grids (levels)** that simulate a multi-floor setting. These levels are interconnected by:

- A **pair of ramps** (yellow zone)
- An **elevator** (purple zone)

These special tiles appear in both level grids and allow **vertical transitions** between floors. The elevation metadata and tile type (zone) dictate whether a cross-level move is possible and at what cost.

Actions and Search Strategy

The core of the agent's decision-making lies in the get_neighbors() method (Lines 1300–1350 approx.). For any given Tile, this function:

- Checks **adjacent tiles** (up, down, left, right).
- Ensures tiles are **within map bounds**, **not walls (red zone)**, and that **elevation differences are reasonable** (≤ 0.15).
- Evaluates special transitions for **ramps and lifts**.
- Computes the **cost to move** to each neighbour, based on friction and elevation.

These neighbouring states are treated as **actions** in the AI model. They are **added to a priority queue**, and the algorithm continues exploring until it either finds the goal state or exhausts all options.Once a valid path to the destination is found, it is **traced back** using the **came_from** dictionary — forming a complete action sequence from source to goal.

Learning Integration

The architectural choices for agents, environment modeling, and search strategies were influenced by the following workshops and class resources:

- **Week 3:** *DFS and BFS Agent Foundation*
- **Week 4:** *State Space Search Formulation*
- **Week 5:** *A** and Heuristic Design in Agent-Based Navigation*

These foundational exercises were key in developing the modular structure of the Problem, Map, Agent, and Analyzer classes, as well as the overall CI-based approach.

Heuristics in A*

The A* algorithm was extended to work with **three different heuristics**, each simulating varying levels of spatial awareness and realism. These were implemented in the **Analyzer** class of the **Project.ipynb** file: (Lines 870-900)

1. Manhattan Distance

```
def heuristic_manhattan(self, a: "Problem.Tile", b: "Problem.Tile") -> float:
    """
    Compute the Manhattan distance between two tiles.
    """
    dx = abs(a.pos_x - b.pos_x)
    dy = abs(a.pos_y - b.pos_y)
    dz = abs(a.level - b.level)
    return dx + dy + dz
```

2. Euclidean Distance

```
def heuristic_euclidean(self, a: "Problem.Tile", b: "Problem.Tile") -> float:
        """
```

```
        Compute the Euclidean distance between two tiles.
        """
        dx = a.pos_x - b.pos_x
        dy = a.pos_y - b.pos_y
        dz = a.z_level - b.z_level
        return math.sqrt(dx * dx + dy * dy + dz * dz)
```

3. Chebyshev Distance

```
def heuristic_chebyshev(self, a: "Problem.Tile", b: "Problem.Tile") -> float:
        """
        Compute the Chebyshev distance between two tiles.
        """
        dx = abs(a.pos_x - b.pos_x)
        dy = abs(a.pos_y - b.pos_y)
        dz = abs(a.z_level - b.z_level)
        return max(dx, dy, dz)
```

Two of These heuristics were inspired by the principles discussed in the **Week 5 Workshop** and online lecture materials. The inclusion of **z-level** elevation in all heuristics enhances realism by penalizing paths with unnecessary elevation changes, making the A* algorithm accessibility-aware.

Implementation Overview

Map Parsing and Tile Initialization
The project begins by parsing a .tmx map file created in Tiled. Each tile from the map is dynamically converted into a Tile object containing essential properties such as friction, pos_x, pos_y, z_level, zone, and location.

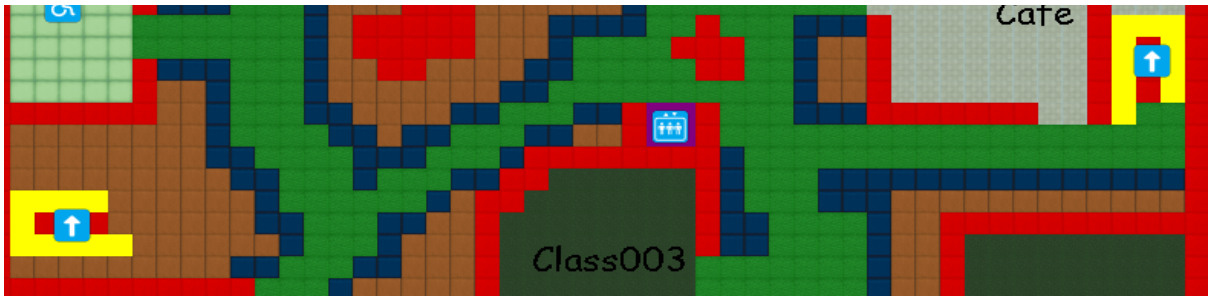
The map is structured as a Map object that holds a dictionary of Level objects (each representing a floor). Each Level object maintains a 2D grid of Tile objects.

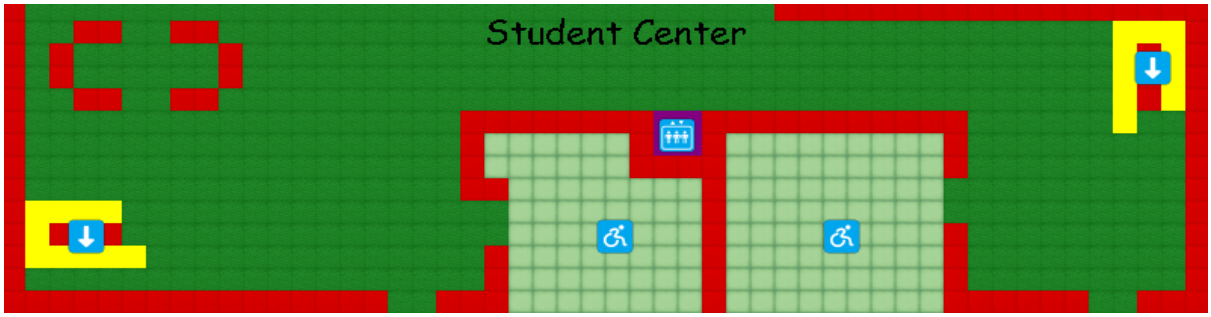
During initialization, elevation for ramps and stairs is automatically adjusted:

- In the method fix_stairs(): *Lines ~240–250*

```
inc_a = 1 / (len(stairs_a_indices) + 1)
for i, pos in enumerate(stairs_a_indices):
    tile = self.grid[pos[0]][pos[1]]
    if tile:
        tile.z_level = round((i + 1) * inc_a, 2)
```

- All tiles in Level 1 are initialized with z_level = 1.0, simulating the elevation difference between floors.





Problem Initialization (Non-GUI Mode)

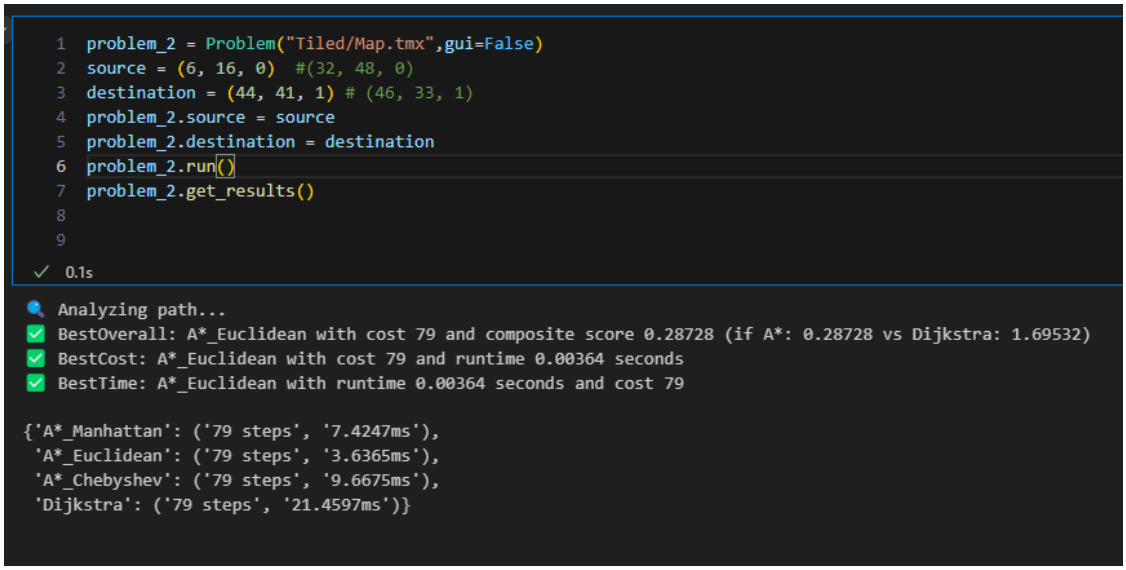
This mode focuses purely on computational logic, excluding graphical interaction. It's ideal for testing and performance benchmarking. When Problem(gui=False) is initialized, the Pygame interface is skipped, allowing a direct focus on logical path analysis.

```
problem = Problem(map_path="Tiled/Map.tmx", gui=False)
problem.source = (x1, y1, level1)
problem.destination = (x2, y2, level2)
problem.analyze()
```

Execution Flow

1. Source and Destination tiles are assigned.
2. analyze() triggers the Analyzer to compute optimal paths using A* and Dijkstra.
3. Analyzer.run_algorithms() executes each strategy and stores results.
4. Output is accessed via get_results() or get_top_results().

No Pygame windows are launched, making this pathway script-friendly for automation or testing environments.



```
problem_2 = Problem("Tiled/Map.tmx",gui=False)
source = (6, 16, 0)  #(32, 48, 0)
destination = (44, 41, 1) # (46, 33, 1)
problem_2.source = source
problem_2.destination = destination
problem_2.run()
problem_2.get_results()
```

0.1s

```
Analyzing path...
BestOverall: A*_Euclidean with cost 79 and composite score 0.28728 (if A*: 0.28728 vs Dijkstra: 1.69532)
BestCost: A*_Euclidean with cost 79 and runtime 0.00364 seconds
BestTime: A*_Euclidean with runtime 0.00364 seconds and cost 79

{'A*_Manhattan': ('79 steps', '7.4247ms'),
 'A*_Euclidean': ('79 steps', '3.6365ms'),
 'A*_Chebyshev': ('79 steps', '9.6675ms'),
 'Dijkstra': ('79 steps', '21.4597ms')}
```

Analyzer Class and Path Calculation

The Analyzer class performs A* and Dijkstra pathfinding. It uses a priority queue to explore the lowest-cost paths and traces back the path using the came_from dictionary. *Lines ~620–940*

```
for neighbor, move_cost in self.get_neighbors(current, self.map, current.level,
use_terrain_cost=True):
    new_cost = cost_so_far[current] + move_cost
    if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
        cost_so_far[neighbor] = new_cost
        priority = new_cost + self.map.problem.get_heuristic_multiplier() * heuristic_func(neighbor,
goal)  # A*
        frontier.append((neighbor, priority))
        came_from[neighbor] = current
```

In A*, cost is computed as:

$$f(n) = g(n) + h(n)$$

Where:

- g(n) is the accumulated cost from the start node.
- h(n) is the estimated cost to the goal (heuristic).

Lines ~681,701

```
self.results[f"A*_{name}"] = (path, cost, runtime)
self.results["Dijkstra"] = (dijkstra_path, cost, dk_runtime)
```

Lines ~928–932

```
while tile != start:
    path.append((tile.pos_x, tile.pos_y, tile.level))
    tile = came_from.get(tile)
    if tile is None:
        return None
```

```
✅ BestOverall: A*_Euclidean with cost 79 and composite score 0.28728 (if A*: 0.28728 vs Dijkstra: 1.69532)
```

D-Level Enhancements: Heuristics & Environmental Constraints

The system evaluates over 150,000 tile segments, requiring robust cost management. Each move considers:

1. **Surface Friction** – Higher values increase resistance.
2. **Elevation (z_level)** – Transition difficulty between tiles.

This dynamic cost calculation allows agents to avoid difficult terrain in favor of smoother, flatter paths.

Lines ~40–60

```
def calculate_cost(source, neighbor, use_terrain_cost):
    elevation_diff = abs(neighbor.z_level - source.z_level)
    if use_terrain_cost:
        return (source.friction + neighbor.friction) / 2 + elevation_diff + 1
    else:
        return elevation_diff + 1
```

HD-Level Enhancements: GUI & Multi-State Interaction

Agent Class Integration (GUI Context)

While the Analyzer handles logic, the Agent class plays a vital role in the visual simulation within the GUI. After path analysis is completed, two agents are animated across the map:

- Blue agent → follows the path generated by A*
- Yellow agent → follows the path generated by Dijkstra

Lines ~560–570

```
def set_path(self, path):
    self.path = path
    self.path_index = 0
    self.trail = []


def update(self):
    if self.path_index < len(self.path):
        self.x, self.y, self.level = self.path[self.path_index]
        self.trail.append((self.x, self.y, self.level))
        self.path_index += 1
```

During the VisualizeResult state, each agent uses its draw_path() method to animate movement. The draw_trail() method ensures the path remains visible throughout the visualization cycle.

GUI State Machine and Interaction

The GUI system is implemented using a state machine defined in State and StateType. The game transitions through the following states:

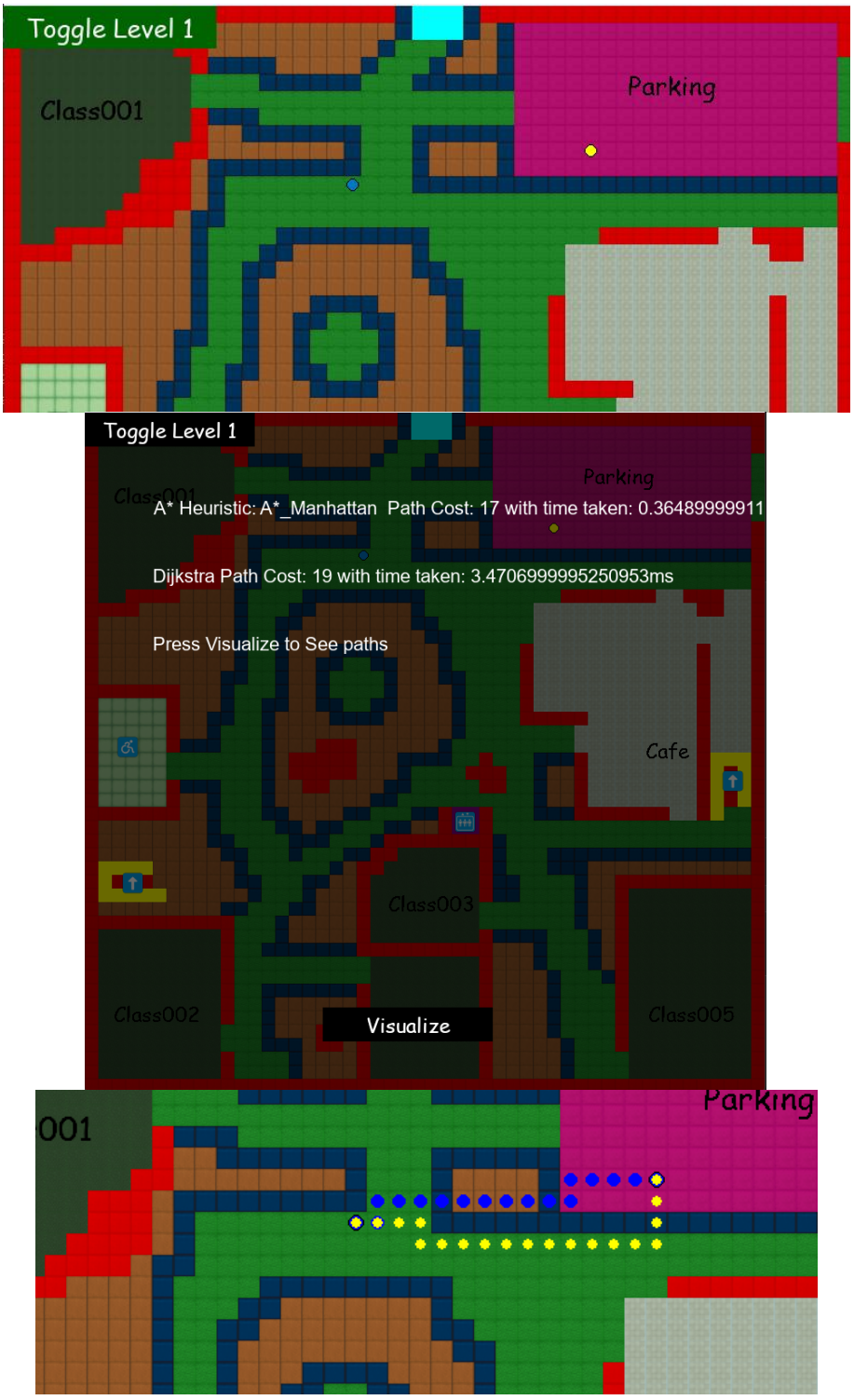
1. PressStart
2. SelectStarting
3. SelectDestination
4. Analyze
5. ShowResult
6. VisualizeResult

Transitions occur through key presses or UI events:

- Enter: Confirms source/destination and advances states
- Esc: Returns to main screen from visualization
- Visualize: Starts agent animation
- Toggle Button: Always accessible to switch between map levels

GUI Workflow

1. User clicks a tile to select Source → press Enter
2. Selects Destination → press Enter
3. System analyzes the path → displays cost & time for A* and Dijkstra
4. Clicking Visualize animates both agents with distinct colors
5. User can loop back using Esc to restart

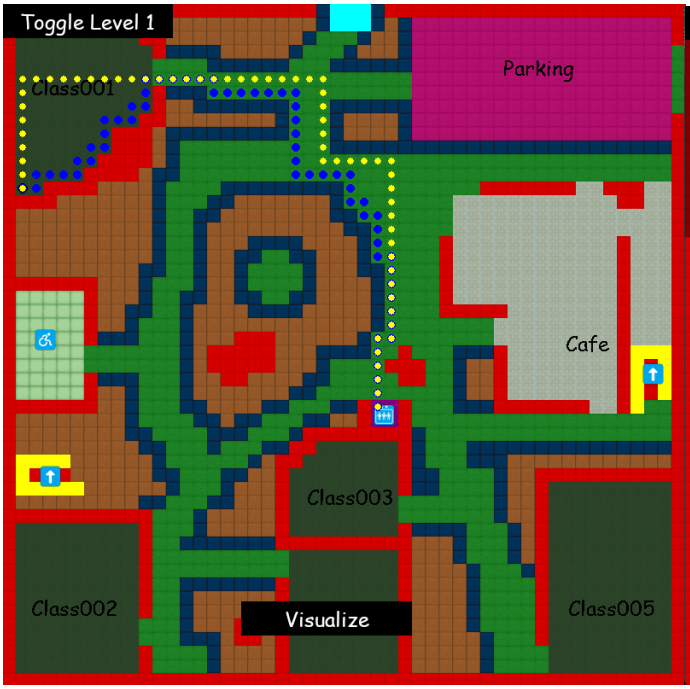

Toggle Level 1
Class001
Parking
Toggle Level 1
Parking
A* Heuristic: A*_Manhattan Path Cost: 17 with time taken: 0.36489999911
Dijkstra Path Cost: 19 with time taken: 3.4706999995250953ms
Press Visualize to See paths
Cafe
Class003
Class002
Visualize
Class005
001
Parking

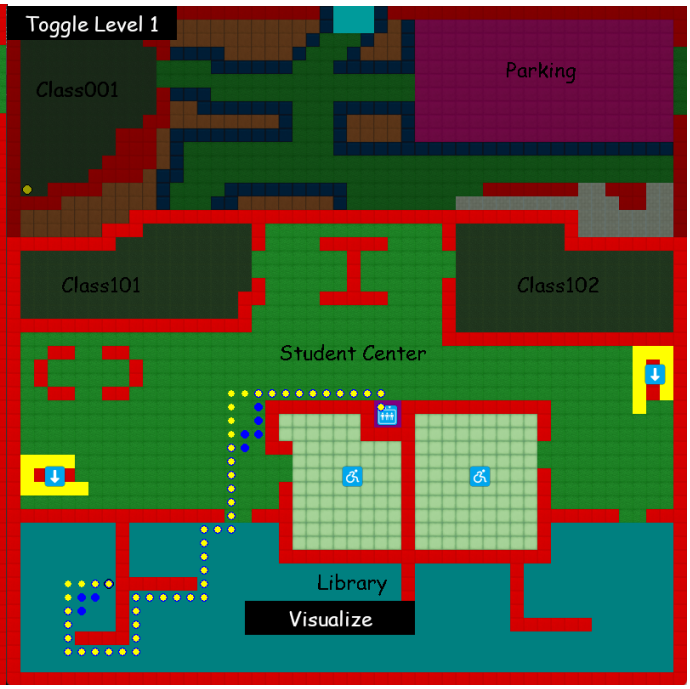
Results and Evaluation

Non-Distinction Level (P/C): Successful Pathfinding Run

A successful execution of the wheelchair navigation system demonstrates that both A* and Dijkstra algorithms are capable of computing accessible paths from a given source to destination tile. The selected environment includes a two-level campus layout connected via stairs and a lift.

The images below shows a sample run where both agents successfully compute and animate their respective paths across the terrain.





Level 0 level1

Results:

```
🔍 Analyzing path...
✅ BestOverall: A*_Euclidean with cost 113 and composite score 1.83927 (if A*: 1.83927 vs Dijkstra: 2.72846)
✅ BestCost: A*_Euclidean with cost 113 and runtime 0.01628 seconds
✅ BestTime: A*_Euclidean with runtime 0.01628 seconds and cost 113
['A* Heuristic: A*_Euclidean  Path Cost: 113 with time taken: 16.276700000162236ms', 'Dijkstra Path Cost: 113 with time taken: 24.1456999999280623ms']
Visualizing
A*: blue
Dijkstra: Yellow
```

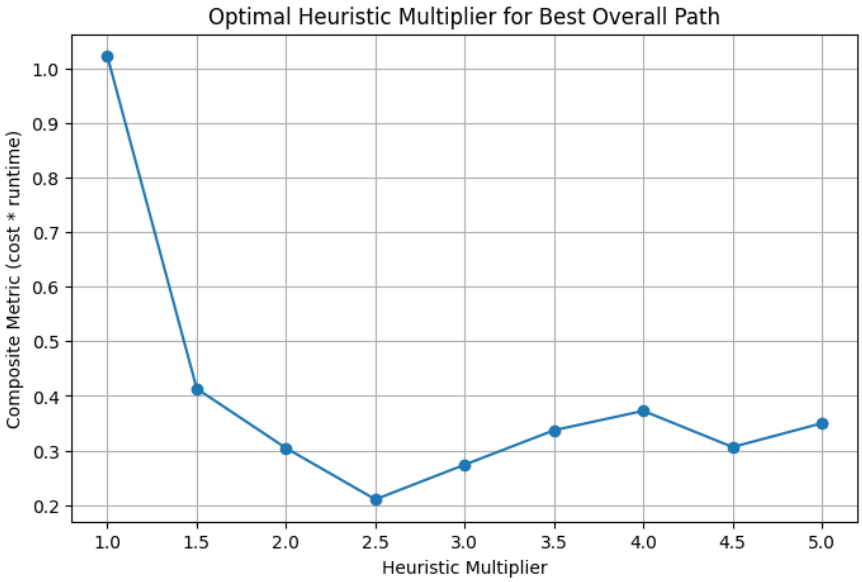
Both agents computed a valid path from *Class001* on Level 0 to the *Library* on Level 1 using the elevator. The resulting paths had identical costs, but A* achieved the result in 16 ms, whereas Dijkstra took 24 ms. This highlights A*'s advantage in terms of computational efficiency.

Distinction Level (D): Enhanced Analysis and Comparison

To evaluate algorithmic performance in dynamic and realistic environments, we conducted multiple experiments using varied heuristics and randomized test cases.

Heuristic Multiplier Tuning

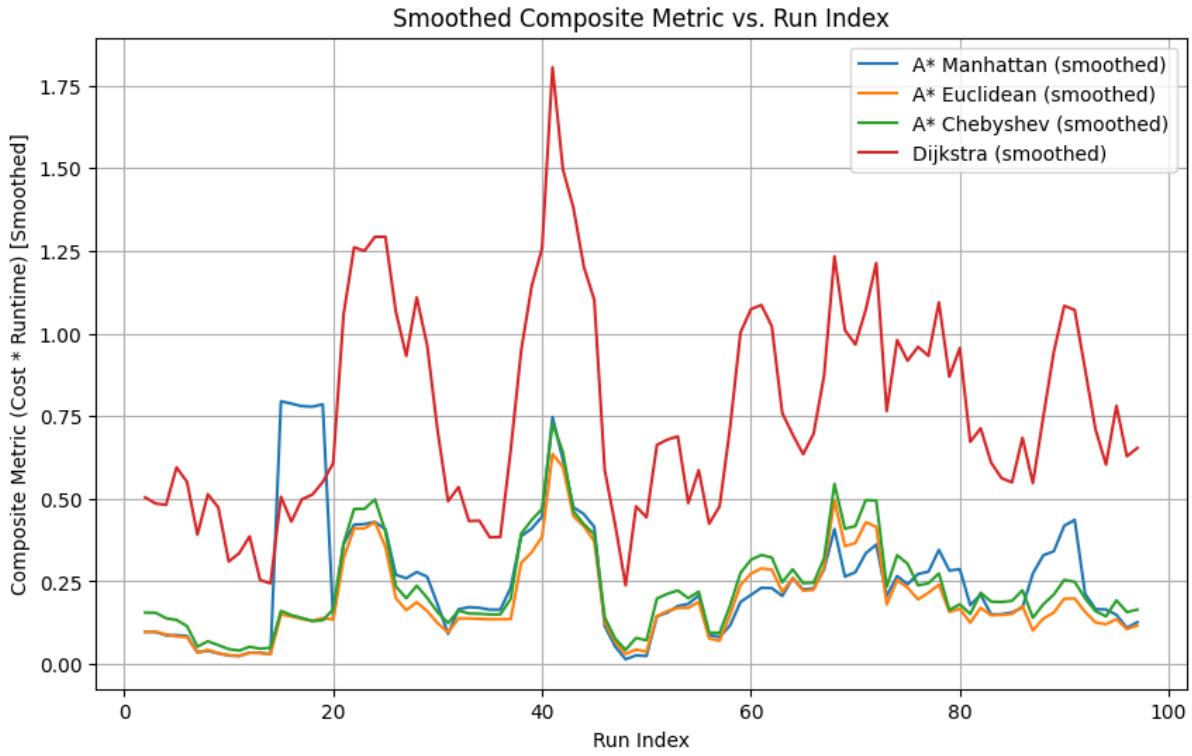
To optimize A*'s performance, the heuristic multiplier was varied from 1.0 to 5.0 in increments of 0.5. The goal was to find the multiplier that minimizes the composite metric (path cost × runtime).



This plot evaluates the impact of heuristic strength (multiplier) on A*'s composite performance.
Observation: A multiplier around **2.5 to 3.5** yields the best balance between speed and accuracy. Extremely high or low multipliers cause either runtime spikes or poor path choices.

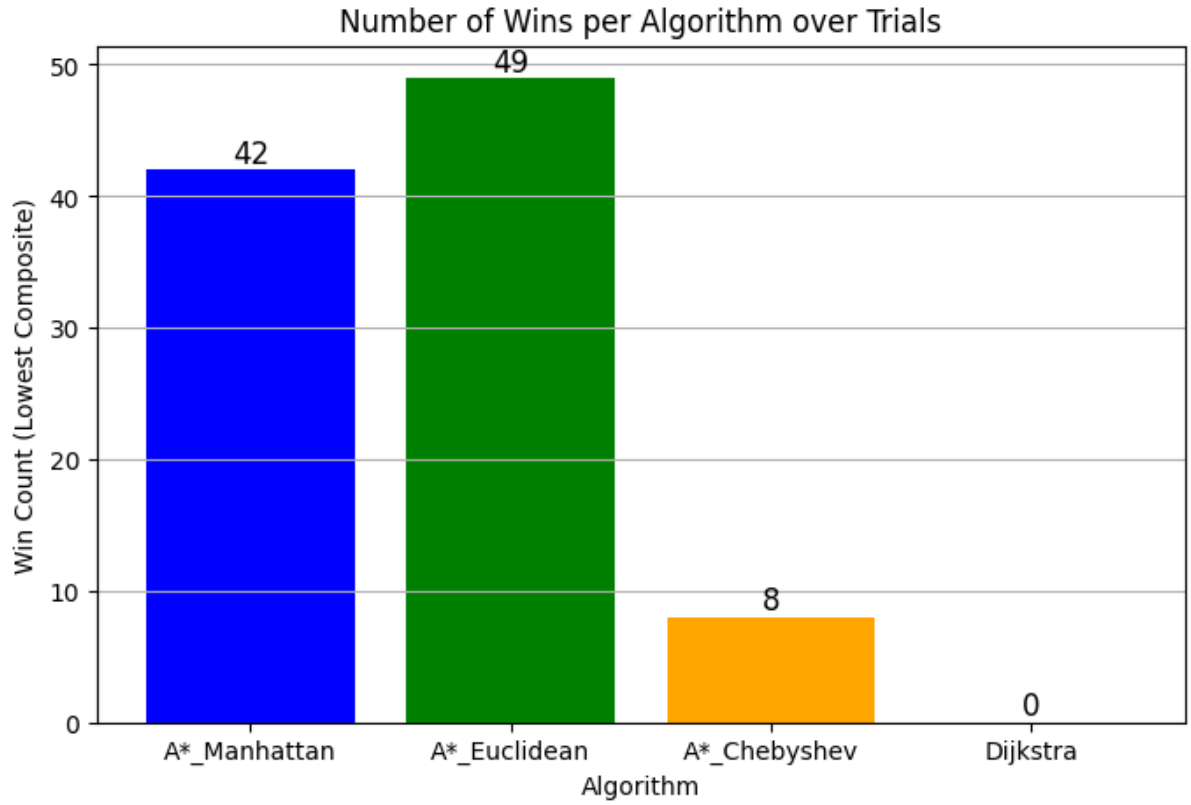
Randomized Trials (100 Runs)

To validate algorithm robustness, we conducted 100 randomized source-destination trials across a complex environment with over 150,000 traversable segments. Each run calculated cost, runtime, and the composite performance metric.



This line graph compares the smoothed composite metric (cost × runtime) over 100 randomized trials.

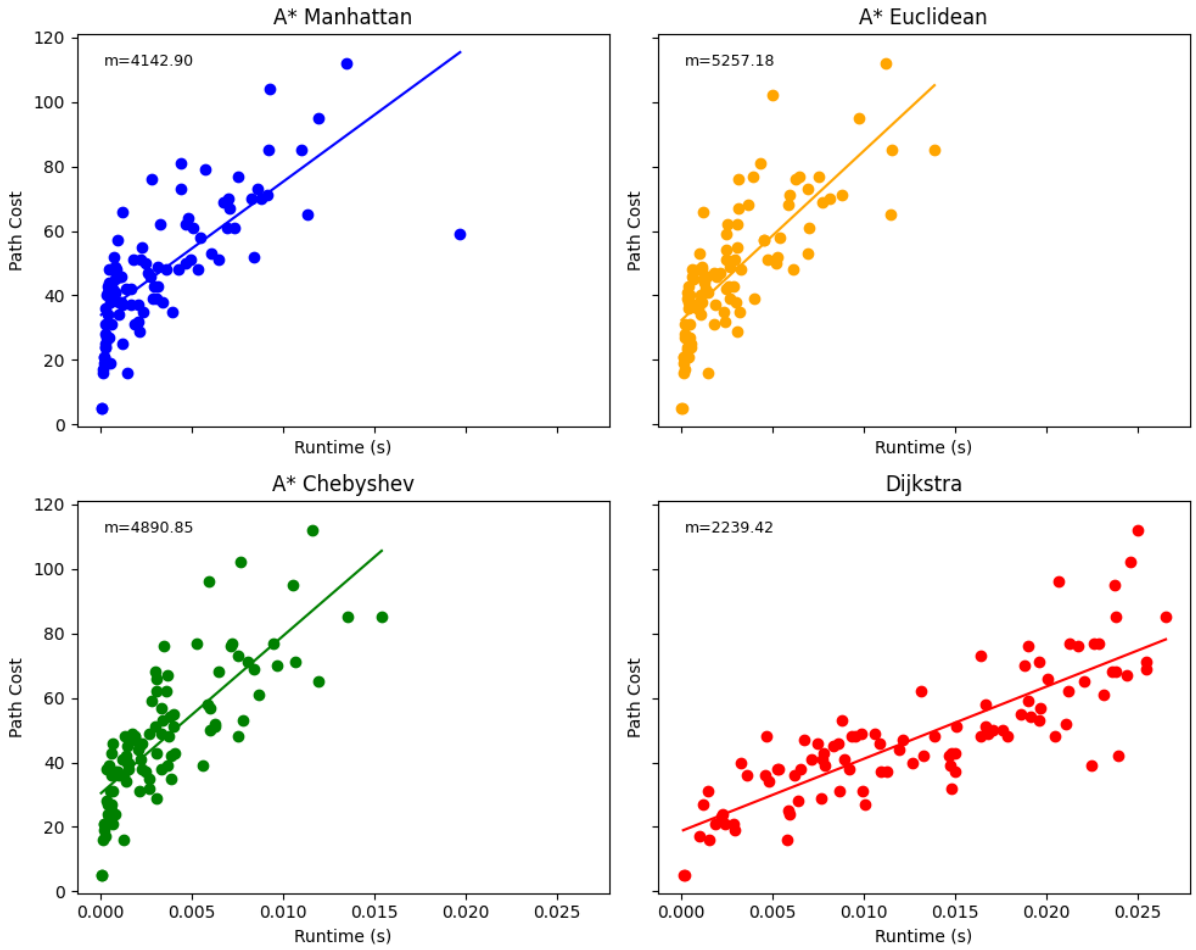
Observation: A* Manhattan, Euclidean, and Chebyshev consistently outperform Dijkstra, which shows significantly higher values throughout. This highlights the efficiency of heuristic-driven approaches, especially in dynamic maps.



This bar chart summarizes which algorithm had the lowest composite score in each trial.
***Observation**: A* Euclidean won 49 out of 100 trials, closely followed by A* Manhattan. Dijkstra, despite its accuracy, failed to win any trial due to longer runtimes, confirming the trade-off between optimality and efficiency.*

High Distinction Level (HD): Runtime-Cost Relationship

To further evaluate algorithmic behaviour, a regression analysis was performed on runtime vs cost.



Insights:

- A* Manhattan and A* Chebyshev showed strong linear relationships.
- A* Euclidean had more variance, possibly due to irregular elevation shifts.
- Dijkstra, with no heuristic, consistently remained on the upper edge in runtime.

These findings support the use of terrain-aware heuristics in pathfinding over large, realistic grids. Combined with GUI-state transitions and dynamic map rendering, this evaluation validates the effectiveness of heuristic-driven planning for accessible navigation.

Conclusion

This project set out to explore intelligent pathfinding in a wheelchair-accessible environment using a layered tile-based map. The core objective was to evaluate the performance of various classical algorithms—including A* (with multiple heuristics) and Dijkstra—in computing optimal paths under real-world constraints like elevation, friction, and accessibility zones. Through extensive implementation, heuristic tuning, and randomized testing across 100 unique trials, we demonstrated that heuristic-based algorithms like A* can significantly outperform uninformed algorithms such as Dijkstra, particularly in balancing runtime efficiency and path cost. The integration of GUI-based interaction, dynamic map rendering, and agent-based simulations enriched the system's usability and made evaluation intuitive. A critical takeaway from the analysis was the identification of an ideal heuristic multiplier range (approximately 2.5 to 3.5), which yielded the lowest composite metric, striking a balance between computational speed and result optimality.

Among the evaluated algorithms, A* with Euclidean heuristic emerged as the most consistently efficient in terms of the composite metric (cost × runtime), winning 49 out of 100 trials. A* Manhattan closely followed, proving to be a dependable heuristic in structured grid-based environments. Chebyshev, while functional, performed less consistently, and Dijkstra, though effective in finding the lowest-cost path in some cases, was ultimately hindered by its exhaustive search nature and higher runtime overhead. Regression analysis revealed a linear correlation between runtime and cost for all A* variants, with Dijkstra occupying the upper end of the runtime spectrum without significant cost reduction. These findings reinforce the importance of informed heuristics in real-time decision-making systems. Overall, the extensive data-driven evaluation provided meaningful insights into algorithm scalability, real-world applicability, and the practical trade-offs between speed and precision in terrain-aware path planning.

I would like to acknowledge the invaluable support received throughout the duration of this project. Special thanks go to our tutors and workshop instructors for providing guidance, feedback, and foundational code snippets that were instrumental during early development stages. I also appreciate the encouragement and collaboration offered by my peers, whose suggestions often helped refine the system's usability and visualization components. Additionally, tools like ChatGPT were immensely helpful in conceptual brainstorming, debugging code, and formulating structured explanations for documentation. The combination of these resources played a crucial role in overcoming challenges related to algorithmic fine-tuning and performance optimization. From a learning standpoint, this project has not only deepened my understanding of heuristic search algorithms but also honed my skills in software design, data visualization, and accessibility-aware system development. Most importantly, it highlighted the value of iterative experimentation, the power of well-tuned heuristics, and the potential of computational intelligence in creating inclusive, real-world solutions.