Department of Electronic & Electrical Engineering

UNIVERSITY OF BATH

Towards autonomous exploration and object recognition with a LEGO Spike robot

Tim Riekeles

Candidate number: 11251

Robotics Engineering

MEng Final Project Report

2024/2025

11917 words [245 in abstract]

Ethical approval was provided by the University of Bath

"I certify that I have read and understood the entry in the Student Handbook for the Department of Electronic and Electrical Engineering on Cheating and Plagiarism and that all material in this assignment is my own work, except where I have indicated with appropriate references. An electronic copy of this work has been uploaded to Moodle to allow submission to Turnitin for Plagiarism Detection."

Signature: Tim Riekeles

Supervisor: Uriel Martinez Hernandez

Assessor: Fang Duan

Table of Content

TABLE OF ACRONYMS 3

ABSTRACT 4

INTRODUCTION 4

LITERATURE REVIEW 5

- REACTIVE BEHAVIOUR 5
  - BRAITENBERG 5
- CLASSIC PATHFINDING ALGORITHMS 6
  - A* 6
  - DIJKSTRA 7
- LEARNING BASED ALGORITHMS 8
  - SOFT ACTOR-CRITIC 8
  - CONVOLUTIONAL NEURAL NETWORKS 9
- ANIMAL INSPIRED ALGORITHMS 9
  - ANT COLONY OPTIMISATION 9
- DISCUSSION ON RELEVANT LITERATURE 10

METHODS 11

- BRAITENBERG-INSPIRED OBSTACLE AVOIDANCE ROBOT 11
- LINE FOLLOWING ROBOT 12
- A* ALGORITHM 12
- DIJKSTRA'S ALGORITHM 15
- CONVOLUTIONAL NEURAL NETWORKS 16
- ANT COLONY OPTIMISATION 18
- PIPELINE IMPLEMENTATION 19

RESULTS 21

- STRUCTURED GRIDS 22
  - ONE CELL WIDE PATHS – A*, DIJKSTRA AND ACO 22
  - ONE CELL WIDE PATHS – CNN 23
  - TWO CELL WIDE PATHS 25
- UNSTRUCTURED GRIDS 26
  - A* AND DIJKSTRA 26
  - CNN 27
- REAL WORLD MAPS 29

DISCUSSION 32
FUTURE WORK 33
CONCLUSION 34
PROJECT REVIEW 35
ACKNOWLEDGEMENTS 36
REFERENCES 36
APPENDICES 37
APPENDIX I - PROJECT COSTING 37
APPENDIX II - TABLES 37
APPENDIX III - PYTHON IMPLEMENTATION 39
APPENDIX IV - CNN HEATMAPS OF GENERATED MAZES 57
APPENDIX V - STRUCTURED GRIDS 59
APPENDIX VI - STRUCTURED DOUBLE CELL PATH GRIDS 63
APPENDIX VII - UNSTRUCTURED GRIDS AND CORRESPONDING HEATMAPS 65
APPENDIX VIII - REAL-WORLD GRIDS 77

TABLE OF ACRONYMS

ACRONYM	DEFINITION
AGV	Automated Guided Vehicle
SAC	Soft Actor Critic
CNN	Convolutional Neural Network
ACO	Ant Colony Optimisation
ROS	Robot Operating System
RGB	Red, Green, Blue
RL	Reinforcement Learning
ReLU	Rectified Linear Unit
VS	Visual Studio (Code)
SLAM	Simultaneous Localization and Mapping

ABSTRACT

The rise of automation in industrial and logistical settings has increased the demand for efficient and reliable navigation systems for Automated Guided Vehicles (AGVs). These systems face the challenge of balancing computational efficiency, adaptability to dynamic environments, and real-world feasibility, particularly on resource-constrained platforms such as the LEGO SPIKE Prime robot used in this study. Smaller projects like LEGO-based robots provide a valuable testbed for experimentation and learning before implementing AGV technologies at scale. This project aimed to develop an interactive, automated LEGO-based AGV that simulates real-world warehouse operations, controlled through voice commands and featuring obstacle avoidance and user-drawn environments. The project implemented and evaluated a range of autonomous navigation algorithms, including Braitenberg-inspired obstacle avoidance, A*, Dijkstra's Algorithm, Convolutional Neural Networks (CNNs), and Ant Colony Optimisation (ACO). These methods were tested on structured, unstructured, and real-world grid environments. The evaluation demonstrated that while ACO and CNNs faced limitations in computational efficiency and prediction reliability for real-time navigation, A* and Dijkstra's algorithm consistently provided optimal paths and proved most suitable for the constrained LEGO platform. A* balanced optimality and efficiency, while Dijkstra offered more stable execution times but was less efficient on large, sparse maps. The project successfully met its objectives, including colour classification, database connectivity, navigation algorithm integration, and voice command integration using OpenAI's Whisper model. By bridging the gap between theoretical research and practical deployment on accessible educational robots, this work contributes a framework for selecting and optimizing navigation algorithms for constrained AGV platforms [245 words (250 max)].

INTRODUCTION

The rise of automation in industrial and logistical settings has heightened the demand for efficient and reliable navigation systems for Automated Guided Vehicles (AGVs). These systems must balance computational efficiency, adaptability to dynamic environments, and real-world feasibility - a challenge that has spurred extensive research into diverse pathfinding algorithms. While classical methods like A* and Dijkstra's algorithm guarantee optimal paths in static environments, biologically inspired approaches such as Ant Colony Optimization (ACO) and learning-based techniques like Convolutional Neural Networks (CNNs) offer adaptability at the cost of computational complexity. However, the practical deployment of these algorithms on resource-constrained platforms, such as the LEGO SPIKE Prime robot used in this study, remains underexplored in the literature.

Before implementing AGV technologies at scale in large, industrial-grade vehicles, smaller projects like LEGO-based robots provide a valuable testbed for experimentation and learning. These smaller platforms allow for rapid prototyping, cost-effective trials, and hands-on exploration of complex navigation strategies without the significant financial and technical barriers associated with full-scale AGV systems. This approach enables researchers to refine algorithms, develop effective user interfaces, and tackle real-world challenges like dynamic obstacle avoidance, all within a manageable scope.

The project was conducted in collaboration with a company, focusing on developing an interactive training tool to demonstrate AGV navigation in warehouse digitization. By integrating voice commands, dynamic obstacle avoidance, and user-drawn environments, the system highlights the trade-offs between algorithmic sophistication and practical implementation.

The aim of this project is to develop an interactive, automated LEGO-based AGV that simulates real-world warehouse operations. The AGV is designed to detect and identify coloured LEGO blocks and record their locations in a database. Users can control the AGV through voice commands, enabling it to navigate autonomously and return to previously identified blocks. In keeping with LEGO's reputation for simplicity and accessibility, the final system is intended to be user-friendly and intuitive, making it well-suited for client demonstrations and training applications.

This project has following objectives:

1. **Colour Classification:** Develop a colour classification system using the LEGO SPIKE colour sensor, achieving an accuracy of at least 95%
2. **Database Connectivity:** Establish reliable and real-time communication with a database for logging and retrieving object locations during AGV operation
3. **Navigation Algorithm Integration:** Implement and evaluate a range of autonomous navigation algorithms to simulate real-world logistics scenarios. These algorithms utilize multimodal sensor input for environmental awareness, with motor control acting as the primary means of executing navigation decisions
4. **Voice Command Integration:** Implement voice command functionality via a microphone to enable hands-free interaction. The system should accurately interpret navigation and retrieval commands with a minimum accuracy of 90%.

The thesis is structured as follows:

- Literature Review: Categorizes navigation techniques and their trade-offs
- Methods: Details the implementation of each algorithm on the LEGO platform
- Results: Presents quantitative comparisons across grid types (structured, unstructured, real-world)
- Discussion: Analyzes findings and proposes future directions

By bridging the gap between theoretical research and practical deployment, this work contributes a framework for selecting and optimizing navigation algorithms for constrained AGV platforms.

LITERATURE REVIEW

Robotic navigation forms a foundational pillar for mobile autonomy, with increasing demands from industrial, logistical, and service applications. The capability of a robot to move safely and efficiently within dynamic environments is inherently tied to the quality of its navigation strategies. Accordingly, navigation algorithms can be categorised based on the degree of environmental knowledge and adaptivity they require (Illustrated in Figure 1).

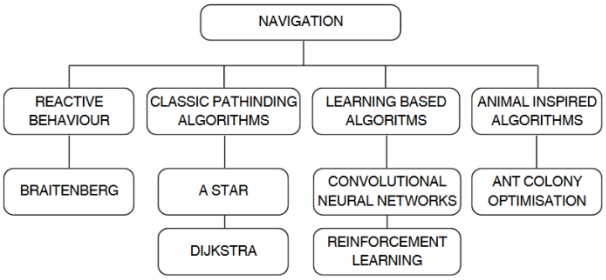


Figure 1: Taxonomy of navigation techniques covered in literature review

Reactive Methods (e.g., Braitenberg Vehicles) rely solely on local sensor data without mapping the environment. Classic Pathfinding Algorithms (e.g., A*, Dijkstra) assume a known static environment and focus on optimal route computation. Learning-Based Methods (e.g., SAC, CNNs) learn navigation strategies from environmental interaction or data. Biologically Inspired Methods (e.g., Ant Colony Optimisation) mimic natural processes for probabilistic path discovery.

This review systematically examines each category, evaluating strengths, limitations, and suitability for practical robotic deployment.

REACTIVE BEHAVIOUR

BRAITENBERG

One of the foundational examples of reactive behaviour is found in Braitenberg vehicles 2 and 3 (Shown in Figure 2). These vehicles illustrate simple yet effective sensor-motor coupling: Vehicles 2a and 3a employ direct-connection, where each sensor control the motor on the same side, while models 2b and 3b use cross-connection, where each sensor controls the opposite motor. A positive connection (‘+’) implies increased sensor activation results in faster motor rotation, while negative connection (‘-’) results in slower rotation. The direction of movement depends on this configuration and the location of a stimulus source (denoted ‘s’ in Figure 2). For

example, vehicles 2a and 3b move away from the source, while 2b and 3a approach it.

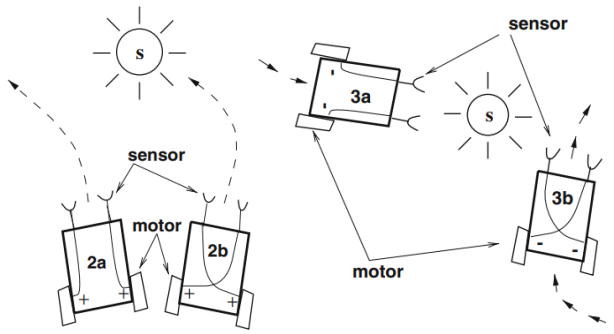


Figure 2: Braitenberg Vehicles 2 and 3 in the vicinity of a source [1]

In the realm of navigation and obstacle avoidance, 2a is particularly useful for evading threats from the rear: as a rear sensor detects proximity, the same-side motor accelerates, causing the robot to turn away from the threat. Meanwhile, 3b slows the opposite side motor and turning away from the obstacle – making it suitable for frontal and lateral obstacle avoidance [1].

Despite their simplicity, Braitenberg vehicles have key limitations. Their purely reactive design often results in inefficient and unpredictable behaviour, especially in cluttered environments. They lack planning capabilities and broader environmental awareness, making them unsuitable for complex navigation tasks requiring goal orientation, obstacle mapping, or path optimisation. As such, they are best suited for basic reactive functions, rather than advanced autonomous mobility [1].

CLASSIC PATHFINDING ALGORITHMS

A*

The A* algorithm is a widely used path-finding technique for AGV navigation due to balance between finding the optimal path and computational efficiency. It uses an evaluation function defined as:

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

Where is $g(n)$ the actual cost from the start node to the current node n, and $h(n)$ is the heuristic estimate of the cost from node n to the goal. Common heuristic choices are Manhattan distance for cardinal (four-directional) movement and diagonal distance for eight-directional movement (Figure 3), where cardinal moves have a cost of 1 and diagonal moves have a cost of $\sqrt{2}$ [2] [3].

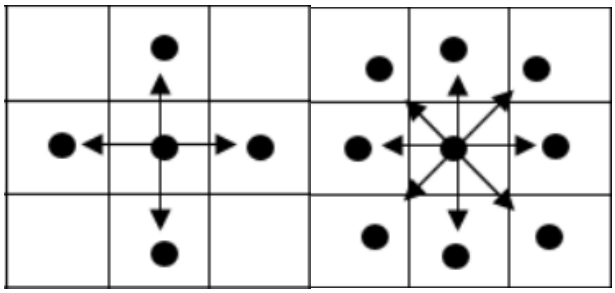


Figure 3: four-directional vs eight-directional movement [3]

A* employs two sets: The open set contains nodes to be explored - prioritising those with lowest cost - and a closed set to prevents revisiting already processed nodes, optimising efficiency by reducing redundant calculations [2] [3].

Several enhancements have been proposed to improve A*'s practical performance: Turn penalties can reduce erratic motion by penalising unnecessary turns, resulting in an overall smoother path (Figure 4) [2]. Bezier curve smoothing ensures kinematic feasibility and smooth path transitions (see Figure 5) [2]. *D Lite**, a dynamic variant of A*, enables on-the-fly path re-planning by updating only the affected segments when obstacles are discovered, improving its utility in real-time AGV environments [4].

Figure 4: Avoiding unnecessary turns through turn penalising (original path in black with optimized turns in red) [2]

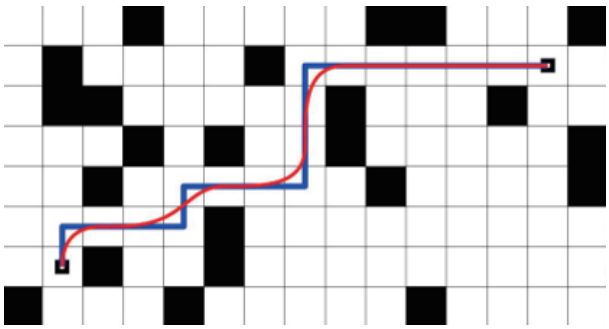


Figure 5: Path Smoothing Process using Bezier curves [2]

An animation illustrating how the A* algorithm traverses the search space is available at: https://learn.saylor.org/pluginfile.php/6824350/mod_page/content/5/Astarpathfinding.gif?time=1728403603187. Figure 6 presents a representative screenshot captured from this animation, depicting the fully solved grid and the path identified by the algorithm.

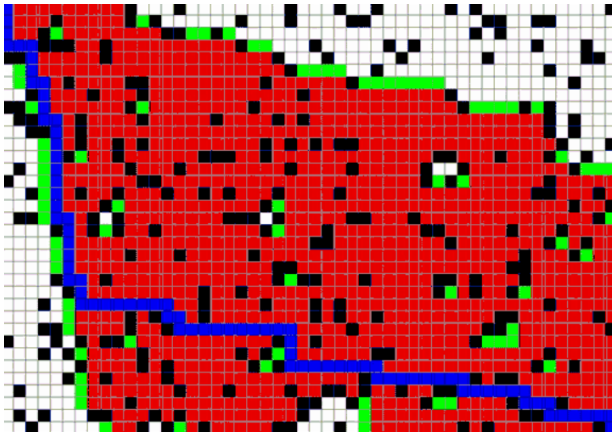


Figure 6: Screenshot captured from A animation, depicting the fully solved grid and the path identified by the algorithm [5]*

While A* has been extensively studied in AGV navigation, a notable limitation in the literature is the lack of real-world testing on physical hardware. Many studies focus on simulations within ROS (Robot Operating System), leaving a gap in understanding its performance in practical AGV implementations. Further work is needed to integrate A* with AI techniques to increase adaptability in dynamic, real-world scenarios [2] [3] [4].

DIJKSTRA

Dijkstra's algorithm calculates the shortest path from a source to all other reachable nodes in a weighted graph (example graph shown in Figure 7). For AGVs this enables efficient route planning across a mapped warehouse. The algorithms works by assigning a distance of zero to a source node and infinity to all others (referred to as vertices). All vertices are divided into two sets: one containing vertices for which the shortest path so far has been found and the other containing the remaining vertices. The algorithm iteratively selects the closest unvisited node and updates distances to its neighbours. Once a node is visited, it is not revisited, and the process continues until all nodes are evaluated [6] [7] [8].

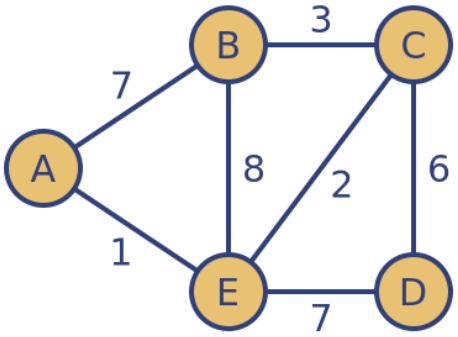


Figure 7: Example of a Dijkstra weighted graph [7]

Table 2 in Appendix II and Figure 8 illustrate the stepwise progression of the Dijkstra algorithm applied to the sample graph 7.

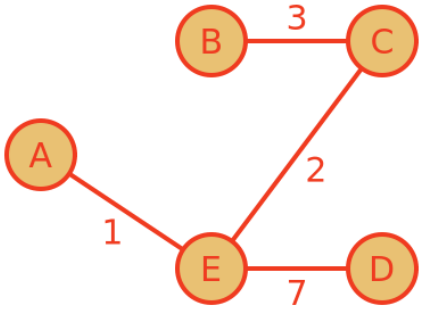


Figure 8: Dijkstra tree showing shortest distance from start for any vertex [7]

Enhanced versions of the algorithm selectively sample critic nodes into a more manageable search space to reduce computational time and overall turns, while maintaining similar path quality [8]. Some applications integrate dynamic costs to reflect real-time conditions like potholes, accidents and traffic congestions, making it more useful to guide autonomous vehicles such as the one represented in Figure 9 [9].

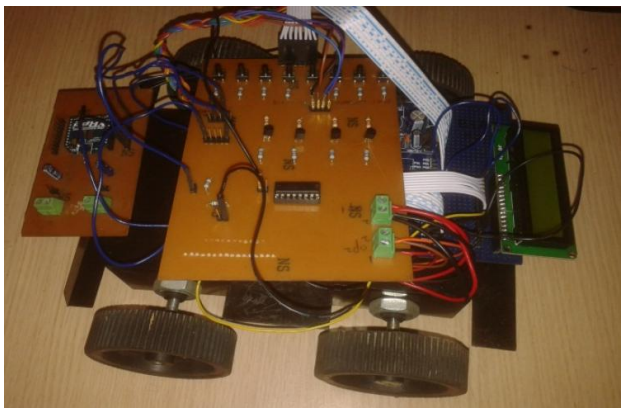


Figure 9: Autonomous driving prototype for Dijkstra testing [8]

Despite its strengths, the basic Dijkstra's algorithm is best suited for static environments and may require re-computation when faced with dynamic obstacles or changes in the environment. Dijkstra's algorithm is computationally intensive in large environments and struggles in dynamic contexts. While edge costs can be adapted, modelling real-world complexities remains a challenge - especially when finding the globally optimal paths in the presence of dynamic or unforeseen obstacles [6] [7] [8] [9].

In contrast to A*, Dijkstra's lack of heuristic guidance makes it less efficient in sparse maps but more reliable where heuristic estimation may be unreliable. Nevertheless, like A*, most research evaluates Dijkstra in simulation rather than in embedded AGV systems. Table 1 represents a comparison of different aspects of the A and Dijkstra algorithms.

Table 1: Direct Comparison Between A and Dijkstra*

Aspect	A*	Dijkstra
Optimality	Guaranteed [4]	Guaranteed [4]
Efficiency	Higher (Guided Search) [3]	Lower (Exhaustive search) – $O(n^2)$ [8]
Adaptability	Moderate (static maps preferred) [3]	Low (static maps required) [4]
Heuristic dependence	Yes [4]	No [4]

LEARNING BASED ALGORITHMS

SOFT ACTOR-CRITIC

Soft Actor-Critic (SAC) stands out among reinforcement learning (RL) methods due to its entropy-maximising objective, which promotes robust exploration, ensuring smooth, adaptive movement in dynamic and uncertain warehouse environments. Unlike value-based methods, SAC is a model-free, policy-based approach that maps states directly to actions without needing a predefined model of the environment with state transitions or rewards [10].

SAC's key innovation lies in its dual-objective: maximising expected cumulative rewards and action entropy. SAC empowers an agent to handle tasks with a degree of randomness, thereby enhancing its adaptability and flexibility. This leads to more diverse exploration, preventing premature convergence and enhancing generalisation, while preventing the algorithm from becoming trapped in sub-optimal local minima or even losing the ability to learn [10] [11]. Typically, a Gaussian policy is used, with the policy network predicting the mean and standard deviation of the action distribution [10].

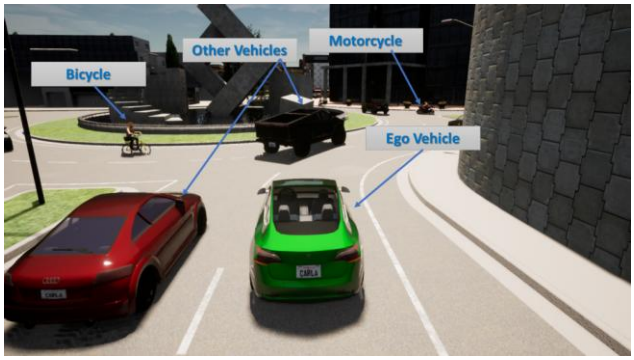


Figure 10: Simulation scenario of the ego vehicle entering the roundabout [9]

Due to its characteristics, the SAC algorithm shows promising results in autonomous driving navigation, particularly in challenging scenarios like entering roundabouts with dense traffic, as demonstrated in Figure 10 [10]. It has also been used for optimal AGV navigation in dynamic environments (shown in Figure 11), demonstrating robustness to original conditions and various obstacle restrictions [11] [12].

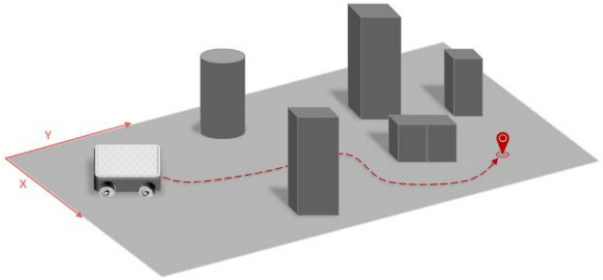


Figure 11: Illustration of AGV navigation in environment with different obstacles present [11]

SAC can suffer from slow convergence, particularly in complex high mobility scenarios such as autonomous driving in urban environments [10] [11]. The practical transferability of policies learned in simulation to real-world applications also remains a significant hurdle for SAC, as training often occurs in simplified environments [10]. The effectiveness of the SAC policy heavily relies on a well-crafted reward structure - poorly designed reward functions can lead to issues like sparse or delayed rewards, hindering effective learning [12].

CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks (CNNs) are a key enabler of intelligent navigation in mobile robots. CNNs are particularly effective in navigation due to their ability to extract hierarchical visual features from sensor data. One major application is visual localisation, where camera images are analysed to determine the robot's position within a mapped environment, often through image classification techniques that match visual input to known locations on a topological map. One study uses a CNN trained on a small indoor dataset with a custom loss function for localisation [13]. CNNs are also essential for object detection, allowing robots to recognise and respond to obstacles such as vehicles, pedestrians, or closed doors, thereby supporting safe and autonomous navigation (Object detection achieved by CNN shown in Figure 12) [14].

Figure 12: Experimental object classification of a CNN [13]

The CNN model described in this context processes RGB map images of 256 × 256 pixels, with an architecture comprising four convolutional layers paired with max-pooling layers, followed by two fully connected layers. As shown in Figure 13, a typical CNN begins with an input layer that receives image data as a three-dimensional tensor (height × width × channels). Convolutional layers extract features using learnable filters, followed by ReLU activation for non-linearity. Max-pooling reduces spatial dimensions, lowering computational load and increasing robustness. The extracted features are then flattened and passed through fully connected layers, with a final Softmax layer generating class probabilities for classification tasks [13].

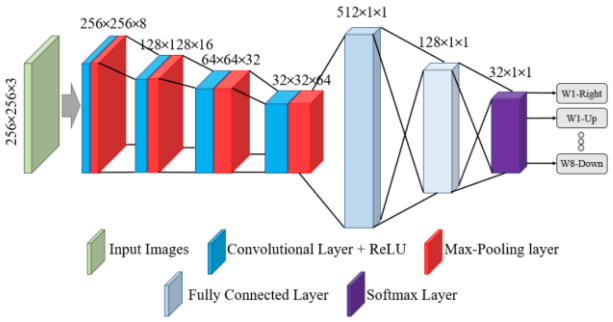


Figure 13: Typical CNN structure [13]

Despite their strengths, CNNs face notable limitations. In indoor settings, they may struggle with visually repetitive environments and are prone to overfitting when trained on small datasets [13]. For outdoor applications, models can be affected by adverse weather and show reduced accuracy with small or unseen object classes [14]. Generalisability across diverse environments remains a challenge, and even lightweight architectures still present computational demands that may limit deployment on embedded systems [13] [14].

ANIMAL INSPIRED ALGORITHMS

ANT COLONY OPTIMISATION

Ant Colony Optimisation (ACO) simulates artificial ants depositing pheromone concentrations to solve optimisation problems such as navigating from a start (ant nest) to a goal position (food source). In nature, ants deposit pheromone on the ground to mark routes between their nest and food sources. Over time paths to promising food sources accumulate higher pheromone concentrations, leading other ants to favour these routes [15] [16].

The algorithms starts with a uniform pheromone concentration across all paths and places artificial ants at the starting point. Each ant constructs a path to a goal position based on a heuristic function - distance from the current point to the goal position - and previously deposited pheromones, as high concentrations lead to promising solutions. The probability p_{ij} of an ant at node i choosing to move to node j is calculated based on pheromone level τ_{ij} and the heuristic information η_{ij}.

$$P_{ij}^{k}(t) = \frac{[\tau_{ij}(t)]^{\alpha}\,[\eta_{ij}(t)]^{\beta}}{\Sigma_{allowed_k}\,[\tau_{ij}(t)]^{\alpha}\,[\eta_{ij}(t)]^{\beta}}$$

$allowed_k$ represents the set of nodes that the ant is allowed to choose for its next step. α reflects the significance of pheromone concentration and β indicates the importance of heuristic information during path selection [15] [16].

During each iteration, the ants deposit pheromones on the path they transverse. The amount of pheromone deposited $\Delta\tau_{ij}$ is proportionally to the quality (length) of the path, with shorter paths leading to more pheromones being deposited, making those paths more attractive for future ants. Over time pheromone levels in the search space decrease (Pheromone evaporation ρ), preventing premature convergence to a suboptimal solution, while encouraging further exploration.

$$\tau_{ij}(t+1) = (1-\rho)\,\tau_{ij}(t) + \Delta\tau_{ij}(t)$$

Where $\tau_{ij}(t+1)$ represents the pheromone concentration between node i and node j at time $t+1$, showcasing the updated pheromone concentration. Through this iterative feedback and the positive feedback mechanism of pheromone accumulation of promising paths, the ant colony converges towards one or multiple optimal paths [15] [16].

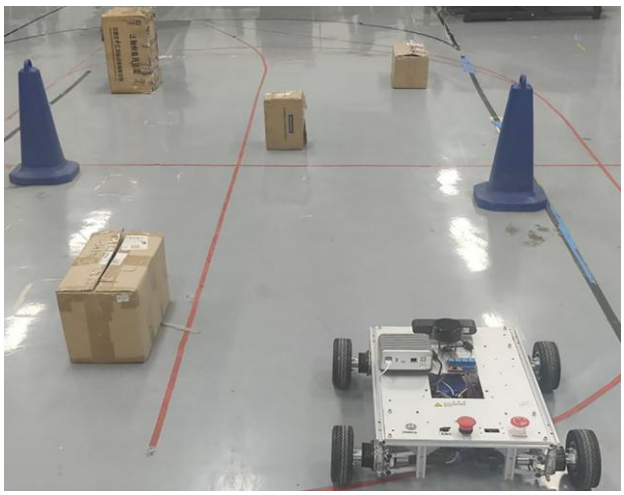


Figure 14: Experimental scene testing ACO [16]

The experimental scene shown in Figure 14, featuring cardboard boxes and cones as obstacles, demonstrates the practical applicability of the algorithm on an AGV experimental platform integrated with the ROS system. This physical validation, alongside the simulation results, distinguishes this paper by showcasing the algorithms performance not just in a virtual environment but also on actual robotic hardware.

While offering benefits like positive feedback, the traditional ACO algorithm has the potential for long computational times, reducing the effectiveness of the system in dynamic environments, especially with larger colonies or iterations [15]. Due to the initial uniform pheromone distribution, ants explore the environment with little guidance, which can be inefficient and delay the discovery of optimal paths [16]. Additionally, ants may converge on sub-optimal paths prematurely, making the discovery of globally optimal paths more difficult [15] [16].

DISCUSSION ON RELEVANT LITERATURE

Given the broad range of papers and navigation approaches reviewed, this study covers a substantial volume of information, much of which is not comprehensively addressed in previous surveys. The comparison presented in Table 3 in Appendix II aims to systematically highlight the key differences and relative advantages of the various navigation techniques.

While A* and Dijkstra both guarantee optimal paths in static maps [4], they differ in efficiency: A* benefits from heuristic guidance, whereas Dijkstra's exhaustive approach becomes computationally intensive in larger graphs. Learning-based methods like SAC offer flexibility in dynamic environments but impose high computational costs and lack guaranteed optimality [10]. Meanwhile, reactive models like Braitenberg vehicles, though computationally trivial, are incapable of long-term path planning - a critical limitation when compared to even basic classical algorithms [1]. CNNs support vision-based navigation through localisation and obstacle detection, but face challenges with generalisation and require significant processing power and training data [14]. Similarly, ACO enables probabilistic path discovery via decentralised search yet tends to converge slowly and can struggle in highly dynamic settings [16]. This range of techniques illustrates key trade-offs between optimality, adaptability, and computational overhead.

This comparative analysis not only categorises key navigation strategies but also serves to guide the selection and adaptation of algorithms appropriate for constrained platforms such as the LEGO Spike Prime used in this project. A notable gap in the literature is the limited real-world validation of these techniques on accessible educational robots, where simplified sensors, restricted processing power, and hardware constraints introduce unique challenges. As such, factors like computational efficiency, adaptability to dynamic conditions, and implementation feasibility are central to algorithm choice. These considerations directly inform the methods explored in the following section, where the practical integration of selected navigation techniques with the LEGO Spike Prime is investigated and evaluated.

METHODS

This section outlines the navigation methods implemented in the project including: Braitenberg 2a/3b, A*, Dijkstra, CNN and ACO. Although reinforcement techniques were discussed in the literature review, they were ultimately excluded due to their high complexity. RL algorithms typically require extensive training over thousands of simulations, which was not feasible within the limited timeframe of this project. Moreover, the relative simplicity of the navigation task did not warrant the use of such resource-intensive methods.

Since the LEGO SPIKE Prime hub runs a complete MicroPython script that is processed directly on the hub, achieving real-time navigation with this hardware is not possible. The closest approximation to real-time tracking is by logging executed navigation commands within the script to a file. This log can then be used later to retrace the actions taken by the LEGO AGV. If an obstacle is detected, the log file can be used to figure out where the obstacle was detected to resume navigation from there. Though this limitation does not affect reactive navigation, planned methods can dynamically update the grid upon obstacle detection and replan routes, mimicking real-time adaptability.

For the implementation and evaluation of path planning algorithms, a virtual environment is required, along with designated start and goal positions. The environment is modelled as a binary occupancy grid, where each cell is assigned a value of 0 to indicate traversable space and 1 to denote non-traversable obstacles. This grid-based representation provides a structured framework for the AGV to compute collision-free paths. In all visualizations, obstacles are depicted in black and free space in white, facilitating intuitive interpretation of the navigation space.

BRAITENBERG-INSPIRED OBSTACLE AVOIDANCE ROBOT

As discussed in the literature review, Braitenberg vehicles 2a and 3b use two sensors to drive two motors, allowing the robot to avoid obstacles by reacting to stimuli. However, the LEGO SPIKE Prime kit provided only one ultrasonic sensor, limiting the system's capability for differential input.

To adapt, a simplified Braitenberg-like behaviour was implemented. The single ultrasonic sensor was mounted at the front of the AGV, where it emits high-frequency sound waves and measures the time it takes for the echoes to return, allowing the system to

calculate the distance to nearby objects [17]. The AGV continuously drives forward until the sensor detects an obstacle within 10 cm. When triggered, the AGV reverses slightly and performs a small left turn before continuing forward. Defaulting to a left turn creates predictable and consistent obstacle avoidance, while the brief reverse movement helps free the AGV from tight corners - an issue encountered in testing. The code for this implementation can be found in Appendix III Figure 74.

However, ultrasonic sensors have a significant real-world limitation: they rely on sound waves reflecting directly back to the sensor [17]. When the AGV approaches an obstacle at an angle, the emitted sound waves may deflect away rather than bounce back to the sensor, resulting in no detection even when an object is present. As shown in Figure 15, this creates blind spots where the AGV can potentially collide with obstacles it fails to "see". As a result, the obstacle avoidance system may perform inconsistently in complex environments or when obstacles are not approached head-on.

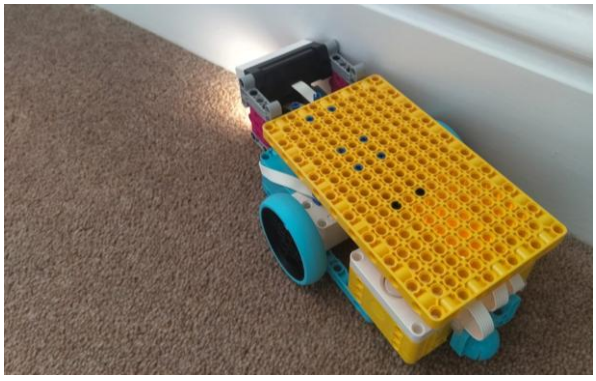


Figure 15: LEGO based Braitenberg vehicle fails to detect obstacle due to sound waves being deflected off object rather than reflected

Despite this, the simplified implementation still enabled the AGV to perform basic reactive navigation in relatively open environments [https://github.com/tlr30/master_thesis/blob/master/ML_navigation/videos/Braitenberg%20Hate.mp4].

LINE FOLLOWING ROBOT

Similar to Braitenberg's reactive concept, another common form of sensory-motor navigation is line following. Traditional line-following robots often use two or more infrared or inductive sensors to track black tape or a wire [18] [19]. However, the LEGO kit included only one colour sensor, making traditional implementations infeasible.

To overcome this, the navigation track was designed using different types of coloured tape: green (left), yellow (right), and a white gap in between (as shown in Figure 16). When the colour sensor detects green, the AGV adjusts slightly right; when it detects yellow, it turns left. The white space allows the AGV to continue moving straight. This design prevents constant oscillation between turns, enabling smoother forward movement.

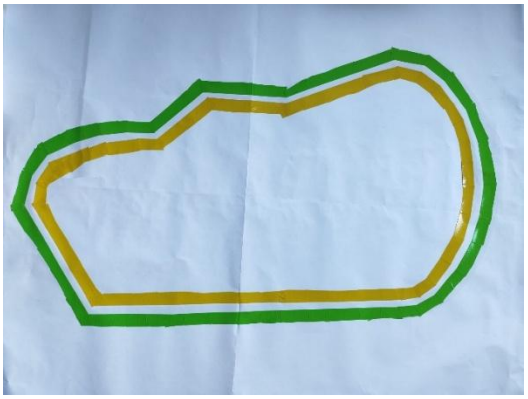


Figure 16: Navigation map including green and yellow tape allowing the LEGO AGV to guide itself along the tape

The AGV follows the coloured track until an obstacle is detected within 5 cm by the ultrasonic sensor. Upon detection, the AGV stops, tilts the colour sensor toward the obstacle, and identifies the object's colour. The Code for this implementation can be found in Appendix III Figure 75 or on GitHub [https://github.com/tlr30/master_thesis/blob/master/ML_navigation/pybricks/colour_navigation.py]. A video of line following robot can be found on GitHub [https://github.com/tlr30/master_thesis/blob/master/ML_navigation/videos/leog_agv_yellow_green_map.mp4]

A* ALGORITHM

The next class of navigation techniques involved planned pathfinding based on an environmental map. Given a start (x, y) and a goal (x, y) position, the AGV is programmed to follow an optimal path determined using A*.

Users can choose whether diagonal movement is allowed. Depending on this, the Manhattan or diagonal heuristic is used, where a cost of 1 is assigned for horizontal and vertical moves, a cost of $\sqrt{2}$ represents diagonal moves.

The A* algorithm maintains an open set - a priority queue that stores nodes yet to be explored, prioritized by their f-score ($f(n) = g(n) + h(n)$), where $g(n)$ is the exact cost from the start to node n, and $h(n)$ is the heuristic estimate of the cost from n to the goal. A closed set tracks nodes that have already been explored to prevent redundant processing. This structure ensures that the algorithm always explores the most promising path first, balancing actual path cost and estimated remaining distance, which makes A* both complete and optimally efficient when the heuristic is admissible.

A helper function *is_valid_move* ensures efficient obstacle avoidance. AGV moves are validated against grid boundaries, as shown in Figure 17.

```
## 1. check that obstacle is not outside
of bounds
    if not (0 <= target_x <
len(self.grid) and 0 <= target_y <
len(self.grid[0])):
    return False
```

Figure 17: Code snippet that ensures A navigates within map boundaries*

LEGO positions are precomputed within dictionary *blocked_positions*, storing coordinates of all obstacles - excluding the goal (Code snippet shown in Figure 18).

```
blocked_positions = {
            (by, bx) for colour, (by,
bx) in self.block_locations.items()
            if (by, bx) != self.goal
        }

if (target_y, target_x) in
blocked_positions:
            return False
```

Figure 18: Code snippet that ensures A can access precomputed block locations*

Using a precomputed set of blocked positions enables O(1) constant-time validity checks for each move. This optimization reduced pathfinding computation time by over 80%, accelerating the system from 2.5 seconds to under 0.5 seconds.

For diagonal moves, the system checks both cardinal neighbours to prevent clipping (clipping vs successful navigation demonstrated in Figure 19 – python implementation shown in Figure 20).

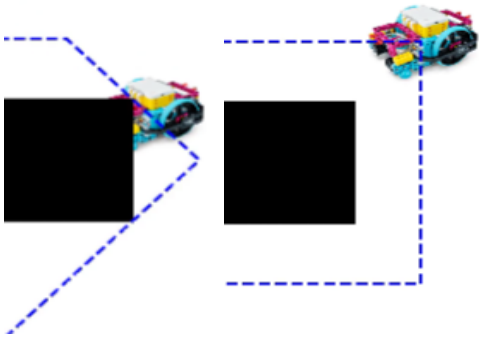


Figure 19: No padding (left side) vs padding (right side) for diagonal movements

```
## 3. additional check for diagonal
moves - look at the "corner" between
current and target
if dx != 0 and dy != 0:  ## if this is a
diagonal move
    if (x + dx, y) in blocked_positions
or (x, y + dy) in blocked_positions:
        return False
```

Figure 20: Code snippet that ensures A doesn't clip obstacles during diagonal moves*

Within the *is_valid_move* helper function, an additional check is performed to accommodate the user's ability to scale the AGV. This feature is particularly useful for large maps, as it mirrors real-world scenarios where AGVs in warehouse environments must adjust their paths if they are too large to fit through certain narrow sections of the optimal route. As the AGV size increases, it is essential to verify that the enlarged AGV does not collide with obstacles and remains within the grid bounds. The AGV size is represented as a scaling factor relative to the map, ensuring that the AGV adjusts its size based on the grid's dimensions.

Additionally, if the scaling factor adjusts the AGV size to a radius of 5, checks are conducted in a square with a size of 9x9, effectively reducing the radius from 5 to 4.5. This approach was chosen over a circular check, as most AGVs are typically rectangular in shape, making a square detection area

a more accurate representation of the vehicle's physical footprint.

Originally, the entire radius in both the x and y directions was checked in each iteration, evaluating the entire square area surrounding the AGV. However, this approach proved to be inefficient for big AGVs. To optimize this, the algorithm was modified to perform a full square check only during the first iteration, and in subsequent iterations, it only checks the outermost layer of the square for boundary and obstacle violations. This optimization reduced the processing time from 346ms to 221ms. The enlarged AGV, depicted in Figure 21, demonstrates the optimal path while successfully avoiding obstacles. Figure 22 demonstrated the same scenario but with a reduced AGV size. The code can be found in Appendix III Figure 76.

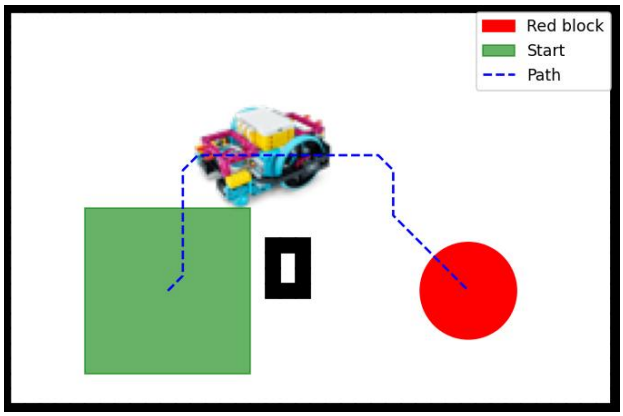


Figure 21: Path of enlarged AGV, successfully avoiding obstacles

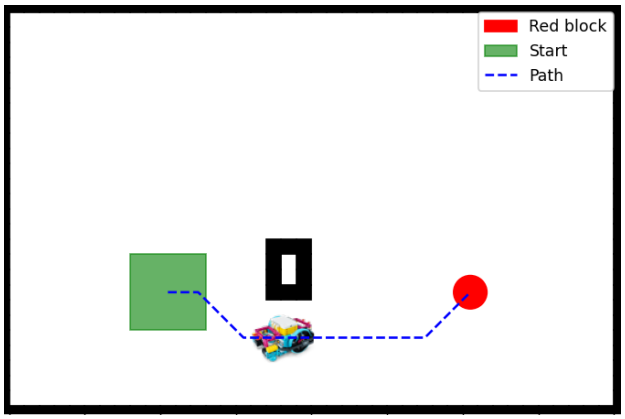


Figure 22: Path of a smaller AGV, successfully avoiding obstacles

To improve real-world performance, unnecessary directional changes were penalised. While multiple paths might have equal costs, favouring routes with fewer turns improves speed and smoothness in real-world AGV movement. The python implementation for this is found in Figure 23.

```
## penalise turns to avoid unnecessary
turns
if (previous_dx, previous_dy) != (dx,
dy):
    tentative_g_score += 0.5
```

Figure 23: Code snippet that ensures A favours fewer overall turns*

Figures 24 and 25 compare routes before and after the turn penalty was introduced. This feature was disabled during testing because it affected each navigation method differently, leading to inconsistent path costs for what should have been the same path.

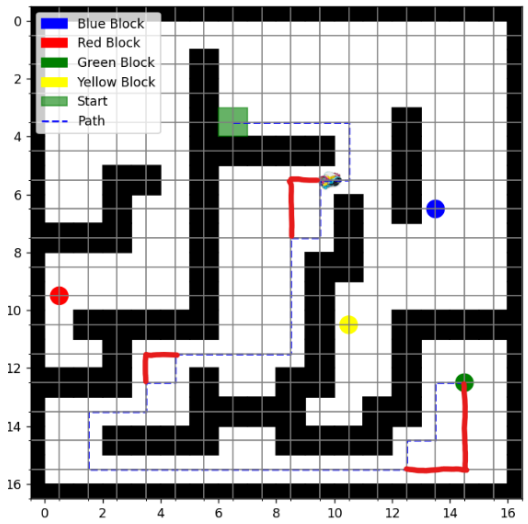


Figure 24: A path before turn penalisation with red paths showing what the optimised path should look like*

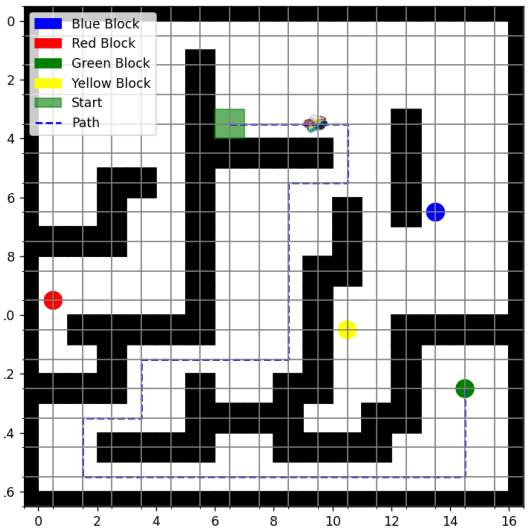


Figure 25: A path after turn penalisation, showing the optimal path*

Another potential improvement identified during the literature review was the use of Bezier curves, which can smooth path transitions while ensuring kinematic feasibility. However, due to the simplicity of the LEGO hardware employed in this project, this feature was ultimately not implemented.

Once the optimal path is found, a function *reconstruct_path* backtracks using a dictionary of visited nodes (e.g., {A: start, B: A, goal: B}) to build the full path from start to goal.

An animation illustrates the AGV's movement from the start to the goal location, providing a clear visual of the computed path. Additionally, an exploration heatmap is generated to show how often each node is visited during the search process, offering insight into the algorithm's behaviour.

To enhance the overall user experience when utilizing the A navigation algorithm, comprehensive error handling was implemented through the use of ValueError exceptions. These errors are raised in cases where the start or goal position is out of bounds, or if either lie on obstacles, with clear messages indicating the cause of the issue. An additional error is triggered if the start and goal positions are identical. The algorithm first attempts to compute a valid path using the user-specified AGV size. If no viable path is found, the AGV size is automatically reduced to its minimum permissible value to check whether the originally chosen size was too large. If a valid path is still not found, a final error is raised, clearly stating that no valid path could be determined. The complete A* implementation, including all ValueError checks and handling, is provided in Appendix III, Figure 77. The complete source code, including its integration within the overall workflow, is available on GitHub [https://github.com/tlr30/master_thesis/blob/master/ML_navigation/astar.py].

DIJKSTRA'S ALGORITHM

Dijkstra's algorithm was also implemented to perform optimal pathfinding from a start to a goal location on a discrete map. Unlike A*, which uses a heuristic to guide exploration, Dijkstra explores all possible paths uniformly by expanding nodes in order of their accumulated path cost. This makes it particularly useful in environments where all movements have equal cost or where heuristic guidance may not be reliable.

Dijkstra's algorithm explores the map by maintaining a priority queue (min-heap) of nodes to be processed, where each node is prioritized solely by the cumulative cost from the start (g(n)), without using any heuristic guidance. A visited dictionary stores the cost and predecessor for each explored node, while a closed set ensures that no node is processed more than once. Unlike A*, which prioritizes nodes based on estimated total cost, Dijkstra expands nodes uniformly in cost order, guaranteeing optimal paths in graphs with non-negative edge weights. In this implementation, movement can be configured to allow cardinal or diagonal directions, with corresponding costs of 1 and $\sqrt{2}$, respectively.

Typically, Dijkstra's algorithm explores all traversable cells within the search space exactly once. To reduce computational overhead, the algorithm was modified to terminate as soon as the goal position was reached, eliminating unnecessary exploration beyond the optimal path.

Movement is validated with helper function *is_valid_move,* ensuring movement stays within map boundaries and the AGV does not collide with coloured blocks or "clip" obstacle corners. Once again, users can observe how differently sized AGVs result in varying optimal paths within warehouse environments. Unnecessary directional changes were avoided by introducing turn penalisation. Additionally, the function *reconstruct_path* builds the full path from start to goal.

As with A*, Dijkstra's implementation includes an animation of the AGV's path and a heatmap showing node visit frequency, illustrating its uniform exploration pattern.

To enhance the user experience, all ValueError handling mechanisms implemented in the A* algorithm were also integrated into the Dijkstra algorithm. This ensures consistent feedback across search methods. The corresponding implementation is provided in Appendix III, Figure 78. The complete source code, including its integration within the overall workflow, is available on GitHub

[https://github.com/tlr30/master_thesis/blob/master/ML_navigation/dijkstra.py].

CONVOLUTIONAL NEURAL NETWORKS

As previously mentioned, the LEGO SPIKE Prime hub operates on MicroPython, which limits its ability to process real-time sensor data for navigation. Consequently, instead of using traditional real-time sensor inputs for obstacle localization in the CNN, the system navigates in environmental grids containing obstacles and free spaces (represented as 1s and 0s respectively). This grid-based approach allows for the navigation path to be computed by the CNN using pre-generated maze-like grids, making it suitable for the LEGO hub's computational constraints.

To train the CNN model, random start and goal positions are selected within each grid. The A* algorithm is then used to find the optimal path between the start and goal positions, which serves as the ground truth for the CNN training. To ensure that the paths used for training are sufficiently complex and informative, only those paths whose length is at least three times the grid width are selected. Assuming a 5x5 grid, each input to the CNN is formatted as a 3-channel tensor of shape (5, 5, 3), where the three channels represent the static environment layout (obstacles and free spaces), the start position, and the goal position - each encoded as a binary 5×5 grid. The corresponding ground truth output is a single-channel tensor of shape (5, 5, 1), representing the optimal path as a binary mask. The CNN is designed to predict the optimal path from the start to the goal within the grid. This model architecture is a deep CNN that processes the environment as input through multiple convolutional layers, each using ReLU activation and increasing filter sizes (32 to 512). The output of the network is a (5, 5, 1) probability map that highlights the most likely path from start to goal. The CNN structure is shown in Appendix III Figure 79.

The CNN architecture that produced the best performance in this project consisted solely of stacked convolutional layers without any max pooling layers. The network concluded with a flattening layer followed by dense layers to generate the final output. This version of the model excluded any use of *MaxPooling2D* or *AveragePooling2D,* which was initially considered to reduce dimensionality and improve efficiency. However, in practice, the pooling layers degraded performance, likely due to the loss of spatial resolution critical for accurate path prediction in the grid-based environment.

Quantitatively, the CNN without pooling achieved an accuracy of 95.69%, while the same architecture with max pooling layers inserted after each convolutional block achieved only 80.42% accuracy, often misinterpreting obstacles with traversable space. In contrast, average pooling resulted in improved performance, achieving 90.58% accuracy. All other hyperparameters, including the number of filters, learning rate, optimizer, and training epochs, were kept constant to ensure a fair comparison.

Figures 26, 27 and 28 illustrate the comparative performance: Figure 26 shows the result using the CNN without pooling layers, maintaining finer spatial features across all layers. In comparison, Figure 27 and Figure 28 demonstrate the performance drop when max pooling and average pooling, respectively, were introduced at various stages of the network. Different configurations of pooling layers were tested to evaluate their impact on spatial feature preservation and overall model accuracy.

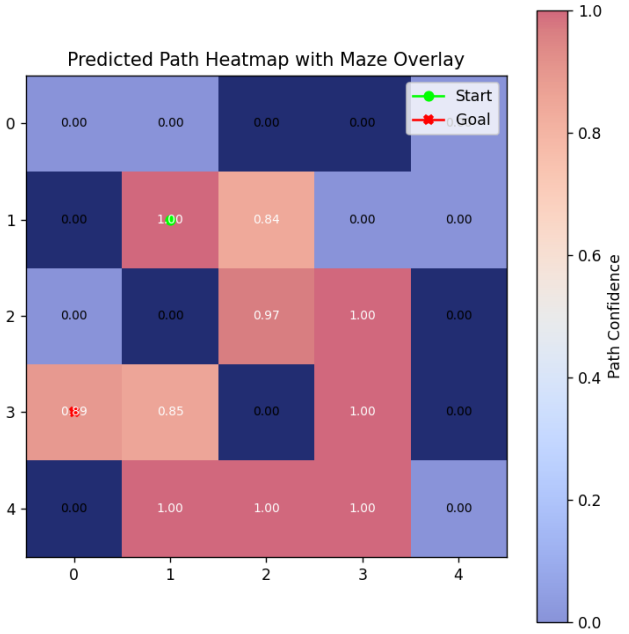


Figure 26: Output heatmap of the CNN for a 5×5 grid excluding maxpooling layers

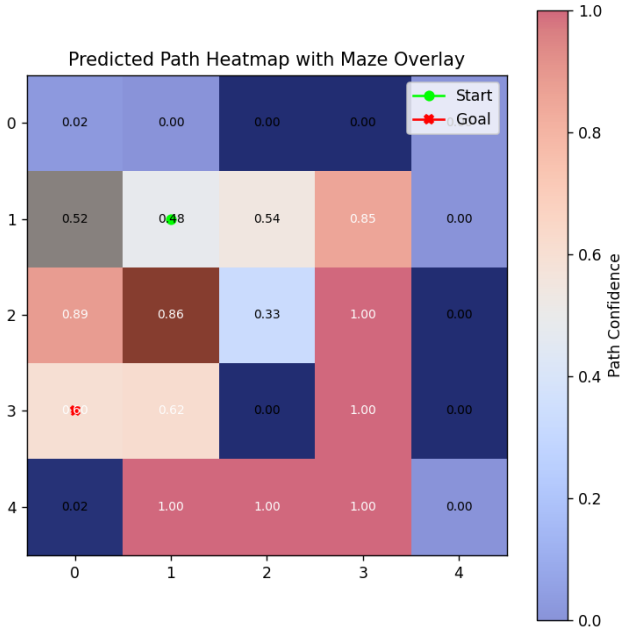


Figure 27: Output heatmap of the CNN for a 5×5 grid including maxpooling layers, keeping the model structure the same otherwise

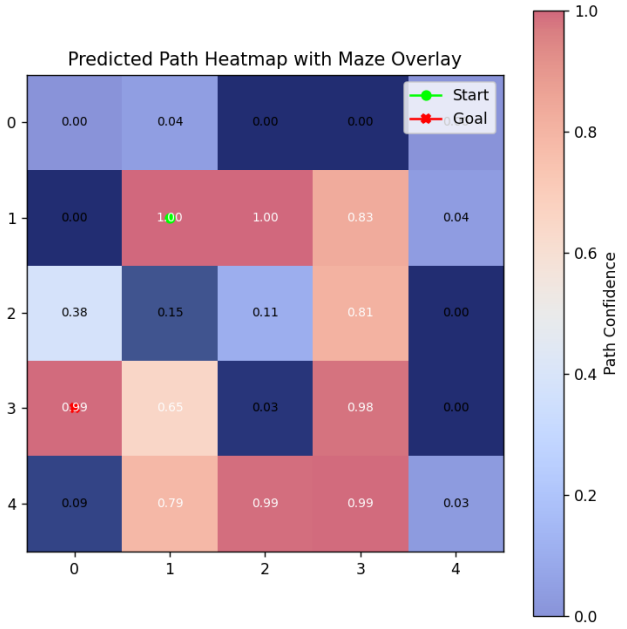


Figure 28: Output heatmap of the CNN for a 5×5 grid including averagepooling layers, keeping the model structure the same otherwise

This outcome suggests that for this specific application - where precise spatial information is essential - preserving feature map resolution throughout the network was more beneficial than reducing computational complexity.

The model is trained using the Adam optimizer, with binary cross-entropy loss to minimize prediction errors. The training process focuses on enabling the model to generalize across various maze configurations by learning from a set of training mazes with known optimal paths. Evaluation of the trained CNN model is carried out by testing it on unseen grid environments, where path accuracy, obstacle avoidance, and computational efficiency are assessed. To ensure the reliability and quality of the predicted paths, the model's output is compared to the A* algorithm's ground truth, allowing for a performance validation based on known optimal paths.

Once the CNN is trained and evaluated, it is tested on the actual input grid, where it generates a heatmap indicating the probability of each cell being part of the path between the start and goal. Users can visually interpret this heatmap by focusing on cells with the highest probabilities to construct a potential path. However, an automated path reconstruction was not implemented, as the CNN occasionally omits cells that belong to the optimal path - an issue influenced by the number of training images and chosen computational parameters. To facilitate practical deployment, the trained CNN model is saved and can be reused for inference on new, unseen grids of the same size and general structure. This eliminates the need for retraining, enabling near real-time path prediction suitable for embedded systems like the LEGO SPIKE Prime hub.

Before evaluating the CNN on the actual input grid, a fixed set of generated test grids was used to assess the model's performance. Each of these grids was solved by the trained CNN and compared against ground truth solutions obtained using the A* algorithm. The purpose of this step was to evaluate how well the CNN could generalize to grids similar to those it was trained on. In several instances, the model demonstrated limited generalization capability, struggling to solve test grids that deviated slightly from the training distribution - highlighting a key limitation in its robustness. Examples of successfully solved grids (Figure 81) and failed attempts (Figure 82) are provided in Appendix IV.

To reduce complexity, the training data was intentionally simplified. Only cardinal movements were allowed, and variable AGV sizes were excluded by defaulting to single-cell navigation. These design choices helped limit the dimensionality of the problem.

The code employs an 80/20 train-test split to ensure the model is evaluated on unseen data, helping reduce overfitting by measuring performance on the held-out test set. Additionally, it uses two key callbacks to optimize training: *ReduceLROnPlateau* reduces the learning rate by half when validation loss stagnates for 3 epochs, preventing overshooting optimal weights, while *EarlyStopping* halts training if no improvement occurs for 5 epochs and restores the best weights, avoiding unnecessary overfitting. Together, these techniques balance efficient training with robust generalization.

A full implementation of the CNN code can be found on GitHub [https://github.com/tlr30/master_thesis/blob/master/ML_navigation/cnn_navigation.py].

ANT COLONY OPTIMISATION

The ACO algorithm was implemented as a biologically inspired algorithms, where virtual ants explore a search space and deposit pheromone on traversed paths, iteratively guiding ants to optimal paths. The implementation supports both cardinal and diagonal movement - the algorithm dynamically selects between Manhattan and diagonal heuristics to match the movement model, ensuring accurate estimation of remaining cost.

Cells representing obstacles are initialized with zero pheromone concentration to ensure ants avoid them entirely during path construction.

Each ant constructs a path based on the pheromone intensity α and heuristic information β. Each move is probabilistic selected using the logic demonstrated in Figure 29:

```
probability = (pheromone ** self.alpha)
* (heuristic ** self.beta)
```

Figure 29: Code snippet of ACO's probabilistic selection process

This allows ants to balance between exploration (trying new paths) and exploitation (following stronger pheromone trails). Incorrect calibration can cause premature convergence or inefficient search. To ensure realistic movement and prevent impractical behaviour around obstacles, diagonal movement is only allowed if the adjacent cardinal cells are free. Furthermore, previously visited nodes are excluded from selection avoiding ants being trapped in local loops, enhancing search efficiency and solution diversity.

Once all ants have completed their paths in each iteration, pheromone levels are updated based on the quality (or length) of the paths discovered. delta_t represents the pheromone deposit constant. The pheromone deposit calculation is represented in Figure 30:

```
pheromone_deposit = self.delta_t / cost
```

Figure 30: Code snippet of ACO's pheromone deposit calculation

where shorter paths receive a higher pheromone deposit. Additionally, to prevent early convergence and maintain diversity in the search, all pheromone trails undergo gradual evaporation - python snippet shown Figure 31.

```
self.pheromone *= (1 - self.rho)
```

Figure 31: Code snippet of ACO's pheromone evaporation

The algorithm iterates until a predefined maximum number of iterations is reached or until improvements plateau. This early stopping condition helps reduce unnecessary computation while maintaining the quality of the solution.

Following the same approach used in the A* implementation, a helper function, *is_valid_move*, ensures that movement remains within the map boundaries and prevents the AGV from "clipping" obstacle corners. In the ACO implementation, support for differently sized AGVs was omitted to avoid increased algorithmic complexity and additional computational overhead, which is already higher in ACO compared to A* and Dijkstra. To improve path quality, unnecessary directional changes were minimized through turn penalisation. Finally, the *reconstruct_path* function assembles the complete path from the start to the goal.

As with A* and Dijkstra, ACO's implementation includes an animation of the AGV's path and a heatmap showing node visit frequency, illustrating its uniform exploration pattern. The generated heatmap provides a spatial frequency analysis of node visitation, enabling insight into the algorithm's coverage and exploration efficiency.

A full implementation of the ACO algorithm can be found in Appendix III, Figure 80. The complete source code, including its integration within the overall workflow, is available on GitHub [https://github.com/tlr30/master_thesis/blob/master/ML_navigation/aco.py].

PIPELINE IMPLEMENTATION

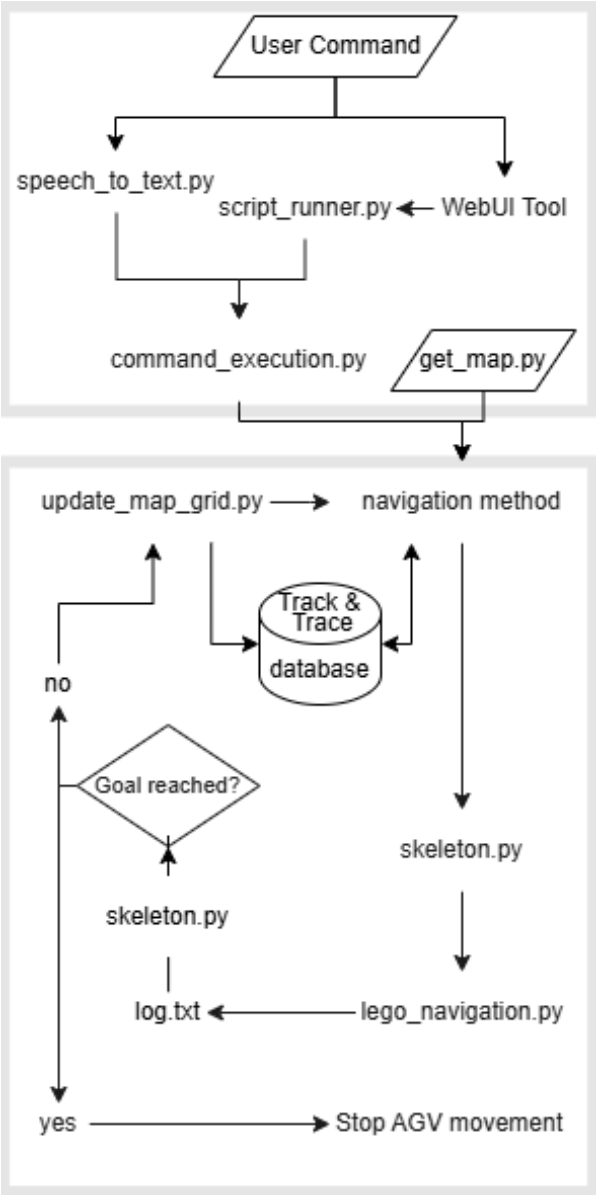


Figure 32: Flowchart of workflow implementation

The flowchart in Figure 32 represents the overall workflow developed in this master thesis. In collaboration with a company the goal of the project was to create an interactive training for the consultancy, aimed at educating clients about digitisation of warehouse environments in an engaging way.

For a smoother user experience, Open WebUI was locally installed using Docker containers. This browser based user interface provides a seamless integration between user issued voice commands and the backend system. To connect Open WebUI to VS code and the rest of the navigation workflow, a custom Open WebUI tool was developed [https://github.com/tlr30/master_thesis/blob/master/ML_navigation/web_ui_tool.py], along with a flask server (script_runner.py) running locally in VS code [https://github.com/tlr30/master_thesis/blob/master/ML_navigation/script_runner.py].

When a voice command is given in WebUI, the tool to passes it to *server.py*, triggering the execution of *command_execution.py* via a subprocess [https://github.com/tlr30/master_thesis/blob/master/speech/command_execution.py]. This script is responsible for normalising natural language – a crucial step in ensuring consistent interpretation of varied user inputs. For instance *go to*, *move to*, *navigate to* and *head to* are all normalised to a single command *go_to*, ensuring the AGV receives clear instructions.

While WebUI provides a seamless user interface, running it locally can be resource intense and complex to set up. As an alternative, users can run the workflow directly from VS code using *text_to_speech.py* [https://github.com/tlr30/master_thesis/blob/master/speech/speech_to_text.py]. Both WebUI and the VS code implementation use OpenAI's pretrained model Whisper - an open source speech-to-text model - for capturing spoken navigation commands. These are again passed to *command_execution.py* for normalisation and further processing.

OpenAI's Whisper models are advanced speech recognition systems designed to handle diverse languages and accents with high accuracy. These models are built on a transformer-based architecture and have been trained on 680,000 hours of audio, including 117,000 hours in 96 non-English languages. The most capable variant, Large V2, has achieved a remarkable 2.7% word error rate on the LibriSpeech test-clean dataset, along with a 55.2% average error reduction across 12 challenging

English datasets when compared to similar models [20].

Once normalised, the navigation command is passed as an input argument to the selected navigation method (A*, Dijkstra or ACO), computing the shortest path from the AGV's current position to the target Lego block, as instructed in the navigation command. To support the selection of an appropriate navigation method for specific tasks, Table 4 in Appendix II was created. It compares the implemented algorithms - Braitenberg Vehicles , A*, Dijkstra, CNN and ACO - across key criteria including computational efficiency, path optimality, heuristic dependence, adaptability to different AGV sizes, and support for diagonal movement.

The start position and the location of the Lego blocks are specified in a local csv database [https://github.com/tlr30/master_thesis/blob/master/database/location.csv]. The navigation file generates both an animation showing the AGV's path on the map and a movement sequence for the physical AGV.

The movement sequence acts as the input to *skeleton.py* [https://github.com/tlr30/master_thesis/blob/master/ML_navigation/skeleton.py], which modifies a default Pybricks script to reflect the computed path and creates a new file *lego_navigation.py* [https://github.com/tlr30/master_thesis/blob/master/ML_navigation/pybricks/lego_navigation.py]. This Pybricks script is executed via subprocesses, connecting to the LEGO AGV and instructing it to follow the path. The movement pattern is logged in *log.txt* [https://github.com/tlr30/master_thesis/blob/master/ML_navigation/pybricks/log.txt].

If the Lego distance sensor detects an obstacle, the Lego AGV stops and logs the location of the obstacle in the *log.txt* file. At this point, a downward-facing colour sensor is directed at the obstacle to identify its colour. The colour sensor used is the LEGO SPIKE colour sensor, which provides direct colour classification along with a measurement of surface reflectivity. It is also capable of distinguishing between dark and bright environments, making it suitable for a range of lighting conditions.

Extensive testing was conducted to determine the sensor's reliability across varying brightness levels and obstacle-to-sensor angles. The sensor demonstrated consistent performance within a distance range of 2 to 6 cm and remained accurate when the obstacle was positioned within approximately a 30° angle relative to the sensor, confirming its suitability for the AGV's intended use.

Once an obstacle is detected and logged, the *skeleton.py* script checks the *log.txt* file to determine whether the movement sequence was completed or interrupted. If interrupted, the script calls *update_map_grid.py* [https://github.com/tlr30/master_thesis/blob/master/ML_navigation/update_map_grid.py], providing it with the partially executed path, allowing the AGV to update its internal map accordingly.

The original map is updated to include the new obstacle and the current AGV position is stored in the database. The current orientation of the LEGO AGV is passed back to the navigation method, which uses the updated map and the AGV start position to compute a new path to the original Lego block, starting the navigation process from the beginning until the AGV has successfully navigated to the goal, or no viable path is found.

If no obstacle was detected and the path is completed successfully, the AGV stops at the desired position.

To enhance the interactive learning experience, users are given the option to hand-draw custom warehouse layouts, illustrating how AGV performance is influenced by design. This exercise encourages trainees to consider real-world factors like congestion and inefficient routing. The drawing are digitised using *get_map.py* [https://github.com/tlr30/master_thesis/blob/master/ML_navigation/get_map.py], which converts the sketch into a map format, usable by the navigation method.

Users can select the map resolution - higher resolutions offer more detail but increase computational time for pathfinding and animation generation. For example, Figure 34 shows a map at 100% resolution of original drawing Figure 33, while Figure 35 shows a 5% scaled version that computes significantly faster.

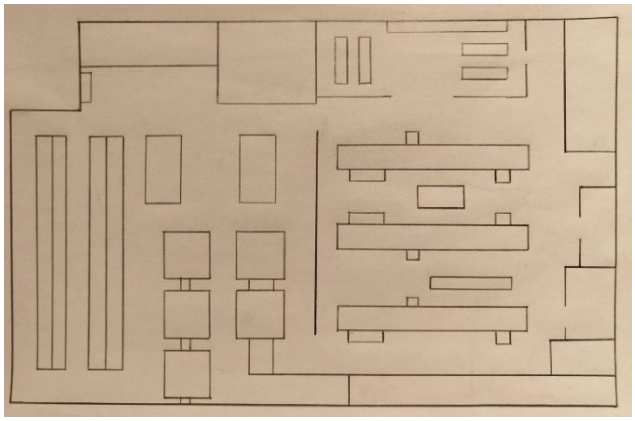


Figure 33: Hand drawn warehouse design

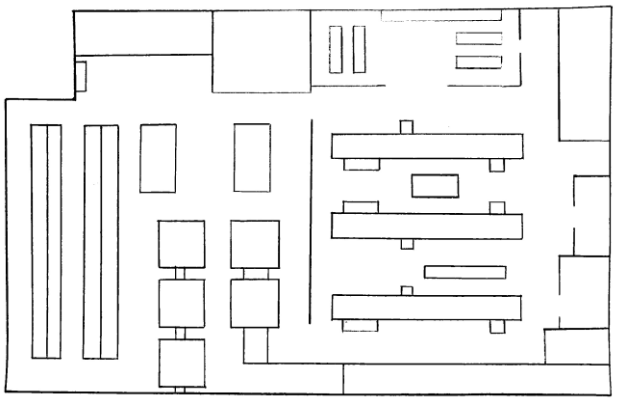


Figure 34: Digitally reconstructed warehouse layout based on the hand-drawn design, preserved at 100% of the original resolution

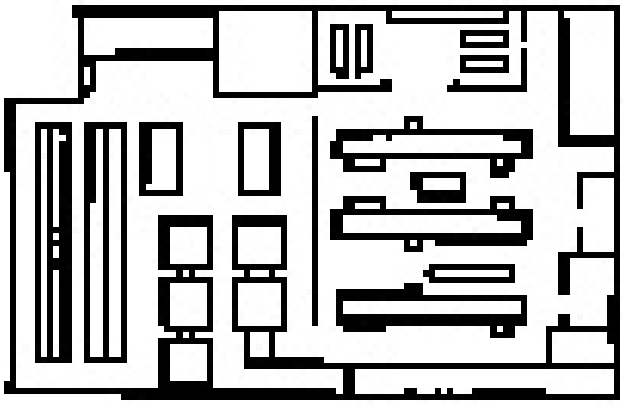


Figure 35: Digitally reconstructed warehouse layout based on the hand-drawn design, preserved at 5% of the original resolution

The database stores block and start positions based on a fixed reference grid, which may differ from the current simulation grid size. To map these locations accurately, row and column scaling factors are computed, and each coordinate is rescaled accordingly. This ensures positions are proportionally distributed across the new grid while remaining within valid bounds.

This end-to-end system allows users to explore AGV navigation in environments of their own design, providing a tangible understanding of how digital systems interact with real-world constraints

A video demonstrating the full navigation workflow of the LEGO AGV - including voice command input via open web UI, real-time heatmap generation, and pathfinding animation - is available on GitHub and can be accessed at: [https://github.com/tlr30/master_thesis/blob/master/ML_navigation/videos/Complete%20Workflow%20Demo.mp4].

RESULTS

To thoroughly evaluate various navigation techniques, multiple types of test environments were developed. Structured grids were designed to resemble maze-like maps featuring multiple routes to the goal as well as numerous dead ends. These grids allowed for comparative performance testing of algorithms across different complexities. To further assess navigation beyond basic pathfinding, additional structured grids were created with wider cell paths. These layouts enabled diagonal movements in addition to cardinal directions, emphasizing the importance of movement optimization rather than merely avoiding obstacles.

For more dynamic evaluation, unstructured grids were introduced. In these, obstacles were randomly distributed across the map, creating a vast range of possible path solutions. This setup challenges navigation algorithms to determine not just any path, but the optimal one in a highly variable environment.

Finally, real-world map data from urban environments such as Berlin and London was incorporated to test algorithm performance under practical, large-scale conditions and to better reflect real AGV deployment scenarios.

Braitenberg and line-following methods were excluded from the structured, unstructured and real-world grid tests due to their fundamentally different navigation principles. These approaches rely on real-time sensor input and reactive behaviours rather than path planning or map-based decisions. As such, they cannot interpret grid layouts or follow predefined routes, making them incompatible with the grid-

based evaluation framework used for A*, Dijkstra, ACO, and CNN pathfinding.

STRUCTURED GRIDS

ONE CELL WIDE PATHS – A*, DIJKSTRA AND ACO

The A*, Dijkstra, and ACO algorithms were tested across 10 different structured grids, each increasing in size - beginning with a simple 4×5 map and culminating in a complex 50×50 maze. The optimal cost of each grid is displayed in Figure 36 and the optimal path for each grid is shown in Appendix V Figure 83-81.

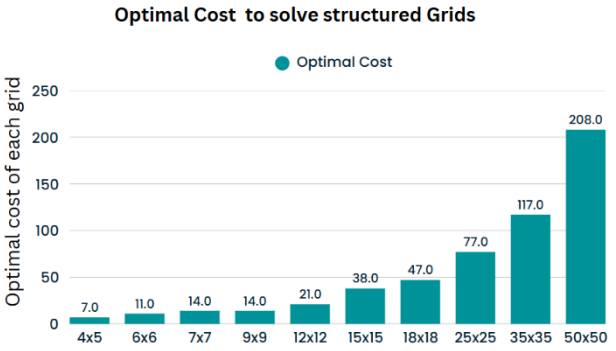


Figure 36: Optimal cost to solve structured grids of sizes 4x5, 6x6, 7x7, 9x9, 12x12, 15x15, 18x18, 25x25, 35x35 and 50x50

Across most grid sizes, A* emerged as the fastest, with Dijkstra closely behind (Figure 37 and 38). ACO, by contrast, was significantly slower in all cases - exhibiting performance gaps of up to 1000× compared to A* and Dijkstra (Figure 39).

To account for timing uncertainties caused by factors like background system processes or temporary CPU load variations, both A* and Dijkstra were run 100 times on each grid. This ensured more reliable and averaged runtime measurements.

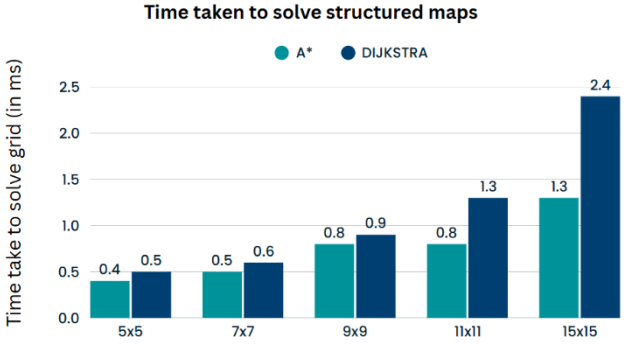


Figure 37: Time taken for A and Dijkstra to solve structured grids of sizes 4x5, 6x6, 7x7, 9x9 and 12x12 (in ms)*

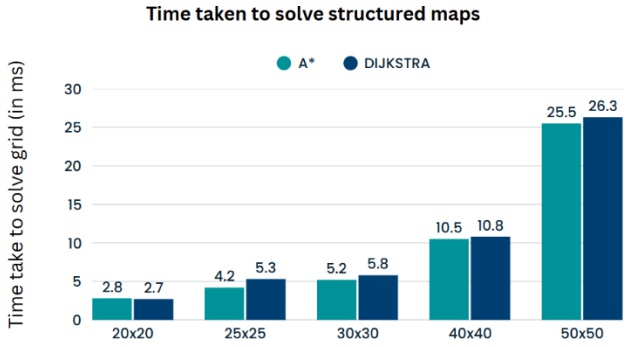


Figure 38: Time taken for A and Dijkstra to solve structured grids of sizes 15x15, 18x18, 25x25, 35x35 and 50x50 (in ms)*

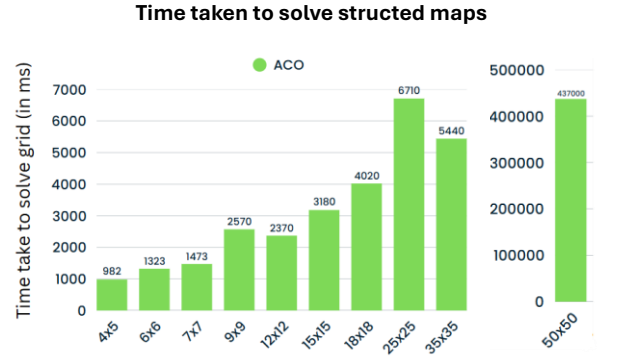


Figure 39: Time taken for ACO to solve structured grids of sizes 4x5, 6x6, 7x7, 9x9, 12x12, 15x15, 18x18, 25x25, 35x35 and 50x50 (in ms)

The parameters for the ACO algorithm were chosen based on initial experimentation and practical considerations, but they were not finely tuned to each specific grid. These parameters, which were set somewhat arbitrarily at first, worked well for smaller grids but proved suboptimal for larger ones. The initial settings - 100 ants, 20 iterations, a pheromone-to-heuristic weighting of 1.0, an evaporation rate of 10%, and a pheromone deposit constant of 2 - resulted in successful pathfinding for smaller maps. However, as grid size increased, these parameters led to significant performance issues, particularly on the

largest grid sizes, where the algorithm struggled to effectively explore the search space, requiring over 7 minutes to solve the problem, compared to previously less than 7 seconds (Figure 39). Figure 40 illustrates the optimal path generated by A* and Dijkstra on Grid 7, achieving a total path cost of 47. In contrast, Figure 41 shows the path produced by ACO on the same grid, resulting in a higher total cost of 53, reflecting the suboptimal convergence typical of this approach.

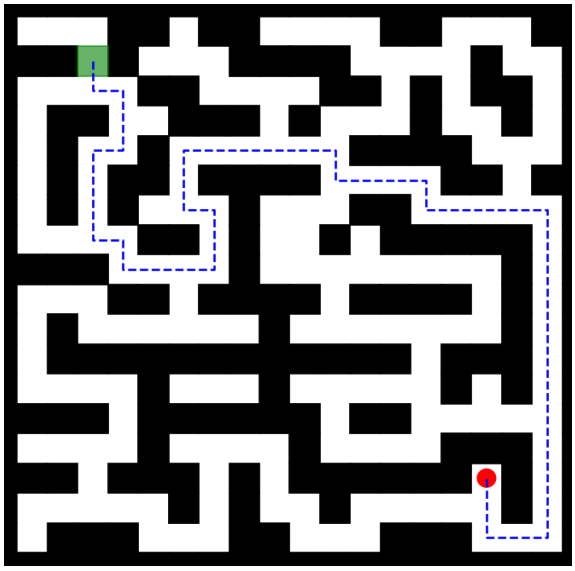


Figure 40: Grid #7: 18x18 sized grid showing the optimal path (cost of 47) from start (green square) to end (red circle)

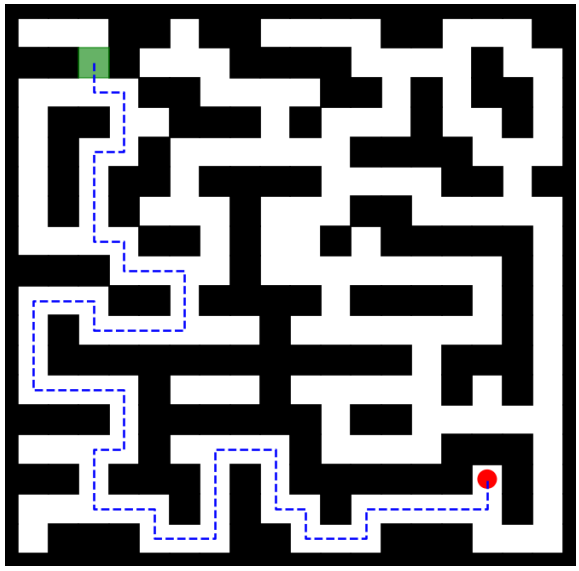


Figure 41: Grid #7: 18x18 sized grid showing the suboptimal ACO path (cost of 53) from start (green square) to end (red circle)

The cost achieved using the original ACO parameters can be viewed in Figure 42. The 50x50 grid does not have an entry, as the algorithm initially failed to find a path entirely. With only 100 ants and 20 iterations, the colony could not explore the grid sufficiently, often getting trapped in local optima or missing the goal altogether. Attempts to improve performance using slightly higher parameter values (e.g., 200-500 ants, 50-500 iterations, or moderate increases in pheromone/heuristic weightings) also failed to produce a valid path. Only with a substantial increase in search effort - specifically, 1000 ants over 1000 iterations, pheromone and heuristic importance set to 2.0, an evaporation rate of 20%, and a pheromone deposit constant of 2 - was the model able to reliably discover the optimal path. However, these more demanding settings came at the cost of significantly increased computation time, as demonstrated by the 50x50 grid in Figure 38.

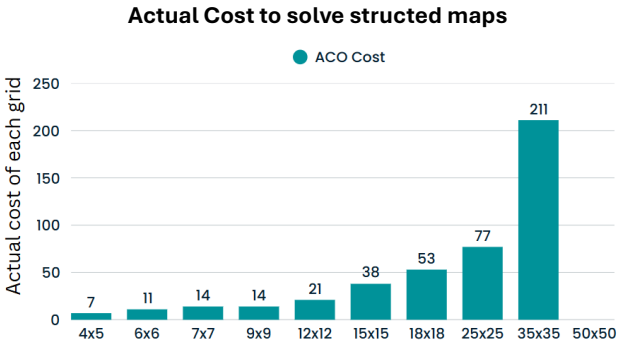


Figure 42: ACO cost to solve structured grids of sizes 4x5, 6x6, 7x7, 9x9, 12x12, 15x15, 18x18, 25x25, 35x35 and 50x50

In comparison to ACO, algorithms A* and Dijkstra managed to solve all grids with the optimal cost in a fraction of the time.

ONE CELL WIDE PATHS – CNN

The grids used in this section are consistent with those previously employed for A*, Dijkstra and ACO algorithm, and the optimal path for the grids can be found in Appendix V, Figure 85 and 86 respectively.

To create grids for the structured CNN navigation approach, a maze generation algorithm is used that starts with a grid full of walls. The maze is carved out by recursively removing walls between cells, ensuring that all open spaces are interconnected. The carving process uses random direction choices and moves two units at a time in either horizontal or vertical directions. This guarantees that each newly created open path is separated by at least one wall, avoiding big open spaces. The recursive nature of this process ensures that all open spaces are

connected, forming a valid and accessible maze layout.

The *add_loops* function further enhances the complexity of the maze by introducing loops. It iterates over the maze grid and, with a certain probability defined by *loop_chance*, removes walls between adjacent open spaces, thereby creating loops. This additional randomness mimics more dynamic maze environments and introduces greater variability for training the CNN. Each generated grid is then solved using the A* algorithm, and the resulting path is used as ground truth to train the CNN, helping it learn optimal navigation strategies.

The sample grid features a starting position at (2,2) and a finishing position at (6,6). After generating 10,000 mazes and spending approximately one hour on testing, training, and maze generation, the resulting output heatmap was produced, as shown in Figure 43. Training was halted after 12 epochs due to the *EarlyStopping* callback, which monitors validation loss and prevents overfitting.

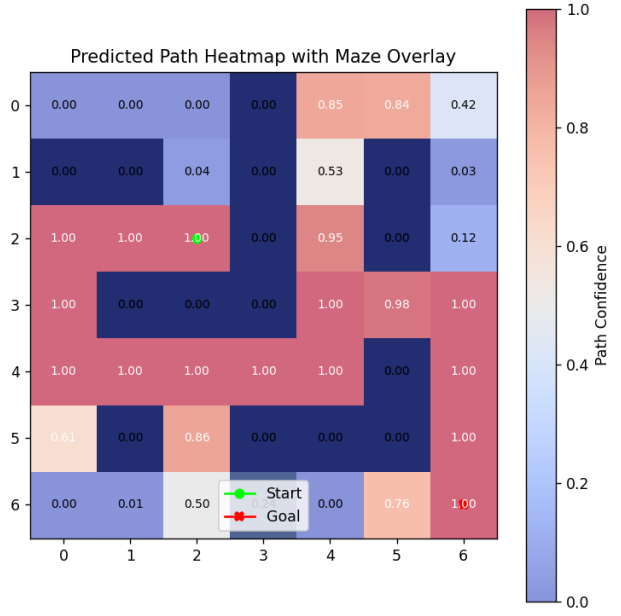


Figure 43: Output heatmap of the CNN for the 7×7 grid generated from 10,000 test mazes, illustrating the predicted navigation path

While the figure represents the optimal path with at least 98% confidence, the CNN also classifies additional cells as part of the path. Notably, the CNN correctly learns that paths cannot cross walls, classifying all wall cells with a 0% chance of being part of the path. When the number of mazes generated is reduced to 1,000, the total processing time drops to just over 5 minutes, with the training completing in 7 epochs for the 7×7 grid. However, this reduction comes at the cost of lower confidence, as demonstrated in Figure 44.

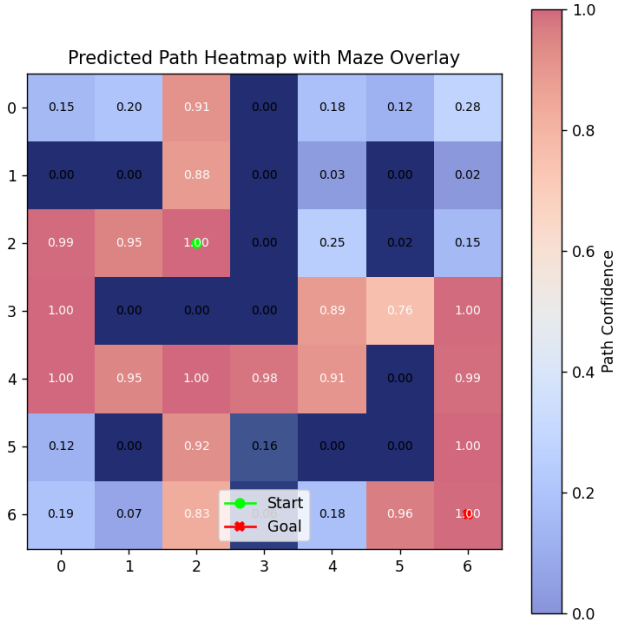


Figure 44: Output heatmap of the CNN for the 7×7 grid generated from 1,000 test mazes, illustrating the predicted navigation path with reduced confidence

The optimal path remains clearly identifiable in Figure 44 - despite some peripheral exploration. With all other parameters held constant, the increase in grid size from 7x7 to 9x9 (as shown in Figure 45), results in a slightly longer processing time of 10 minutes, with the training completing in 9 epochs.

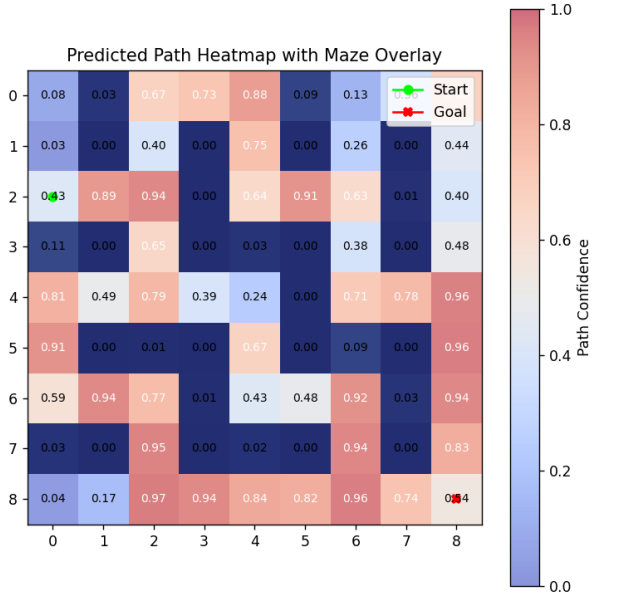


Figure 45: Output heatmap of the CNN for the 9×9 grid generated from 1,000 test mazes, illustrating the predicted navigation path

After increasing the number of generated maze samples to 10,000, the script completed maze creation and CNN training in approximately 1 hour and 40 minutes, with the training completing in 13 epochs. Figure 46 presents the resulting heatmap, where cells belonging to the optimal path exhibit a significantly higher likelihood, while traversable cells not on the optimal route display reduced path probabilities.

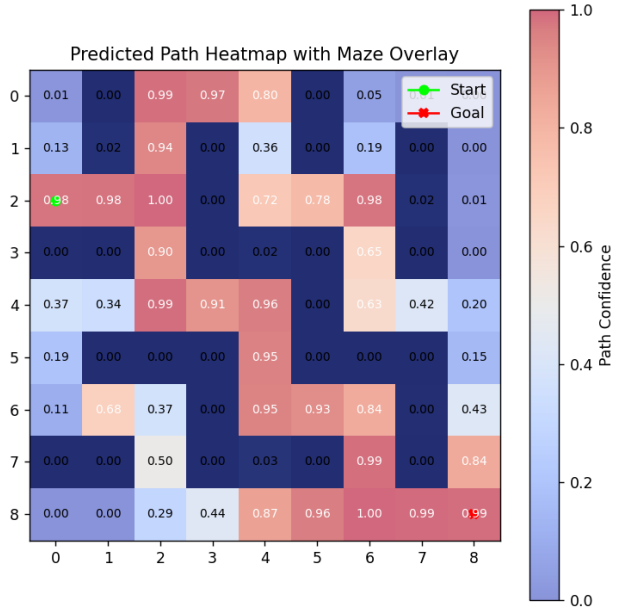


Figure 46: Output heatmap of the CNN for the 9×9 grid generated from 10,000 test mazes, illustrating the predicted navigation path with increased confidence

TWO CELL WIDE PATHS

The previously tested grids featured paths that were only one cell wide, thereby restricting movement to cardinal directions and disallowing diagonals. Since both ACO and the CNN-based approach exhibited limitations under cardinal movement conditions, the next set of experiments focused solely on A* and Dijkstra algorithms. For this purpose, five new grids were generated with sizes ranging from 15×15 to 50×50, each incorporating two-cell-wide paths to support more flexible navigation. Figure 47 illustrates an example of grid 11, sized 15x15, with the optimal path, having a cost of 15.7, highlighted. This grid serves as a representative example, and the remaining grids for this section can be found in Appendix VI. While ACO previously exhibited limitations, the ACO path for grid 11 is shown in Figure 48, clearly demonstrating how the algorithm struggles in open spaces when using the same parameters as initially.

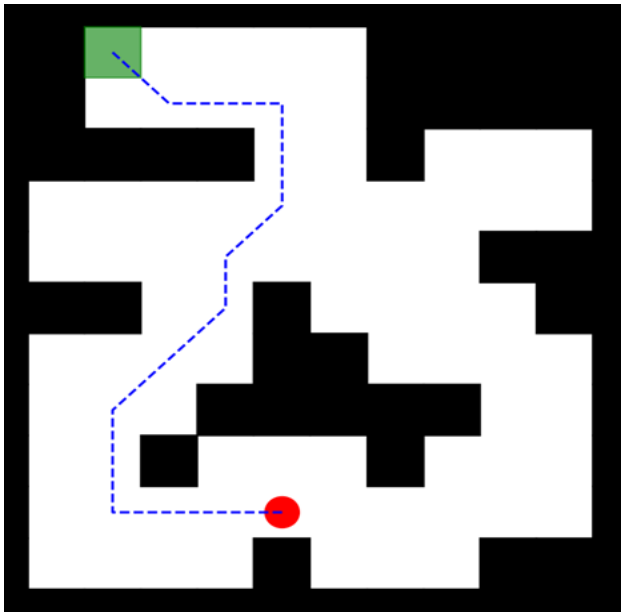


Figure 47: Grid #11: 15x15 sized grid showing the optimal path from start (green square) to end (red circle)

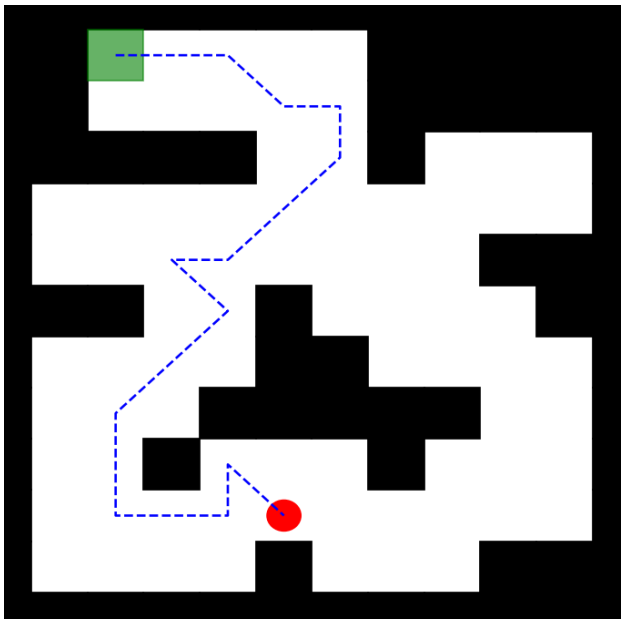


Figure 48: Grid #11: 15x15 sized grid showing the ACO path from start (green square) to end (red circle)

The optimal path costs computed by both A and Dijkstra are illustrated in Figure 49, confirming that both algorithms reliably found the shortest paths. The corresponding computation times are shown in Figure 50.

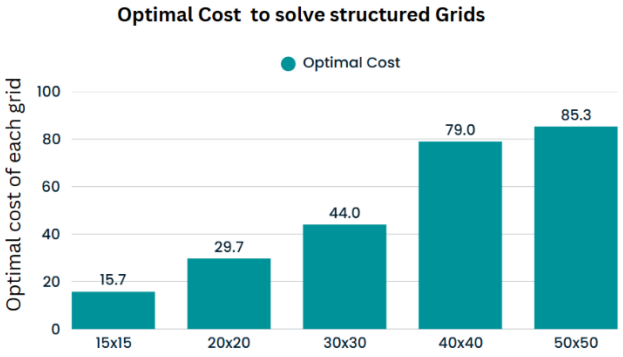


Figure 49: Optimal cost to solve structured grids of sizes 15x15, 20x20, 30x30, 40x40 and 50x50 containing two cell wide paths

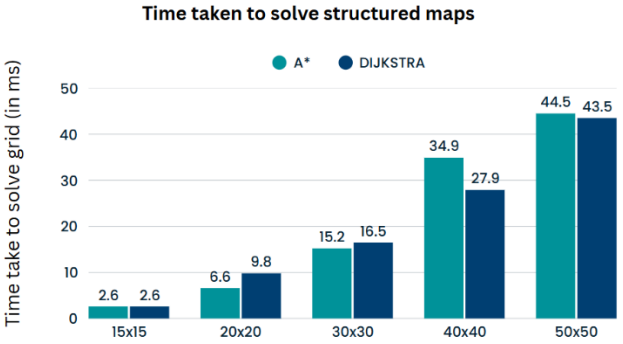


Figure 50: Time taken for A and Dijkstra to solve structured grids of sizes 15x15, 20x20, 30x30, 40x40 and 50x50 containing two cell wide paths (in ms)*

UNSTRUCTURED GRIDS

To further assess the robustness and adaptability of the navigation algorithms, the evaluation was extended beyond structured grid environments to include unstructured maps with randomly distributed obstacles. Unlike the previous maze-like configuration with clearly defined paths, these irregular grids introduce increased complexity, including unpredictable obstacle patterns. This shift aims to simulate more realistic and challenging conditions, thereby providing a more comprehensive understanding of the algorithms' performance in dynamic and less constrained environments.

A* AND DIJKSTRA

To evaluate algorithm performance under increasing complexity, grid maps ranging in size from 5×5 to 50×50 were generated. Grid 19 is shown in Figure 51, with a path cost of 16, while Figures 52 and 53 illustrate how A* and Dijkstra algorithms, respectively, evaluate the search space in search of the optimal solution. To further illustrate how A* and Dijkstra explore unstructured grids, Figures 96-122 in the Appendix VII present optimal paths for each grid, along with exploration heatmaps for each algorithm.

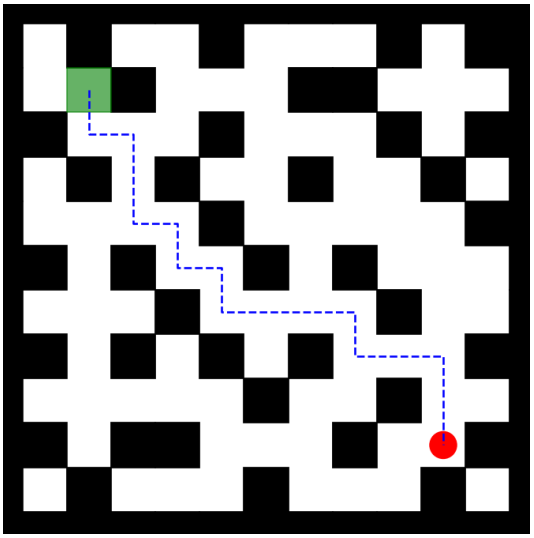


Figure 51: Grid #19: 11x11 sized grid showing the optimal path from start (green square) to end (red circle)

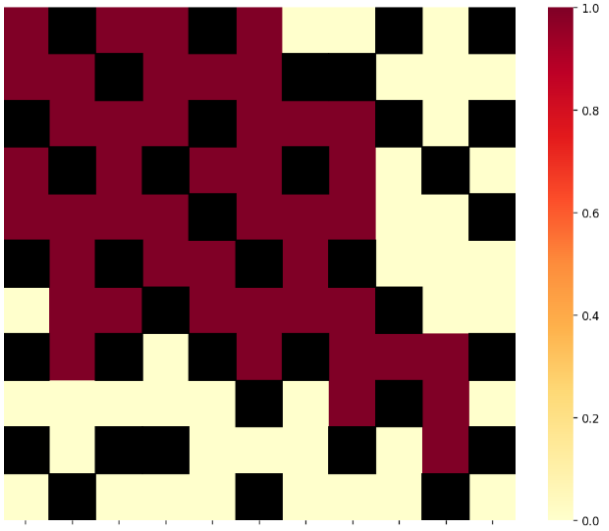


Figure 52: Grid #19: Exploration heatmap of A for a 11x11 sized grid*

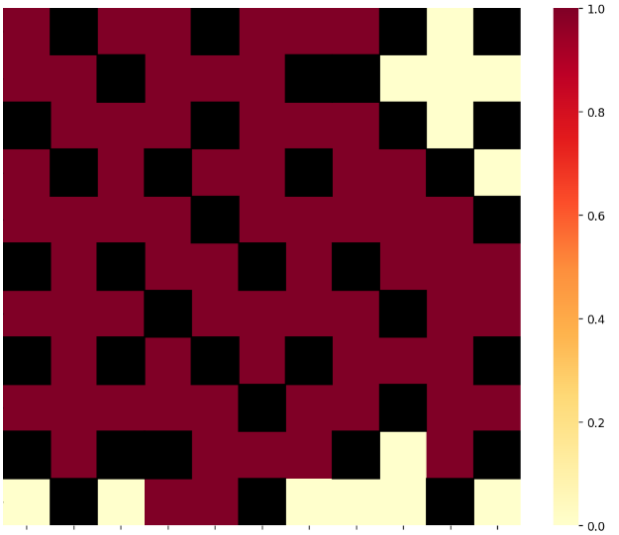


Figure 53: Grid #19: Exploration heatmap of Dijkstra for a 11x11 sized grid

Both A and Dijkstra successfully identified the optimal path costs across all grids, as shown in Figure 54. The corresponding computational times are presented in Figures 55 and 56.

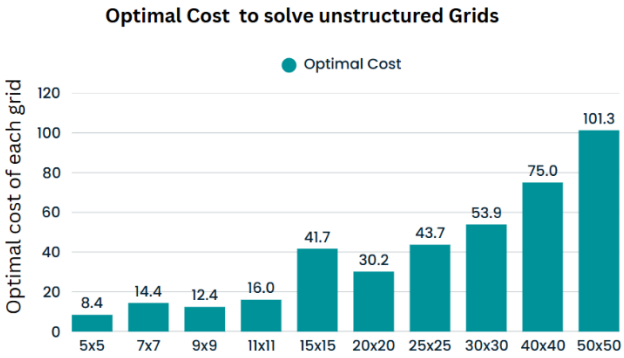


Figure 54: Optimal cost to solve unstructured grids of sizes 5x5, 7x7, 9x9, 11x11, 15x15, 20x20, 25x25, 30x30, 40x40 and 50x50

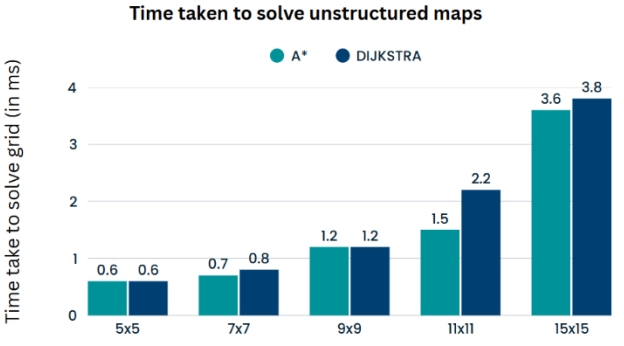


Figure 55: Time taken for A and Dijkstra to solve unstructured grids of sizes 5x5, 7x7, 9x9, 11x11 and 15x15 (in ms)*

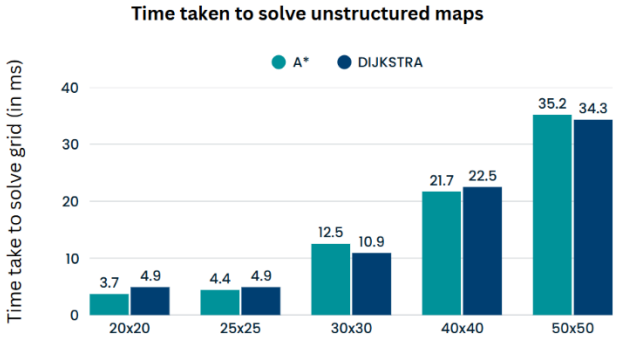


Figure 56: Time taken for A and Dijkstra to solve unstructured grids of sizes 20x20, 25x25, 30x30, 40x40 and 50x50 (in ms)*

This was not included in the previous section, as the heatmaps for unstructured grids were deemed more informative in comparison to those for structured grids.

CNN

The grids used in this section are consistent with those previously employed for A* and Dijkstra algorithm, and the optimal path for the grids can be found in Appendix VII, Figures 96, 99, 102 and 105.

For unstructured CNN testing, grids were generated using a randomized obstacle placement strategy instead of structured maze generation. Each grid begins as a fully traversable matrix, after which a certain percentage of cells are randomly turned into obstacles (walls), while ensuring that both the start and goal positions remain unblocked. To prevent trivial or unsolvable scenarios, each unstructured grid is validated using the A* algorithm. If no valid path exists between the start and goal, or if the path is shorter than twice the grid size - indicating insufficient complexity - the grid is discarded and regenerated.

The random distribution of walls results in a wider variety of path topologies compared to structured mazes, more closely simulating real-world environments where obstacles are often non-uniformly distributed. This makes unstructured CNN testing a valuable benchmark for assessing the model's generalization ability.

obstacle_chance controls the sparsity or density of walls in the grid - set to 0.3 to balance complexity and solvability. Each generated grid is used to compute the optimal path via the A* algorithm, which is then encoded into a label map for CNN training.

A total of 10,000 random 5×5 grids were generated with a wall probability of 0.3, and the entire process - including grid generation and CNN training - was completed in approximately 11 minutes. The CNN was trained for 15 epochs, with early stopping based on validation loss to ensure efficient training and to avoid overfitting. As shown in Figure 57, the resulting heatmap illustrates that the CNN successfully identifies viable paths despite the lack of structural regularity. Notably, the model learns to avoid randomly placed walls with high reliability, even in irregular layouts.

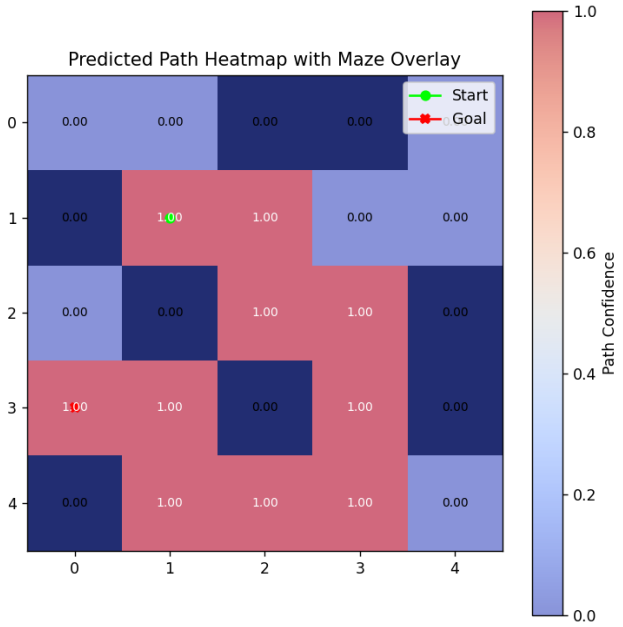


Figure 57: Output heatmap of the CNN for the unstructured 5×5 grid generated from 10,000 test mazes, illustrating the predicted navigation path with high confidence

Using the same parameters as before, the grid size was increased from 5×5 to 7×7. This adjustment resulted in relatively high path prediction confidence, as shown in Figure 58, with the full process taking approximately 15 minutes.

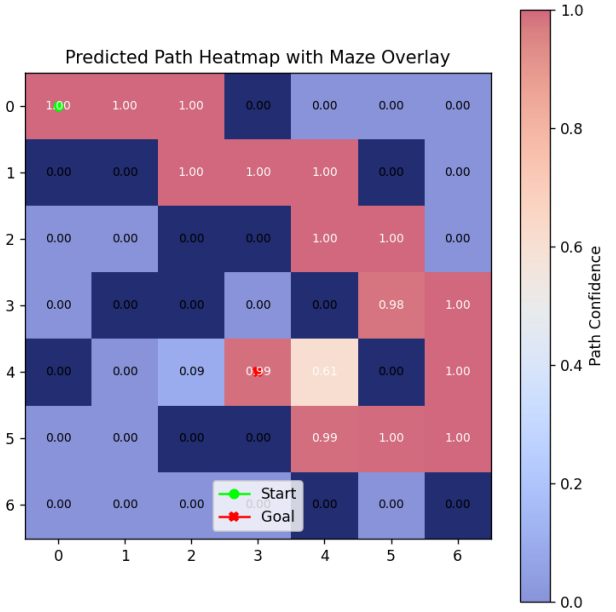


Figure 58: Output heatmap of the CNN for the unstructured 7x7 grid generated from 10,000 test mazes, illustrating the predicted navigation path with high confidence

When the grid size was further expanded to 9×9 with 20,000 generated mazes, the resulting heatmap (Figure 59) showed reduced path confidence after 40 minutes of processing. To address this, the number of training and testing grids was increased to 50,000, improving path quality as illustrated in Figure 60, which was produced after 100 minutes.

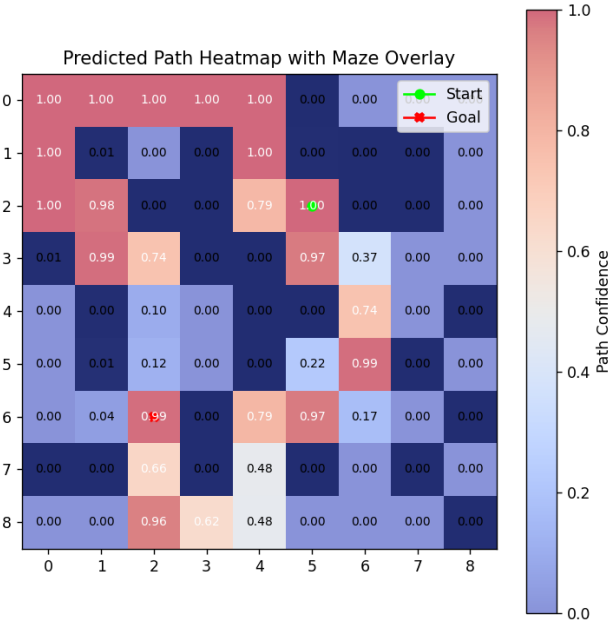


Figure 59: Output heatmap of the CNN for the unstructured 9×9 grid generated from 20,000 test mazes, illustrating the predicted navigation path with low confidence

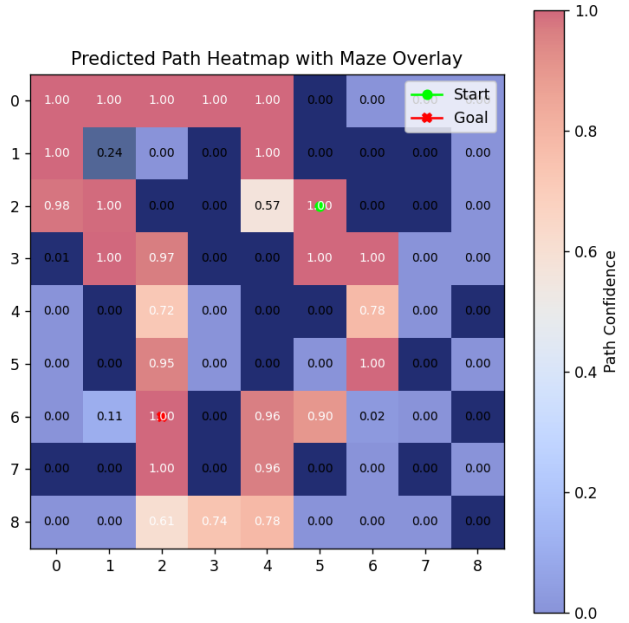


Figure 60: Output heatmap of the CNN for the unstructured 9×9 grid generated from 50,000 test mazes, illustrating the predicted navigation path with increased confidence

Finally, a larger 11×11 grid was tested using 50,000 generated samples. This more complex configuration required approximately 2.5 hours of processing to produce the heatmap shown in Figure 61. Despite the increased computational demand, the CNN maintained its ability to learn navigable paths with reasonable confidence.

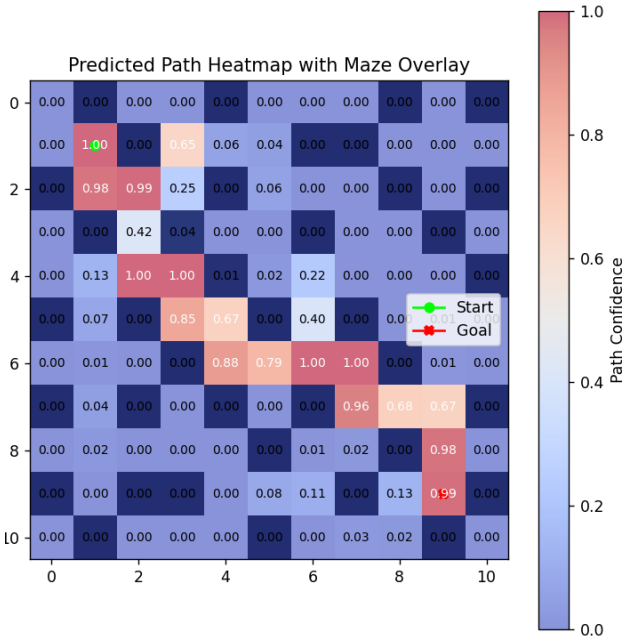


Figure 61: Output heatmap of the CNN for the unstructured 11×11 grid generated from 50,000 test mazes, illustrating the predicted navigation path with medium confidence

As with the structured approach, the trained model is saved after completion, enabling further real-time testing on similarly sized and formatted unstructured grids without the need for retraining.

REAL WORLD MAPS

Due to the high data requirements and computational cost of training CNNs, as well as the inefficiency of ACO for large-scale problems, this section focuses solely on the A* and Dijkstra algorithms.

Previously, grid maps were manually created - a time-consuming and labour-intensive process. To facilitate more robust testing and ensure comparability, benchmark grids of sizes 256×256, 512×512, and 1024×1024 were used in this section [21]. These maps represent real-world urban environments, specifically different scenarios of Berlin, London, Boston, Milan and Moscow (Appendix VIII).

Although it would have been valuable to evaluate differently sized AGVs on these real-world maps, this was not pursued due to the significant increase in computational demands. As a result, both the start and goal positions are represented by single 1×1 cells, which unfortunately makes them difficult to visually distinguish on larger maps.

For the larger grids, the number of iterations was reduced to 10 to decrease overall running time while still maintaining reliable results. This adjustment ensured that the tests were not significantly affected by timing uncertainties and allowed for more efficient testing while still providing averaged runtime measurements that account for these potential variations.

Figures 62, 64 and 66 present the optimal path costs obtained across all tested city environments. Figures 63, 65, and 67 show the corresponding computational times, measured in seconds, for each algorithm. Appendix VIII provides a detailed visual comparison of the optimal paths generated by A* and Dijkstra in each environment. Notably, both algorithms consistently identified the same optimal path in all cases. The appendix also includes exploration heatmaps that illustrate the areas evaluated by each algorithm during the search

process.

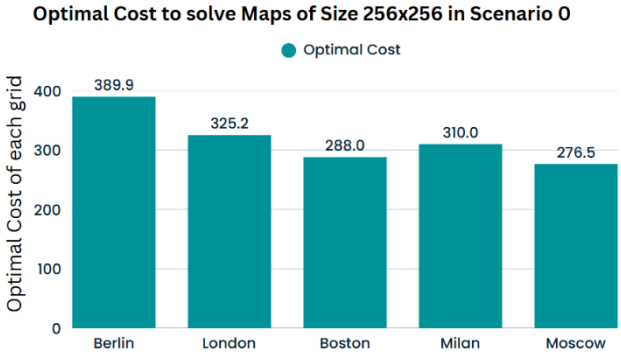


Figure 62: Optimal cost to solve real-world maps of Berlin, London, Boston, Milan and Moscow of sizes 256x256 in scenario 0

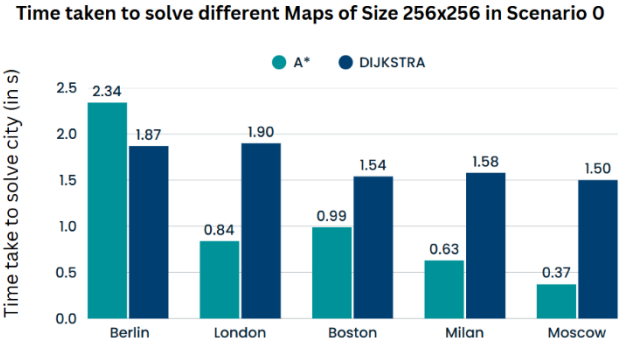


Figure 63: Time taken to solve real-world maps of Berlin, London, Boston, Milan and Moscow of sizes 256x256 in scenario 0 (in seconds)

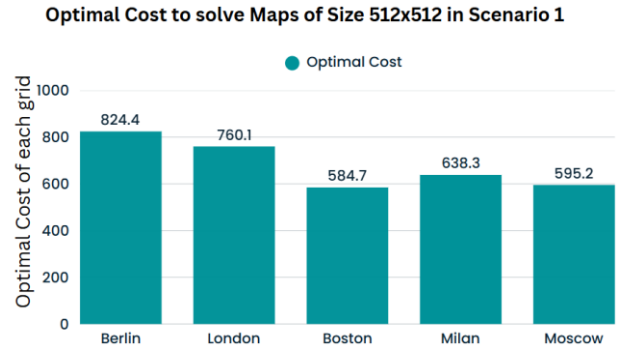


Figure 64: Optimal cost to solve real-world maps of Berlin, London, Boston, Milan and Moscow of sizes 512x512 in scenario 1

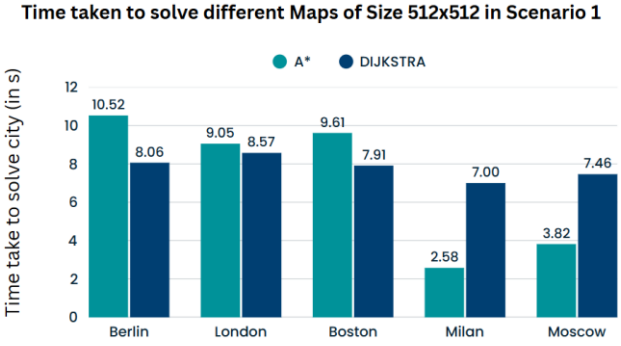


Figure 65: Time taken to solve real-world maps of Berlin, London, Boston, Milan and Moscow of sizes 512x512 in scenario 1(in seconds)

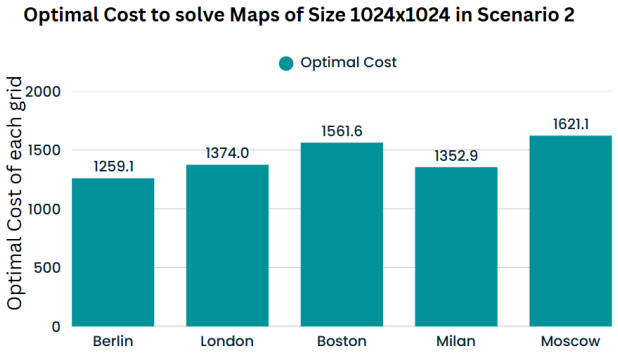


Figure 66: Optimal cost to solve real-world maps of Berlin, London, Boston, Milan and Moscow of sizes 1024x1024 in scenario 2

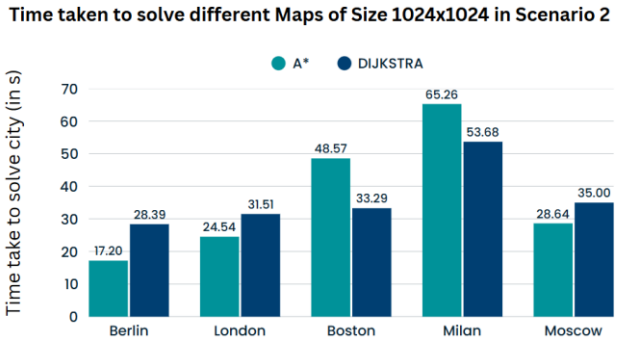


Figure 67: Time taken to solve real-world maps of Berlin, London, Boston, Milan and Moscow of sizes 1024x1024 in scenario 2 (in seconds)

Examining the computational times presented in Figures 63, 65 and 67, it is evident that Dijkstra's algorithm consistently produced more stable execution times compared to A*. While A* occasionally achieved very fast pathfinding, it also exhibited significant variability, sometimes taking considerably longer depending on the grid configuration. Figures 68 and 71 show the optimal paths for Grid 29 and Grid 38, which were successfully identified by both Dijkstra's algorithm and A*. Figures 69 and 72 represent good and poor A* exploration respectively, In contrast, the heatmaps for Dijkstra's algorithm, shown in Figures 70 and 73, demonstrate the algorithm's more uniform exploration of the search space.

Dijkstra's consistency arises from its exhaustive exploration strategy, which expands all traversable nodes until the goal is reached, making it a more predictable but computationally intensive pathfinding method. In contrast, A* attempts to reduce unnecessary exploration by using a heuristic to guide the search toward the goal. However, this advantage is highly dependent on the relative

positions of the start and goal nodes. A* performs particularly well when these nodes are aligned along the same axis but struggles with efficiency when the start and goal are positioned diagonally from each other, leading to more extensive search paths.

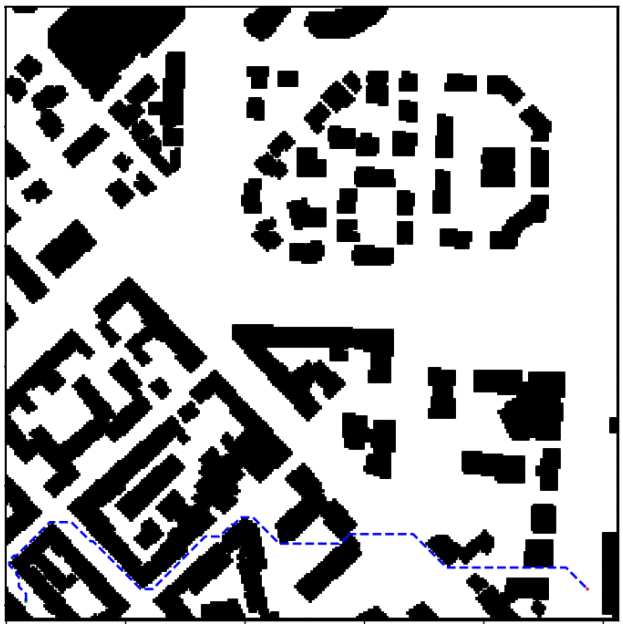


Figure 68: Grid #29: 256x256 sized grid representing Milan scenario 0 showing the optimal path from start (bottom-left) to end (bottom-right) [21]

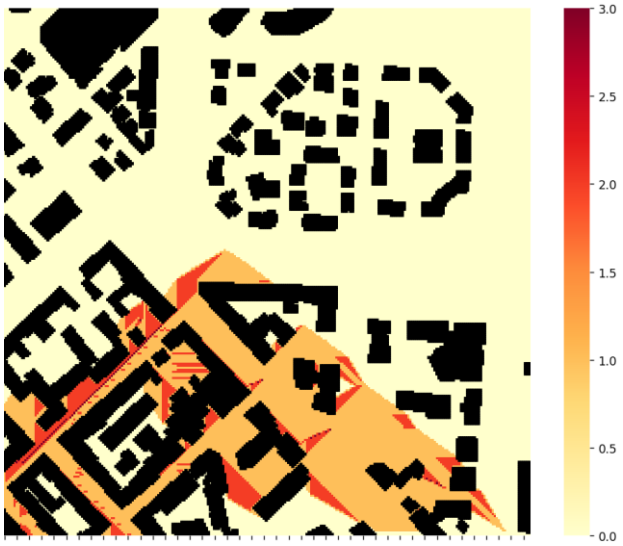


Figure 69: Grid #29: Heatmap of A for a 256x256 sized grid representing Milan scenario 0 with optimal path from start (bottom-left) to end (bottom-right) [21]*

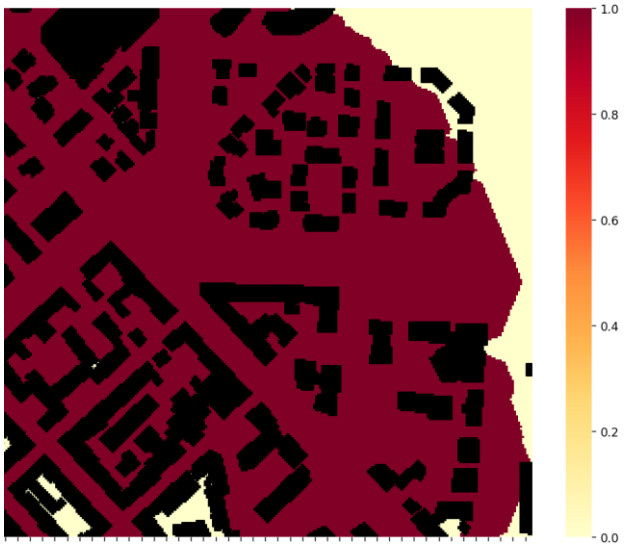


Figure 70: Grid #29: Heatmap of Dijkstra for a 256x256 sized grid representing Milan scenario 0 with optimal path from start (bottom-left) to end (bottom-right) [21]

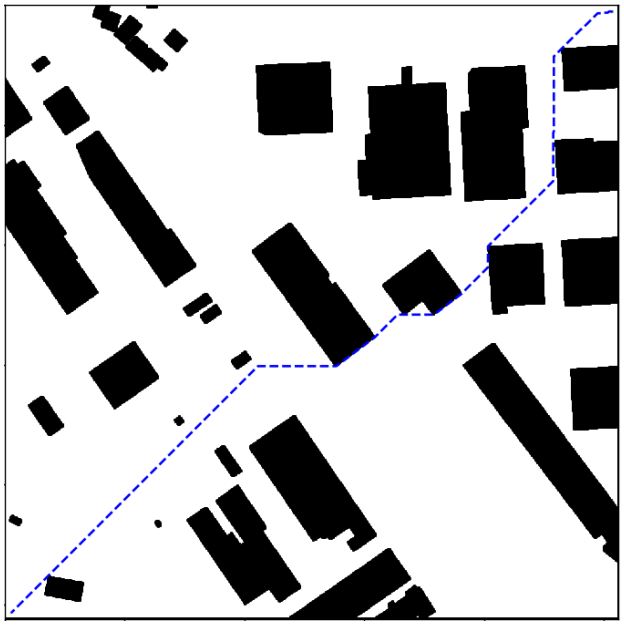


Figure 71: Grid #38: 1024x1024 sized grid representing Boston scenario 2 showing the optimal path from start (top-right) to end (bottom-left) [21]

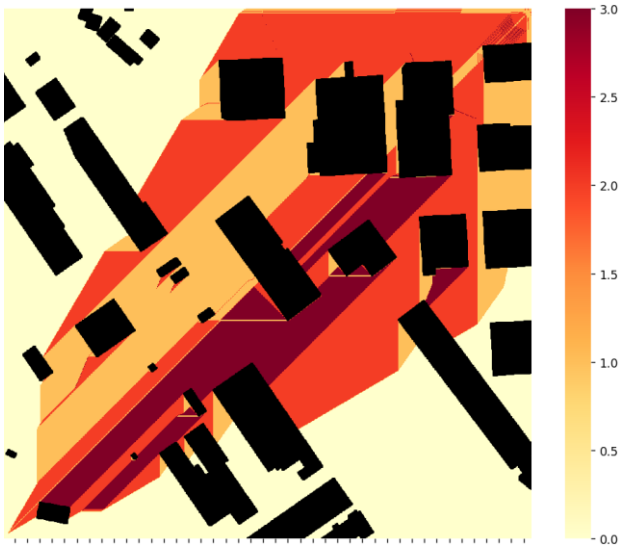


Figure 72: Grid #38: Heatmap of A for a 1024x1024 sized grid representing Boston scenario 2 with optimal path from start (top-left) to end (top-right) [21]*

Figure 73: Grid #38: Heatmap of A for a 1024x1024 sized grid representing Boston scenario 2 with optimal path from start (top-left) to end (top-right) [21]*

Appendix VIII includes the heatmaps for the remaining 13 grids of various sizes, showing the exploration patterns for both A* and Dijkstra's algorithms.

DISCUSSION

Braitenberg and line-following methods were not tested on the grid environments used for AGV navigation, as they rely on reactive, sensor-based behaviour rather than map-based planning, making them unsuitable for structured grid-based evaluation.

As demonstrated by the graphs in the Results section, the performance of ACO is highly sensitive to the chosen parameter configuration. Increasing the size of the ant colony improves the quality of the solution and enables convergence toward the optimal path length. However, this benefit comes at the cost of significantly higher computational time. Although ACO was only evaluated on structured single-cell grids, its computational inefficiency and parameter sensitivity make it suboptimal for real-time AGV navigation.

Similarly, the results from the CNN model show that output quality is heavily influenced by the number of test grids. As grid size increases, the performance of the CNN model deteriorates, and the computational demands for generating sample grids, training, and evaluation rise substantially. While a trained CNN model can, in theory, be deployed for near real-time inference, its utility is limited by inconsistent cell-level confidence scores. This makes it difficult to reliably construct an optimal path based solely on predicted cell probabilities. CNNs were tested on both structured single-cell and unstructured paths, but due to their high computational cost and inconsistent prediction reliability, they were ultimately found to be unsuitable for real-time AGV navigation.

The most promising methods are A* and Dijkstra. Across single and wider structured, unstructured and real world grids, no clear one optimal solution emerges. While ACO only converges towards the optimal solution, A* and Dijkstra always provide an optimal solution, as proven by all the testing in the Results section.

Dijkstra's time complexity ($O(n^2)$) becomes a significant bottleneck in large or sparse environments (e.g., 1024×1024 real-world maps), where it tends to explore many irrelevant nodes. In contrast, A* attempts to reduce this overhead by using a heuristic to guide the search, but this

advantage can diminish if the heuristic is poorly calibrated, as in environments with non-uniform costs.

Additionally, computational time analysis on real-world maps (Figures 63, 65, and 67) indicates that Dijkstra's algorithm provides more consistent execution times, owing to its exhaustive search strategy, which guarantees optimal paths at the cost of increased computational overhead. A*, on the other hand, can be faster when start and goal nodes are aligned along the same axis (Figure 69), but its performance becomes more variable when nodes are positioned diagonally (Figure 72), reflecting a key trade-off between efficiency and consistency.

For smaller grids (e.g., <50×50), the runtime difference between the two algorithms is negligible, making Dijkstra a viable alternative in scenarios where heuristic design is challenging or unnecessary.

The results obtained for A* and Dijkstra closely align with findings from Benchmark studies like the PathBench study. The source confirm that these algorithms consistently produce optimal paths, as demonstrated by PathBench's 0.00% path deviation. Furthermore, the observed time complexity bottleneck for Dijkstra ($O(n^2)$) in larger environments is consistent with PathBench's data, which shows significantly longer execution times for Dijkstra as map sizes increase (e.g., 9.928s for Dijkstra versus 3.816s for A* on 512×512 city maps).

The trade-offs between the implemented navigation methods are summarized in Appendix II, Table 4, considering factors such as efficiency, optimality, heuristic dependence, scalability for different AGV sizes, and support for diagonal movement.

Overall the project aim of building an interactive LEGO-based AGV has been achieved. Below are the outcomes of each objective associated with this project.

1. **Colour Classification:** This objective was met. The LEGO colour sensor reliably classified colours within a 2-6 cm range under varying lighting and brightness conditions, demonstrating high accuracy. This ensured consistent performance in real-world scenarios, aligning with the target accuracy threshold.
2. **Database Connectivity:** This objective was met. A lightweight .csv database was implemented to store key information about the navigation grid, including grid size, start positions, and obstacle locations. This simple yet effective approach provided a flexible foundation for retrieving critical data during AGV operation.
3. **Navigation Algorithm Integration:** This objective was met. Several navigation methods were explored, with A* and Dijkstra's algorithm emerging as the most suitable choices for the LEGO-based AGV. These algorithms provided efficient, reliable pathfinding within the constraints of the small-scale platform, balancing computational cost and path optimality.
4. **Voice Command Integration:** This objective was met. OpenAI's Whisper speech-to-text model was integrated, offering two deployment options: a user-friendly Docker-based WebUI and a local development environment for advanced users. This flexibility allows for a seamless voice command experience, aligning with the project's hands-free interaction goals.

FUTURE WORK

The current AGV configuration, with a single distance sensor for obstacle detection and a colour sensor for object identification, limits its ability to make context-aware navigation decisions. Adding multiple distance sensors or more advanced systems like LIDAR would improve real-time obstacle detection, reduce unnecessary path recalculations, and enhance real-time responsiveness. Additionally, integrating data from multiple sensor types, such as camera and ultrasonic sensors, through sensor fusion and multi-modal perception, could further enhance the AGV's situational awareness. This approach would allow for more accurate localization, object detection, and environmental understanding, potentially enabling more sophisticated path planning and obstacle avoidance strategies. Such

improvements would particularly benefit A*, allowing more precise heuristic adjustments based on richer local data, reducing computational overhead and improving overall navigation efficiency.

Dijkstra's algorithm, while exhaustive, could also benefit from improved sensing by reducing exploration of irrelevant nodes, partially mitigating its $O(n^2)$ complexity in sparse environments.

With more comprehensive sensor input, the AGV could incorporate Simultaneous Localization and Mapping (SLAM) to map its surroundings and localize itself simultaneously, removing the need for predefined grids. This approach would allow for more adaptive path planning in unfamiliar environments, particularly in dynamic settings like warehouse automation or search-and-rescue missions.

To validate these improvements, building a test environment to evaluate real-world performance under various conditions would be essential for refining these algorithms and verifying their practical viability. Such a setup would allow for controlled experimentation with different sensor configurations, dynamic obstacles, and varying terrain types, providing critical insights into the AGV's ability to navigate complex, real-world environments. Additionally, this approach would enable the testing of SLAM systems and multi-sensor fusion, ensuring that the AGV can effectively adapt to the unpredictable nature of real-world operations.

RL methods and CNNs would also benefit from advanced real-time sensor data and could be further explored to assess their potential for dynamic path optimization and obstacle avoidance. RL methods, in particular, could use this data to continuously improve the AGV's navigation strategies through trial and error, potentially outperforming classical algorithms in complex, unpredictable environments. CNNs could enhance object recognition and terrain classification, providing richer context for path planning decisions.

However, integrating these methods would significantly increase computational demands, requiring more powerful onboard processors or edge computing capabilities to maintain real-time performance. Additionally, advanced sensor arrays and machine learning approaches would necessitate robust data processing pipelines and efficient training methods to ensure reliable, low-latency decision-making.

To effectively utilize these approaches, significant time should be invested in collecting high-quality training data, as the performance of RL models and CNNs is heavily dependent on the quality and diversity of the data used for training.

Additionally, real-world deployment and scaling challenges should be addressed, including hardware durability, cost constraints, and integration with existing industrial systems. Ensuring robust performance in diverse, real-world environments will be critical for transitioning from prototype to production-ready AGVs.

Moreover, exploring hybrid methods that combine classical algorithms like Dijkstra with heuristic approaches - such as D* Lite - could further optimize AGV performance. For instance, making Dijkstra more goal-oriented through the incorporation of heuristics could reduce unnecessary search space exploration, improving efficiency in large environments while maintaining path optimality.

Ultimately, combining classical pathfinding algorithms like A* and Dijkstra with more sophisticated machine learning approaches could offer a comprehensive, adaptable navigation system capable of handling the diverse challenges encountered in real-world AGV operations.

CONCLUSION

The project successfully achieved its aim of building an interactive LEGO-based AGV that simulates real-world warehouse operations. Several navigation methods were implemented and evaluated, including Braitenberg vehicles, A*, Dijkstra, CNN, and ACO, with Soft Actor-Critic being reviewed but excluded due to its high complexity and training requirements. Braitenberg vehicles were included in a simplified form but proved unreliable in complex environments due to hardware limitations and inconsistent obstacle detection. The performance evaluation on structured,

unstructured, and real-world grids highlighted significant trade-offs between the algorithms. ACO was found to be highly sensitive to parameter configuration and computationally inefficient for real-time AGV navigation, often requiring extensive computation to converge towards optimal solutions. CNNs, while capable of visual feature extraction for navigation, demonstrated limitations in generalisation and suffered from high computational demands and inconsistent path prediction reliability, making them unsuitable for real-time AGV navigation within the project's scope.

The most promising methods for the constrained LEGO platform were A* and Dijkstra's algorithm, both of which consistently provided optimal paths across all tested grid types. A* balanced optimality with higher efficiency due to heuristic guidance, particularly when start and goal nodes were favourably aligned. Dijkstra, in contrast, employed an exhaustive search strategy, guaranteeing optimal paths at the cost of higher computational overhead, particularly noticeable in large or sparse environments where its $O(n^2)$ complexity became a bottleneck. Dijkstra did, however, demonstrate more consistent execution times compared to A*'s variability. For smaller grids, the performance difference between A* and Dijkstra was negligible. These findings align with benchmark studies like PathBench, confirming the algorithms' ability to produce optimal paths and Dijkstra's efficiency bottleneck in larger maps. Ultimately, A* emerged as the preferred choice for most scenarios due to its balance of optimality and efficiency, although Dijkstra remains a viable alternative in small or static environments or where heuristic design is challenging.

The project successfully met its specific objectives: Colour Classification was achieved with high accuracy using the LEGO SPIKE colour sensor. Database Connectivity was established using a lightweight .csv database for storing navigation grid information and object locations. Navigation Algorithm Integration was successful, with A* and Dijkstra identified as the most suitable algorithms for the platform. Voice Command Integration was implemented via OpenAI's Whisper model, offering flexible user interaction options.

Future work recommendations include enhancing the AGV's real-time environmental awareness through the addition of multiple distance sensors or more advanced systems like LIDAR and integrating data using sensor fusion. Implementing SLAM would remove the need for predefined maps, improving adaptability in unfamiliar environments. Building a dedicated test environment is essential for evaluating real-world performance under various conditions. Further exploration of RL methods and CNNs is warranted with access to more advanced hardware, better sensor data, and comprehensive training datasets. Exploring hybrid methods that combine classical algorithms like Dijkstra with heuristic approaches could further optimise navigation. Future research should focus on advanced environmental sensing methods, hybrid approaches, and rigorous real-world validation to bridge the gap between simulation and practical deployment.

PROJECT REVIEW

During the initial stages, integrating the LEGO SPIKE Prime, which runs on a lightweight Python variant, with a full Python environment required extensive coordination. After multiple meetings with technicians and direct consultations with LEGO support, the decision was made to flash the default LEGO operating system and replace it with Pybricks. This approach enabled the use of VS Code and smoother integration into the broader project workflow. In future iterations, this process could be streamlined by leveraging pre-configured firmware or automated setup scripts, reducing the need for extensive technical support and minimizing setup time.

A thorough project plan, supported by weekly meetings with my university supervisor and biweekly meetings with my company supervisor, kept the development on track. This approach streamlined progress by facilitating the implementation of various navigation methods. The critical application of engineering knowledge enabled rapid diagnosis and resolution of challenges related to algorithmic performance.

ACKNOWLEDGEMENTS

Generative Artificial Intelligence (GenAI) was utilized for initial research guidance and language refinement and code debugging, including rewording and phrasing adjustments, to enhance the clarity and precision of the text and code. This approach ensured the articulation of complex ideas and alignment with the required academic standards.

I would like to express my sincere gratitude to my academic supervisor, Uriel Martinez Hernandez, for their invaluable guidance, support, and mentorship throughout this project. Their insights and encouragement have been instrumental in shaping this work.

Finally, I would like to express my heartfelt gratitude to my family, girlfriend, and peers, whose unwavering support, guidance, and belief in my potential have been crucial in overcoming challenges and reaching important milestones throughout my academic journey.

REFERENCES

[1] X. Yang, R. V. Patel and M. Moallem, "A Fuzzy–Braitenberg Navigation Strategy for Differential Drive Mobile Robots," *Journal of Intelligent and Robotic Systems,* vol. 47, no. 2, pp. 101-124, 21 September 2006.

[2] X. Wang, J. Lu, F. Ke, X. Wang and X. Wang, "Research on AGV task path planning based on improved A* algorithm," *Virtual Reality & Intelligent Hardware,* vol. 5, no. 3, pp. 249-265, 16 June 2023.

[3] M. Zhong, J. Shi and X. Zhou, "Research on AGV Intelligent Parking Navigation Path Planning Based on A* Algorithm," in *In Proceedings of the 2023 7th International Conference on Electronic Information Technology and Computer Engineering*, 2023.

[4] Aizat, Muhammad, A. Azmin and W. Rahiman, "A Survey on Navigation Approaches for Automated Guided Vehicle Robots in Dynamic Surrounding," April 2023. [Online]. Available: https://www.researchgate.net/publication/369931624_A_Survey_on_Navigation_Approaches_for_Automated_Guided_Vehicle_Robots_in_Dynamic_Surrounding. [Accessed March 2025].

[5] saylor.org, "Overview of A* Search and Analysis of Performance," [Online]. Available: https://learn.saylor.org/mod/page/view.php?id=80817. [Accessed May 2025].

[6] Sun, Yinghui, Fang, Ming, Su and Yixin, "AGV Path Planning based on Improved Dijkstra Algorithm," Jan 2021. [Online]. Available: https://www.researchgate.net/publication/348597712_AGV_Path_Planning_based_on_Improved_Dijkstra_Algorithm. [Accessed April 2025].

[7] M. McBride, "Dijkstra's algorithm," 16 October 2023. [Online]. Available: https://graphicmaths.com/computer-science/graph-theory/dijkstras-algorithm/. [Accessed April 2025].

[8] I. A. Ayman H. Tanira, "An Improved Sampling Dijkstra Approach for Robot Navigation and Path Planning," December 2023. [Online]. Available: https://www.researchgate.net/publication/376351454_An_Improved_Sampling_Dijkstra_Approach_for_Robot_Navigation_and_Path_Planning. [Accessed April 2025].

[9] M. Parulekar, V. Padte, T. Shah, K. Shroff and R. Shetty, "Automatic vehicle navigation using Dijkstra's Algorithm," in *Proc. IEEE Int. Conf. Control Applications (CCA)*, 2013.

[10] B. B. Elallid, N. Benamar, M. Bagaa and Y. Hadjadj-Aoul, "Enhancing Autonomous Driving Navigation Using Soft Actor-Critic," *Future Internet,* vol. 16, no. 7, p. 238, 4 July 2024.

[11] H. Guo, Z. Ren, J. Lai, Z. Wu and S. Xie, "Optimal navigation for AGVs: A soft actor–critic-based reinforcement learning approach with composite auxiliary rewards," *Engineering Applications of Artificial Intelligence,* vol. 124, 21 June 2023.

[12] N. B. Nicholas Mohammad, "Soft Actor-Critic-based Control Barrier Adaptation for Robust Autonomous Navigation in Unknown Environments," arXiv Preprint, arXiv:2503.08479, 2025.

[13] F. Foroughi, Z. Chen and J. Wang, "A CNN-Based System for Mobile Robot Navigation in Indoor Environments via Visual Localization with a Small Dataset," *Robotics,* vol. 12, no. 3, p. 134, 26 August 2021.

[14] U. Aulia, I. Hasanuddin, M. Dirhamsyah and N. Nasaruddin, "A new CNN-BASED object detection system for autonomous mobile robots based on real-world vehicle datasets," *Heliyon,* vol. 10, no. 15, 29 July 2024.

[15] H. Liu, "AGV Path Planning based on Improved Ant Colony Algorithm," *Second International Conference on Algorithms, Microchips, and Network Applications (AMNA 2023),* vol. 12635, pp. 59-63, 28 March 2024.

[16] B. Jiang, Y. Liu, Z. Xu and Z. Chen, "Enhancing AGV path planning: An improved ant colony algorithm with nonuniform pheromone distribution and adaptive pheromone evaporation," *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science,* 24 February 2025.

[17] LEGO Education, “Exploring SPIKE™ Prime Sensors,” [Online]. Available: https://community.legoeducation.com/blogs/31/220. [Accessed April 2025].

[18] XRP, “Following the Line: Proportional Control with 2 Sensors,” [Online]. Available: https://introduction-to-robotics.readthedocs.io/en/latest/course/line_following/pcontrol2s.html. [Accessed April 2025].

[19] RanHam, “Inductive Guided Robot (Wire Guided),” 2019. [Online]. Available: https://www.youtube.com/watch?v=QLSjhAHZmGU. [Accessed April 2025].

[20] A. Radford, J. Kim, T. Xu, G. Brockman, C. McLeavey and I. Sutskever, “Robust speech recognition via large-scale weak supervision,” *International conference on machine learning,* pp. 28492--28518, 2023.

[21] N. Sturtevant, “Benchmarks for Grid-Based Pathfinding,” *Transactions on Computational Intelligence and AI in Games,* vol. 4, no. 2, pp. 144-148, 2012.

[22] F. Smaoui, “Smart Navigation in Warehouses: Dijkstra’s Algorithm for Mobile Robot Path Planning,” 9 June 2024. [Online]. Available: https://medium.com/@smaouifouad/smart-navigation-in-warehouses-dijkstras-algorithm-for-mobile-robot-path-planning-be8ec050ca8f. [Accessed April 2025].

[23] J. Liu, P. Xie, J. Liu and X. Tu, “Task Offloading and Trajectory Optimization in UAV Networks: A Deep Reinforcement Learning Method Based on SAC and A-Star,” *CMES-Computer Modeling in Engineering & Sciences,* vol. 141, no. 2, 27 September 2024.

[24] B. Zhang, X. Liang, W. Song and Y. Chen, “Multi-Dimensional AGV Path Planning in 3D Warehouses Using Ant Colony Optimization and Advanced Neural Networks,” arXiv preprint, arXiv:2504.01985, 2025.

APPENDICES

APPENDIX I - PROJECT COSTING

The project incurred minimal costs, as the LEGO SPIKE Prime kit - purchased by the company - was the only required hardware, costing approximately £444.99.

APPENDIX II - TABLES

Table 2: Dijkstra Table finding the shortest path from the start node to all nodes

Step	Distance from A	Visited (V)/ unvisited (U) sets
1	A → A: 0 A → B: ∞ A → C: ∞ A → D: ∞ A → E: ∞	V: {A} U: {B, C, D, E}
2	A → A: 0 A → B: 7 A → C: ∞ A → D: ∞ A → E: 1 (shortest)	V: {A, E} U: {B, C, D}
3	A → A: 0 A → B: 7 A → E → C: 3 (shortest) A → E → D: 8 A → E: 1	V: {A, C, E} U: {B, D}
4	A → A: 0	V: {A, B, C, E} U: {D}

	A→ E → C → B: 6 (shortest) A → E → C: 3 A → E → D: 8 A → E: 1	
5	A → A: 0 A→ E → C → B: 6 A → E → C: 3 A → E → D: 8 A → E: 1	V: {A, B, C, D, E} U: {}

Table 3: Table comparing Braitenberg Vehicles, A, Dijkstra, Soft Actor-Critic, Convolutional Neural Networks and Ant Colony Optimisation in terms of complexity, optimality, adaptability and real-world suitability*

Method	**Complexity**	**Optimality**	**Adaptability**	**Real-World Suitability**	**Notes/Explanation**
Braitenberg Vehicles	Very low, time complexity is O(1) due to reactive behaviour [1]	No [1]	Low [1]	Basic, simple robots only [1]	Direct sensor-motor links, no mapping, no planning [1]
A*	Moderate, time complexity worse case: $O(b^d)$, where b is the branching factor, and d is the depth of the optimal solution [2]	Yes (if heuristic is admissible) [4]	Limited (static or known maps) [2]	Strong in structured environments (e.g., warehouses) [2] [3]	Uses heuristics for efficiency, struggles with dynamic obstacles unless extended (e.g., D* Lite) [2] [4]
Dijkstra's Algorithm	Moderate-High, time complexity: $O(n^2)$ [8]	Yes [8]	Low (requires recomputing) [8]	Suitable for small graphs or simple layouts [6] [8]	Explores exhaustively without heuristic efficient only for small graphs [6] [8]
Soft Actor-Critic (SAC)	High, time complexity depends on the cost of forward and backward passes through the actor and two critic networks over multiple episodes and steps [11]	Probabilistic optimality (not guaranteed) [11]	High [11]	Research-phase; not yet widely practical [11]	"Probabilistic" means SAC stochastically samples actions to encourage exploration, may not always find the true globally optimal path but adapts better to uncertainties and dynamic environments. Requires heavy computation and careful tuning [11] [12]
Convolutional Neural Networks (CNNs)	High, time complexity depends on number of layers, filter and feature map size, input/output channels [13]	High (in known conditions) [13] [14]	Medium [13]	Limited (depends on model size, sensor robustness) [13] [14]	Visual feature extraction enables localisation and obstacle detection, vulnerable to environmental variation like lighting or occlusion. Needs significant training data [13] [14]
Ant Colony Optimisation (ACO)	Moderate-High, time complexity depends on number of iterations, nodes and ants [15] [16]	Near-optimal (not guaranteed) [15]	Medium [15]	Viable but computationally heavy [15]	Solution quality depends on the number of ants and pheromone tuning, no strict guarantee of absolute optimality but very good performance for complex, dynamic maps [15] [16]

Table 4: Table comparing implemented navigation techniques Braitenberg Vehicles, A, Dijkstra, Convolutional Neural Networks and Ant Colony Optimisation in terms of efficiency, optimality, heuristic dependence, ability to test different size AGVs and diagonal movement*

Method	**Efficiency**	**Optimality of path**	**Heuristic dependence**	**Different AGV sizes implementation**	**Diagonal movement**

Braitenberg Vehicles	Very Low (simple sensor-motor connection lead to limited and inefficient pathfinding)	No	No (reactive behaviour only)	No (real-world only)	No (reactive behaviour only)
A*	High (heuristic guidance reduces the search space)	Yes	Yes	Yes	Yes
Dijkstra's Algorithm	Moderate to High (exhaustive search with $O(n^2)$ time complexity	Yes	No (exhaustive search)	Yes	Yes
Convolutional Neural Networks (CNNs)	Low (highly computational expensive)	No (inconsistent paths)	No	No	No
Ant Colony Optimisation (ACO)	Low to Moderate (probabilistic approach with long convergence times to find near optimal solutions)	Converges toward optimal	Yes	No	Yes

APPENDIX III - PYTHON IMPLEMENTATION

All appendices who include code, were provided without docstring to reduce the overall length of the document. Each file can be found on GitHub with docstring.

```
## check if LEGO AGV is close to obstacles
## if obstacle detected within 10cm of sensor turn to the left
## otherwise drive straight forward
while True:

    ## current distance
    distance = distance_sensor.distance()

    ## drive forward
    motor_a.run(-500)
    motor_b.run(500)

    ## distance check to obstacles
    ## briefly drive back to avoid obstacle and then turn to the left
    if distance < 100:
        ## drive back
        motor_a.run(100)
        motor_b.run(-100)
        wait(200)

        ## turn to the left
        motor_a.run(160)
        motor_b.run(160)
        wait(400)

        print("LEGO AGV has detected an obstacle within its vicinity. Initiating
left turn")
```

Figure 74: Code for modified Braitenberg Vehicles 2a and 3b

```
def continue_movement():
    while True:

        ## go straight if white colour detected
        if colour_sensor.color() == (Color.WHITE or Color.NONE):
            motor_a.run(-200)
            motor_b.run(200)

        ## left turn if yellow colour detected
        elif colour_sensor.color() == Color.YELLOW:
            motor_a.run(100)
            motor_b.run(100)

        ## right turn if green colour detected
        elif colour_sensor.color() == Color.GREEN:
            motor_a.run(-100)
            motor_b.run(-100)

        ## stop motors
        else:
            motor_a.stop()
            motor_b.stop()
        wait(10)

        ## additional code to check if obstacle is present
        if distance_sensor.distance() < 45 and distance_sensor.distance() !=  -1:

            ## stop motors
            motor_a.stop()
            motor_b.stop()

            ## turn colour motor towards obstacle and print colour of detected
obstacle
            colour_motor.run_target(200, -100)
            wait(500)
            print(f"Obstacle detected is {colour_sensor.color()}")

            break


## reposition colour sensor to face down
colour_motor.run_target(200, 0)

## move along the green and yellow tape, until obstacle is detected
continue_movement()
```

Figure 75: Code for line following robot using singular colour sensor

```
        def get_iteration_offsets(radius, first_iteration):
            ## if radius is smaller than 3 or if it's the first iteration
            ## a thorough radius check needs to be conducted
```

```
            if radius < 3 or first_iteration:
                ## return list of all (i,j) within square radius
                return [
                    (i, j)
                    for i in range(-radius + 1, radius)
                    for j in range(-radius + 1, radius)
                ]
            else:
                ## if it's not the first iteration and radius is larger than 2,
offsets are defined
                offsets = []

                ## loop through the range of the radius to find the valid (i, j)
offsets
                for i in range(-radius + 1, radius):
                    for j in range(-radius + 1, radius):

                        ## calculate the Chebyshev distance, which is the max of
the absolute
                        ## x or y distance
                        dist = max(abs(i), abs(j))

                        ## only add offsets where the distance is radius - 1
                        ## this represents the outer layer of the radius square
                        if dist == radius - 1:
                            offsets.append((i, j))
                return offsets

        ## 4. add safety radius -> enables making the agv bigger
        if self.grid[target_y][target_x] == 0:
            ## check for obstacles in the vicinity
            for i, j in get_iteration_offsets(radius, first_iteration):
                check_y, check_x = target_y + i, target_x + j

                ## check if new location is outside if bounds
                if not (
                    0 <= check_y < len(self.grid) and 0 <= check_x <
len(self.grid[0])
                ):
                    return False

                ## if within bounds check for obstacle presence
                else:
                    if self.grid[check_y][check_x] == 1:  ## obstacle detected
                        return False

            return True  ## no obstacles found in vicinity
```

Figure 76: Code for checking if move is valid for different sized AGVs

```
class Navigation:
    ## define grid-based map (0 = free space, 1 = obstacle)
    map_grid = np.array(
        [
            [0, 0, 0, 0, 0],
            [1, 1, 0, 1, 0],
            [0, 0, 0, 0, 0],
            [0, 1, 0, 0, 0],
        ]
    )
CARDINAL_NEIGHBOURS = [(0, 1), (0, -1), (1, 0), (-1, 0)]  ## Up, Down, Left, Right
    DIAGONAL_NEIGHBOURS = [0, 1), (1, 0), (0, -1), (-1, 0),  ## Cardinal directions
        (1, 1), 1, -1), (-1, 1), (-1, -1), ]  ## Diagonal directions

   def __init__(self):

        self.grid = Navigation.map_grid
        self.exploration_counts = np.zeros_like(self.grid, dtype=int)
        self.non_optimal_paths = []
        self.block_locations = Database().block_locations
        self.goal = None

    @staticmethod
    def heuristic(node, goal):
        return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

    @staticmethod
    def heuristic_diagonal(node, goal):
        dx = abs(node[0] - goal[0])
        dy = abs(node[1] - goal[1])
        return max(dx, dy)

    def a_star_search(self, start, goal, agv_size, diagonal=False):
        self.goal = goal
        ## validate start and goal positions
        ## start position out of bounds
        if not (
            0 <= start[0] < self.grid.shape[0] and 0 <= start[1] <
self.grid.shape[1]
        ):
            raise ValueError(f"Start position {start} is out of map bounds.")

        ## goal position out of bounds
        if not (
            0 <= goal[0] < self.grid.shape[0] and 0 <= goal[1] < self.grid.shape[1]
        ):
            raise ValueError(f"Goal position {goal} is out of map bounds.")

        ## start position is on an obstacle
        if self.grid[start[0], start[1]] == 1:
            raise ValueError(f"Start position {start} is on an obstacle.")
```

```
        ## goal position is on an obstacle
        if self.grid[goal[0], goal[1]] == 1:
            raise ValueError(f"Goal position {goal} is on an obstacle.")

        ## start and goal position are within the same cell
        if self.goal == start:
            raise ValueError(
                f"Start position {start} and goal position {goal} are located
within the same cell."
            )

        ## run normal A* and return path if path was found
        path_result = self.run_a_star(start, goal, agv_size, diagonal)
        if path_result:
            return path_result

        ## retry with agv size of 1 to check if agv size was the issue
        fallback_result = self.run_a_star(start, goal, 1, diagonal)
        if fallback_result is not None:
            raise ValueError(
                f"The AGV size ({agv_size}) is too large for the current map
layout. Try reducing"
                    "it."
            )

        ## otherwise, no path even with agv size of 1
        raise ValueError(
            "No valid path found: A* could not reach the goal from the given
start."
        )

    def run_a_star(self, start, goal, agv_size, diagonal=False):
        self.goal = goal

        ## exploration count keeping track of which nodes are explored how often
        exploration_counts = np.zeros_like(self.grid, dtype=int)
        ## mark obstacles in the grid as -1 in the exploration counts
        exploration_counts[self.grid == 1] = -1

        ## priority queue (min-heap) to store open nodes sorted by f-score
        open_set = []
        heapq.heappush(open_set, (0, start, 0, 0))  ## (cost, node, dx, dy)

        ## hash set for quick membership check in open set
        open_set_hash = set()
        open_set_hash.add(start)

        ## keep track of visited nodes
        closed_set = set()
```

```
        ## dictionaries to store movement cost and estimated cost
        came_from = {}  ## tracks optimal path
        g_score = {start: 0}  ## actual movement cost from start node
        heuristic_func = (
            self.heuristic_diagonal if diagonal else self.heuristic
        )  ## chose heuristic
        f_score = {
            start: heuristic_func(start, self.goal)
        }  ## estimated total cost (g + h)

        ## flag to check for obstacles within agv radius more efficiently
        first_iteration = True

        ## loop until all nodes are explored, meaning path has been found or no
valid path exists
        while open_set:
            ## get node with the lowest f-score
            _, current, previous_dx, previous_dy = heapq.heappop(open_set)
            exploration_counts[current] += 1


            ## reconstruct path if goal has been reached
            if current == self.goal:
                print(
                    f"Total Cost of A Star path with Radius = {agv_size}:
{f_score[current]}"
                )

                return self.reconstruct_path(
                    came_from, current, start
                ), exploration_counts

            closed_set.add(current)  ## mark node as evaluated

            x, y = current
            for dx, dy in (
                self.DIAGONAL_NEIGHBOURS if diagonal else self.CARDINAL_NEIGHBOURS
            ):
                ## check that move is valid (some diagonal moves collide with
obstacles)
                if self.is_valid_move(agv_size, x, y, dx, dy, first_iteration):
                    neighbour = (x + dx, y + dy)
                    tentative_g_score = g_score[current] + (
                        np.sqrt(2) if dx != 0 and dy != 0 else 1
                    )

                    ## penalise turns to avoid unnecessary turns
                    if (previous_dx, previous_dy) != (dx, dy):
                        tentative_g_score += 0.0
```

```
                        # tentative_g_score += 0.5

                    ## if neighbour is already evaluated, skip
                    if neighbour in closed_set:
                        continue

                    ## if neighbour is not in open set or a shorter path is found,
update it
                    if (
                        neighbour not in g_score
                        or tentative_g_score < g_score[neighbour]
                    ):
                        came_from[neighbour] = current
                        g_score[neighbour] = tentative_g_score
                        f_score[neighbour] = tentative_g_score + heuristic_func(
                            neighbour, goal
                        )
                        heapq.heappush(
                            open_set, (f_score[neighbour], neighbour, dx, dy)
                        )

                        first_iteration = False

        return None

    def is_valid_move(self, radius, x, y, dx, dy, first_iteration):
        ## target positions of next move
        target_y, target_x = x + dx, y + dy

        ## precompute blocked positions (excluding goal)
        blocked_positions = {
            (by, bx)
            for colour, (by, bx) in self.block_locations.items()
            if (by, bx) != self.goal
        }

        ## 1. check that obstacle is not outside of bounds
        if not (0 <= target_y < len(self.grid) and 0 <= target_x <
len(self.grid[0])):
            return False

        ## 2. check for blocks in the way (unless it's the target)
        if (target_y, target_x) in blocked_positions:
            return False

        ## 3. additional check for diagonal moves - look at the "corner" between
current and target
        if dx != 0 and dy != 0:  ## if it is a diagonal move
            if (x + dx, y) in blocked_positions or (x, y + dy) in
blocked_positions:
```

```
                return False
            if self.grid[x + dx][y] == 1 or self.grid[x][y + dy] == 1:
                return False

        def get_iteration_offsets(radius, first_iteration):
            ## if radius is smaller than 3 or if it's the first iteration
            ## a thorough radius check needs to be conducted
            if radius < 3 or first_iteration:
                ## return list of all (i,j) within square radius
                return [
                    (i, j)
                    for i in range(-radius + 1, radius)
                    for j in range(-radius + 1, radius)
                ]
            else:
                ## if it's not the first iteration and radius is larger than 2,
offsets are defined
                offsets = []

                ## loop through the range of the radius to find the valid (i, j)
offsets
                for i in range(-radius + 1, radius):
                    for j in range(-radius + 1, radius):

                        ## calculate the Chebyshev distance, which is the max of
the absolute
                        ## x or y distance
                        dist = max(abs(i), abs(j))

                        ## only add offsets where the distance is radius - 1
                        ## this represents the outer layer of the radius square
                        if dist == radius - 1:
                            offsets.append((i, j))
                return offsets

        ## 4. add safety radius -> enables making the agv bigger
        if self.grid[target_y][target_x] == 0:
            ## check for obstacles in the vicinity
            for i, j in get_iteration_offsets(radius, first iteration):
                check_y, check_x = target_y + i, target_x + j

                ## check if new location is outside if bounds
                if not (
                    0 <= check_y < len(self.grid) and 0 <= check_x <
len(self.grid[0])
                ):
                    return False

                ## if within bounds check for obstacle presence
                else:
```

```
                    if self.grid[check_y][check_x] == 1:  ## obstacle detected
                        return False

            return True  ## no obstacles found in vicinity

    @staticmethod
    def reconstruct_path(came_from, current, start):
        path = []

        ## traverse the 'came_from' directory to build the path from goal to start
        while current in came_from:
            path.append(current)
            current = came_from[current]

        ## reverse path to get it from start to goal
        path.reverse()

        ## ensure start node is included in path
        if path[0] != start:
            path.insert(0, start)

        return path
```

Figure 77: Code snippet for A implementation*

```
class Navigation:
    ## define grid-based map (0 = free space, 1 = obstacle)
    map_grid = np.array(
        [
            [0, 0, 0, 0, 0],
            [1, 1, 0, 1, 0],
            [0, 0, 0, 0, 0],
            [0, 1, 0, 0, 0],
        ]
    )
    CARDINAL_NEIGHBOURS = [(0, 1), (0, -1), (1, 0), (-1, 0)]  ## Up, Down, Left,
Right
    DIAGONAL_NEIGHBOURS = [0, 1), (1, 0), (0, -1), (-1, 0),  ## Cardinal directions
        (1, 1), 1, -1), (-1, 1), (-1, -1), ]  ## Diagonal directions

    def __init__(self):
        self.grid = Navigation.map_grid
        self.exploration_counts = np.zeros_like(self.grid, dtype=int)
        self.non_optimal_paths = []
        self.block_locations = Database().block_locations
        self.goal = None

    def dijkstra_search(self, start, goal, agv_size, diagonal=False):
```

```
        self.goal = goal
        ## validate start and goal positions
        ## start position out of bounds
        if not (
            0 <= start[0] < self.grid.shape[0] and 0 <= start[1] <
self.grid.shape[1]
        ):
            raise ValueError(f"Start position {start} is out of map bounds.")

        ## goal position out of bounds
        if not (
            0 <= goal[0] < self.grid.shape[0] and 0 <= goal[1] < self.grid.shape[1]
        ):
            raise ValueError(f"Goal position {goal} is out of map bounds.")

        ## start position is on an obstacle
        if self.grid[start[0], start[1]] == 1:
            raise ValueError(f"Start position {start} is on an obstacle.")

        ## goal position is on an obstacle
        if self.grid[goal[0], goal[1]] == 1:
            raise ValueError(f"Goal position {goal} is on an obstacle.")

        ## start and goal position are within the same cell
        if self.goal == start:
            raise ValueError(
                f"Start position {start} and goal position {goal} are located
within the same cell."
            )

        ## run normal Dijkstra and return path if path was found
        path_result = self.run_dijkstra(start, agv_size, diagonal)
        if path_result:
            return path_result

        ## retry with agv size of 1 to check if agv size was the issue
        fallback_result = self.run_dijkstra(start, 1, diagonal)
        if fallback_result is not None:
            raise ValueError(
                f"The AGV size ({agv_size}) is too large for the current map
layout. Try reducing"
                    "it."
            )

        ## otherwise, no path even with agv size of 1
        raise ValueError(
            "No valid path found: Dijkstra could not reach the goal from the given
start."
        )
```

```
    def run_dijkstra(self, start, agv_size, diagonal):
        ## exploration count keeping track of which nodes are explored how often
        exploration_counts = np.zeros_like(self.grid, dtype=int)
        ## mark obstacles in the grid as -1 in the exploration counts
        exploration_counts[self.grid == 1] = -1

        ## convert numpy arrays to tuples if needed
        start = tuple(start) if isinstance(start, np.ndarray) else start
        self.goal = tuple(self.goal) if isinstance(self.goal, np.ndarray) else
self.goal

        ## priority queue: (cumulative_cost, y, x)
        heap = [(0, start[0], start[1])]

        ## visited dictionary: {node: (cost, previous_node)}
        visited = {start: (0, None)}

        ## movement directions
        directions = self.DIAGONAL_NEIGHBOURS if diagonal else
self.CARDINAL_NEIGHBOURS

        ## keep track of visited nodes
        closed_set = set()

        while heap:
            current_cost, y, x = heapq.heappop(heap)
            current_node = (y, x)

            ## ensure no nodes are processed multiple times
            if current_node in closed_set:
                continue
            closed_set.add(current_node)

            exploration_counts[y,x] += 1

            ## early exit if goal is reached
            if current_node == self.goal:
                break


            ## explore neighbours
            for dy, dx in directions:
                ny, nx = y + dy, x + dx
                neighbour = (ny, nx)

                # #check if move is valid (bounds, obstacles, AGV size)
                if self.is_valid_move(agv_size, y, x, dy, dx,
first_iteration=False):

                    ## calculate move cost (sqrt(2) for diagonal, 1 for cardinal)
```

```
                    move_cost = math.sqrt(dy**2 + dx**2)
                    new_cost = current_cost + move_cost

                    ## if neighbour not visited or found cheaper path
                    if neighbour not in visited or new_cost <
visited[neighbour][0]:
                        visited[neighbour] = (new_cost, current_node)
                        heapq.heappush(heap, (new_cost, ny, nx))

        ## reconstruct path if goal was reached
        if self.goal in visited:
            path = self.reconstruct_path(visited, self.goal, start)

            total_cost = visited[self.goal][0]
            print(f"Total Cost of Dijkstra path with Radius = {agv_size}:
{total_cost}")
            return path, exploration_counts

        return None

    def is_valid_move(self, radius, x, y, dx, dy, first_iteration):
        ## target positions of next move
        target_y, target_x = x + dx, y + dy

        ## precompute blocked positions (excluding goal)
        blocked_positions = {
            (by, bx)
            for colour, (by, bx) in self.block_locations.items()
            if (by, bx) != self.goal
        }

        ## 1. check that obstacle is not outside of bounds
        if not (0 <= target_y < len(self.grid) and 0 <= target_x <
len(self.grid[0])):
            return False

        ## 2. check for blocks in the way (unless it's the target)
        if (target_y, target_x) in blocked_positions:
            return False

        ## 3. additional check for diagonal moves - look at the "corner" between
current and target
        if dx != 0 and dy != 0:  ## if it is a diagonal move
            if (x + dx, y) in blocked_positions or (x, y + dy) in
blocked_positions:
                return False
            if self.grid[x + dx][y] == 1 or self.grid[x][y + dy] == 1:
                return False

        def get_iteration_offsets(radius, first_iteration):
```

```
            ## if radius is smaller than 3 or if it's the first iteration
            ## a thorough radius check needs to be conducted
            if radius < 3 or first_iteration:
                ## return list of all (i,j) within square radius
                return [
                    (i, j)
                    for i in range(-radius + 1, radius)
                    for j in range(-radius + 1, radius)
                ]
            else:
                ## if it's not the first iteration and radius is larger than 2,
offsets are defined
                offsets = []

                ## loop through the range of the radius to find the valid (i, j)
offsets
                for i in range(-radius + 1, radius):
                    for j in range(-radius + 1, radius):

                        ## calculate the Chebyshev distance, which is the max of
the absolute
                        ## x or y distance
                        dist = max(abs(i), abs(j))

                        ## only add offsets where the distance is radius - 1
                        ## this represents the outer layer of the radius square
                        if dist == radius - 1:
                            offsets.append((i, j))
                return offsets

        ## 4. add safety radius -> enables making the agv bigger
        if self.grid[target_y][target_x] == 0:
            ## check for obstacles in the vicinity
            for i, j in get_iteration_offsets(radius, first_iteration):
                check_y, check_x = target_y + i, target_x + j

                ## check if new location is outside if bounds
                if not (
                    0 <= check_y < len(self.grid) and 0 <= check_x <
len(self.grid[0])
                ):
                    return False

                ## if within bounds check for obstacle presence
                else:
                    if self.grid[check_y][check_x] == 1:  ## obstacle detected
                        return False

            return True  ## no obstacles found in vicinity
```

```
    @staticmethod
    def reconstruct_path(visited, goal, start):
        path = []
        current = goal

        while current != start:
            path.append(current)
            ## get predecessor
            current = visited[current][1]

        path.append(start)
        path.reverse()
        return path
```

Figure 78: Code snippet for Dijkstra implementation

```
    model = models.Sequential()

    ## convolutional layers with increasing filters
    model.add(
        layers.Conv2D(
            32, (3, 3), activation="relu", input_shape=input_shape, padding="same"
        )
    )
    model.add(layers.Conv2D(64, (3, 3), activation="relu", padding="same"))
    model.add(layers.Conv2D(128, (3, 3), activation="relu", padding="same"))
    model.add(layers.Conv2D(256, (3, 3), activation="relu", padding="same"))
    model.add(layers.Conv2D(512, (3, 3), activation="relu", padding="same"))

    ## flatten and dense layers
    model.add(layers.Flatten())
    model.add(layers.Dense(512, activation="relu"))
    model.add(layers.Dense(output_units, activation="sigmoid"))

    ## compile model
    model.compile(optimizer="adam", loss="binary_crossentropy",
metrics=["binary_accuracy"])

    ## callbacks for better training
    reduce_lr = ReduceLROnPlateau(
        monitor="val_loss", factor=0.5, patience=3, min_lr=1e-4
    )
    early_stop = EarlyStopping(
        monitor="val_loss", patience=5, restore_best_weights=True
    )

    ## train model
    history = model.fit(
        X_train,
        y_train,
```

```
        epochs=30,
        batch_size=32,
        validation_split=0.2,
        callbacks=[early_stop, reduce_lr],
```

Figure 79: Code snippet for CNN implementation

```
class Navigation:
    ## define grid-based map (0 = free space, 1 = obstacle)
    map_grid = np.array(
        [
            [0, 0, 0, 0, 0],
            [1, 1, 0, 1, 0],
            [0, 0, 0, 0, 0],
            [0, 1, 0, 0, 0],
        ]
    )
    CARDINAL_NEIGHBOURS = [(0, 1), (0, -1), (1, 0), (-1, 0)]  ## Up, Down, Left,
Right
    DIAGONAL_NEIGHBOURS = [0, 1), (1, 0), (0, -1), (-1, 0),  ## Cardinal directions
        (1, 1), 1, -1), (-1, 1), (-1, -1), ]  ## Diagonal directions

    def __init__(self):
        self.grid = Navigation.map_grid
        self.block_locations = Database().block_locations
        self.goal = None   ## will be set during search
        self.exploration_counts = np.zeros_like(self.grid, dtype=int)
        self.non_optimal_paths = []

        # ACO parameters
        self.num_ants = 100
        self.max_iterations = 20
        self.alpha = 1.0  # Pheromone importance
        self.beta = 1.0   # Heuristic importance
        self.rho = 0.1    # Evaporation rate
        self.delta_t = 2.0      # Pheromone deposit constant
        self.pheromone = np.ones_like(self.grid, dtype=float) * 0.1
        self.pheromone[self.grid == 1] = 0  # No pheromone on obstacles

    @staticmethod
    def heuristic(node, goal):
        return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

    @staticmethod
    def heuristic_diagonal(node, goal):
        dx = abs(node[0] - goal[0])
        dy = abs(node[1] - goal[1])
        return max(dx, dy)
```

```
    def aco_search(self, start, goal, diagonal=False):
        self.goal = goal
        ## validate start and goal positions
        ## start position out of bounds
        if not (
            0 <= start[0] < self.grid.shape[0] and 0 <= start[1] <
self.grid.shape[1]
        ):
            raise ValueError(f"Start position {start} is out of map bounds.")

        ## goal position out of bounds
        if not (
            0 <= goal[0] < self.grid.shape[0] and 0 <= goal[1] < self.grid.shape[1]
        ):
            raise ValueError(f"Goal position {goal} is out of map bounds.")

        ## start position is on an obstacle
        if self.grid[start[0], start[1]] == 1:
            raise ValueError(f"Start position {start} is on an obstacle.")

        ## goal position is on an obstacle
        if self.grid[goal[0], goal[1]] == 1:
            raise ValueError(f"Goal position {goal} is on an obstacle.")

        ## start and goal position are within the same cell
        if self.goal == start:
            raise ValueError(
                f"Start position {start} and goal position {goal} are located
within the same cell."
            )

        ## run ACO
        path_result = self.run_aco(start, diagonal)
        if path_result:
            return path_result

        ## if no path is found
        raise ValueError("No valid path found: ACO could not reach the goal.")

    def run_aco(self, start, diagonal):
        best_path = None
        best_path_length = float('inf')
        best_path_cost = float('inf')

        for iteration in range(self.max_iterations):
            all_paths = []

            for ant in range(self.num_ants):
                path, path_cost = self.construct_ant_path(start, diagonal)
                if path:
```

```
                    path_length = len(path)
                    all_paths.append((path, path_length))

                    if path_length < best_path_length:
                        best_path = path
                        best_path_length = path_length
                        best_path_cost = path_cost

            self.update_pheromone(all_paths)
            self.evaporate_pheromone()

            if best_path and iteration > 200 and len(set([p[1] for p in all_paths[-
50:]])) == 1:
                break

        if best_path:
            print(f"Total cost of ACO path: {best_path_cost}")
            return best_path, self.create_exploration_heatmap_aco()

        return None

    def construct_ant_path(self, start, diagonal):
        path = [start]
        current = start
        visited = set([start])
        total_cost = 0.0

        while current != self.goal:
            neighbours = self.get_valid_neighbours(current, diagonal, visited)
            if not neighbours:
                return None, float('inf')   ## dead end

            next_node = self.select_next_node(current, neighbours, diagonal)
            if next_node is None:
                return None, float('inf')

            ## determine cost for each move
            dx = next_node[0] - current[0]
            dy = next_node[1] - current[1]
            move_cost = np.sqrt(2) if dx != 0 and dy != 0 else 1

            total_cost += move_cost
            path.append(next_node)
            visited.add(next_node)
            current = next_node

        return path, total_cost

    def get_valid_neighbours(self, node, diagonal, visited):
        neighbours = []
```

```
        x, y = node

        ## precompute blocked positions (excluding goal)
        blocked_positions = {
            (by, bx) for colour, (by, bx) in self.block_locations.items()
            if (by, bx) != self.goal
        }

        for dx, dy in (self.DIAGONAL_NEIGHBOURS if diagonal else
self.CARDINAL_NEIGHBOURS):
            nx, ny = x + dx, y + dy

            ## check bounds
            if not (0 <= nx < len(self.grid) and 0 <= ny < len(self.grid[0])):
                continue

            ## check obstacle
            if self.grid[nx][ny] == 1:
                continue

            ## check if already visited
            if (nx, ny) in visited:
                continue

            ## check for LEGO blocks in the way (unless it's the target)
            if (nx, ny) in blocked_positions:
                continue

            ## additional checks for diagonal moves
            if dx != 0 and dy != 0:
                ## check if diagonal path is blocked by obstacles
                if self.grid[x + dx][y] == 1 or self.grid[x][y + dy] == 1:
                    continue

                ## check if diagonal path is blocked by LEGO blocks
                if (nx, y) in blocked_positions or (x, ny) in blocked_positions:
                    continue

            neighbours.append((nx, ny))

        return neighbours

    def select_next_node(self, current, neighbours, diagonal):
        if not neighbours:
            return None

        probabilities = []
        total = 0.0

        ## choose heuristic based on diagonal movement setting
```

```
        heuristic_func = self.heuristic_diagonal if diagonal else self.heuristic

        for neighbour in neighbours:
            pheromone = self.pheromone[neighbour[0], neighbour[1]]
            heuristic = 1 / (1 + heuristic_func(neighbour, self.goal))

            probability = (pheromone ** self.alpha) * (heuristic ** self.beta)
            probabilities.append(probability)
            total += probability

        if total == 0:
            return neighbours[np.random.randint(0, len(neighbours))]

        probabilities = [p / total for p in probabilities]
        return neighbours[np.random.choice(range(len(neighbours)),
p=probabilities)]

    def update_pheromone(self, paths):
        for path, cost in paths:
            if not path or cost == 0 or cost == float('inf'):
                continue

            pheromone_deposit = self.delta_t / cost

            for node in path:
                self.pheromone[node[0], node[1]] += pheromone_deposit

    def evaporate_pheromone(self):
        self.pheromone *= (1 - self.rho)
        self.pheromone[self.pheromone < 0.1] = 0.1  # Minimum pheromone level

    def create_exploration_heatmap_aco(self):
        heatmap = self.pheromone.copy()
        heatmap[self.grid == 1] = -1
        return heatmap
```

Figure 80: Code snippet for ACO implementation

APPENDIX IV - CNN HEATMAPS OF GENERATED MAZES

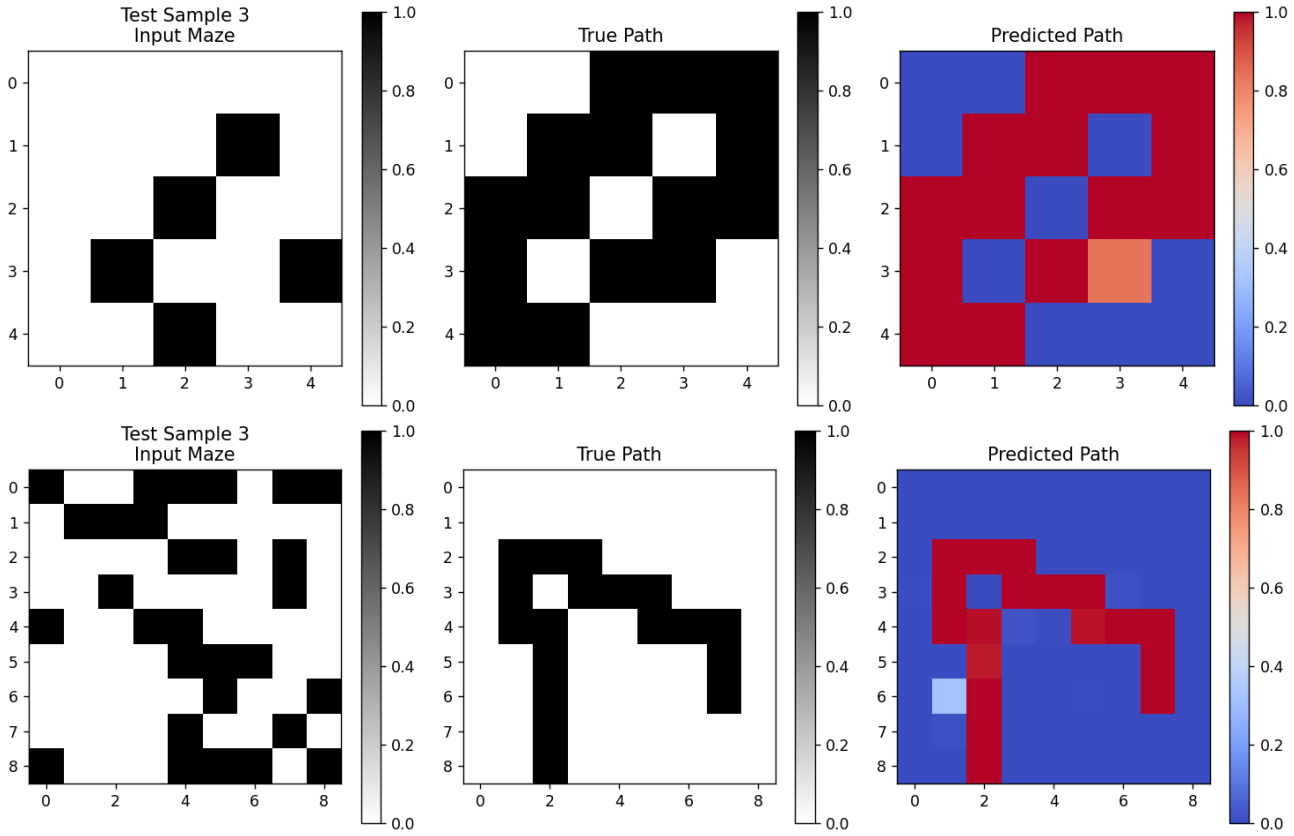
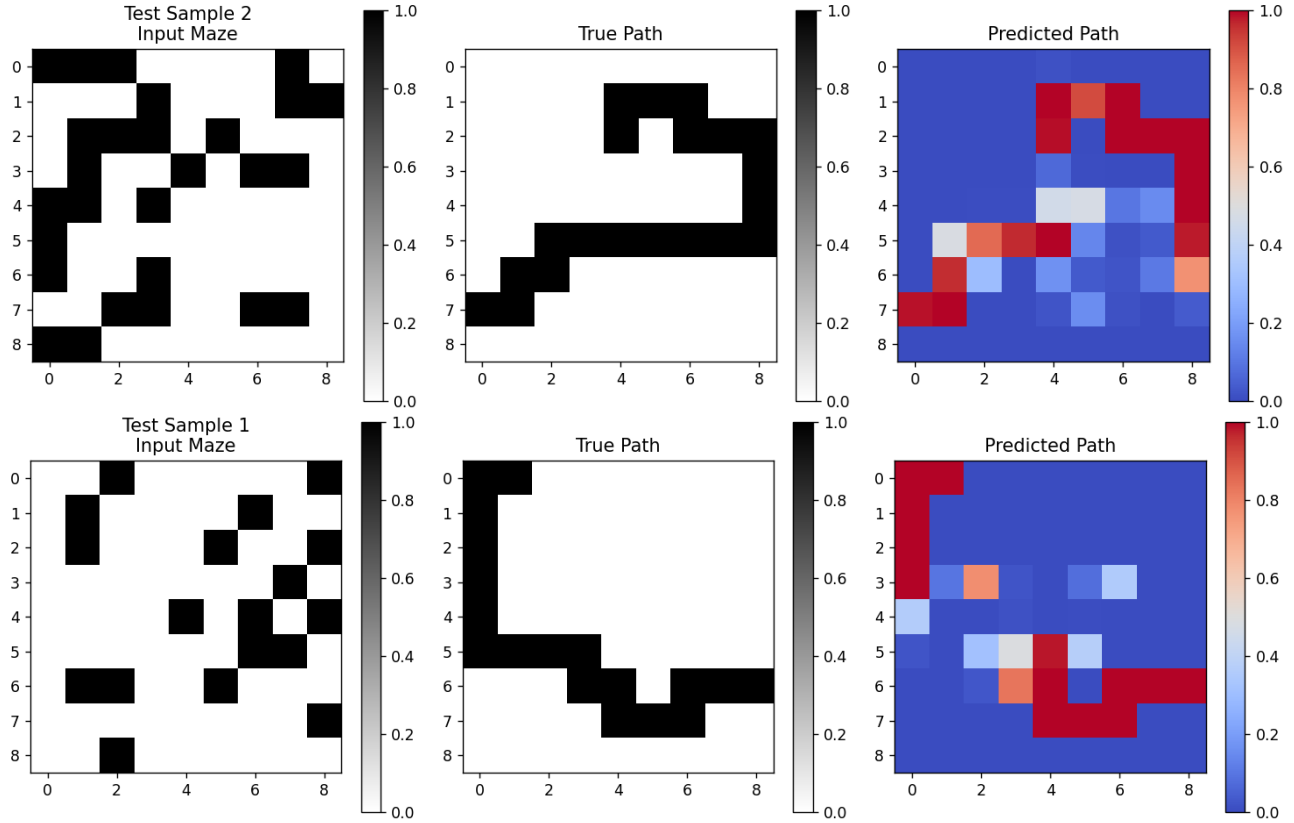


Figure 81: Examples of correctly solved grids by the CNN model generated within the script



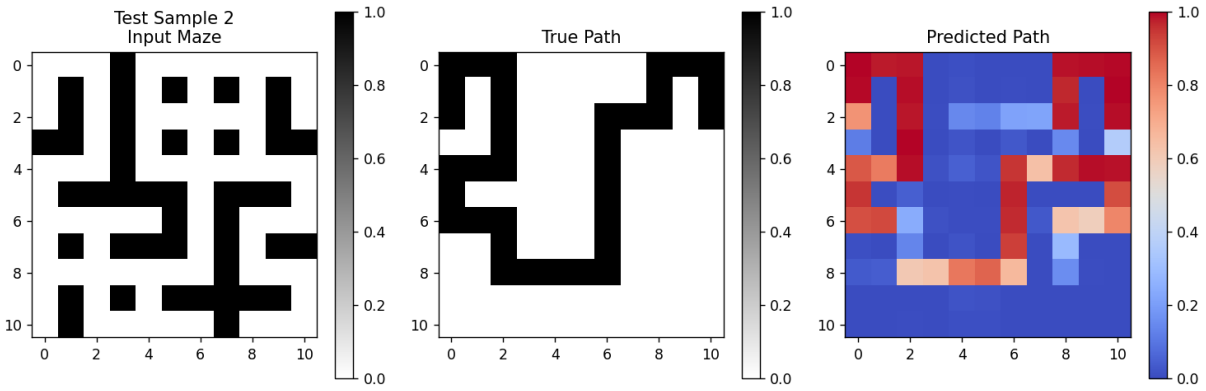


Figure 82: Examples of test grids where the CNN model failed to generalize, resulting in incorrect or incomplete path predictions.

APPENDIX V - STRUCTURED GRIDS

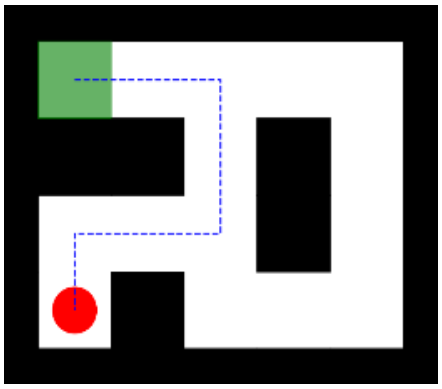


Figure 83: Grid #1: 4x5 sized grid showing the optimal path from start (green square) to end (red circle)

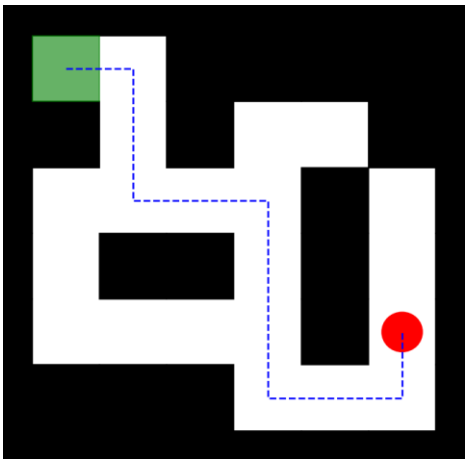


Figure 84: Grid #2: 6x6 sized grid showing the optimal path from start (green square) to end (red circle)

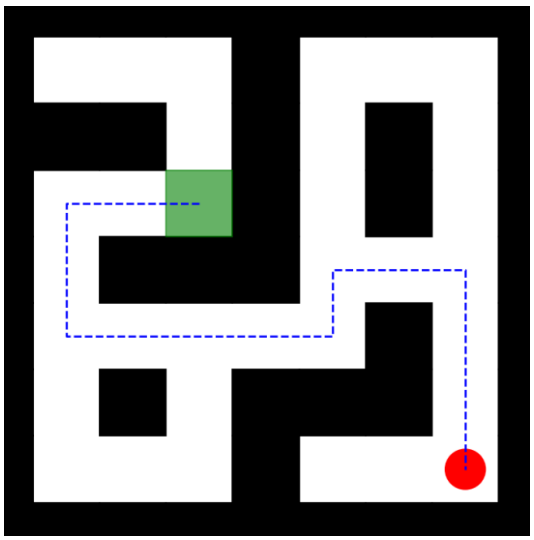


Figure 85: Grid #3: 7x7 sized grid showing the optimal path from start (green square) to end (red circle)

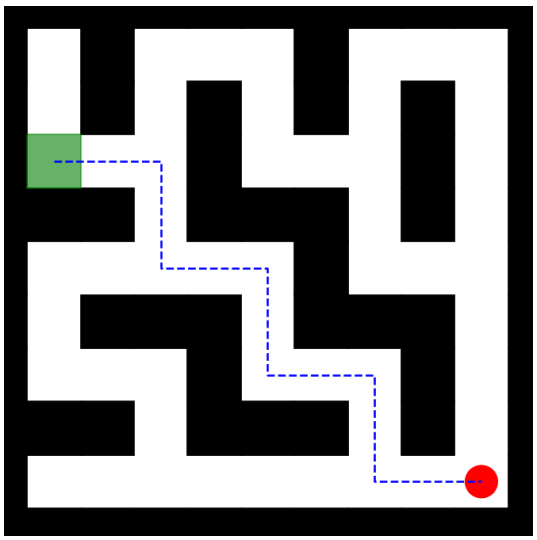


Figure 86: Grid #4: 9x9 sized grid showing the optimal path from start (green square) to end (red circle)

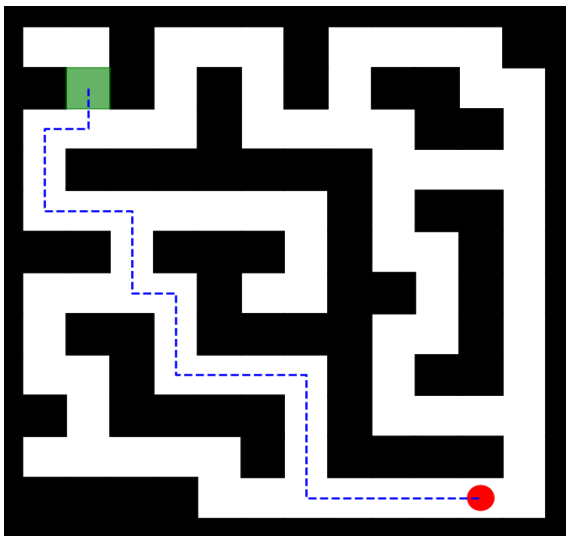


Figure 87: Grid #5: 12x12 sized grid showing the optimal path from start (green square) to end (red circle)

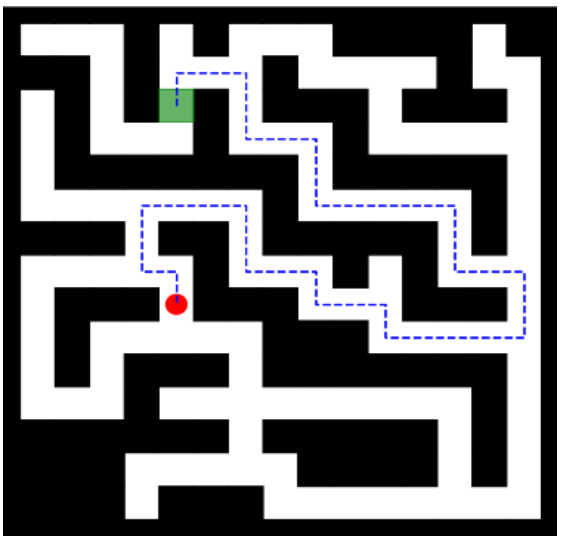


Figure 88: Grid #6: 15x15 sized grid showing the optimal path from start (green square) to end (red circle)

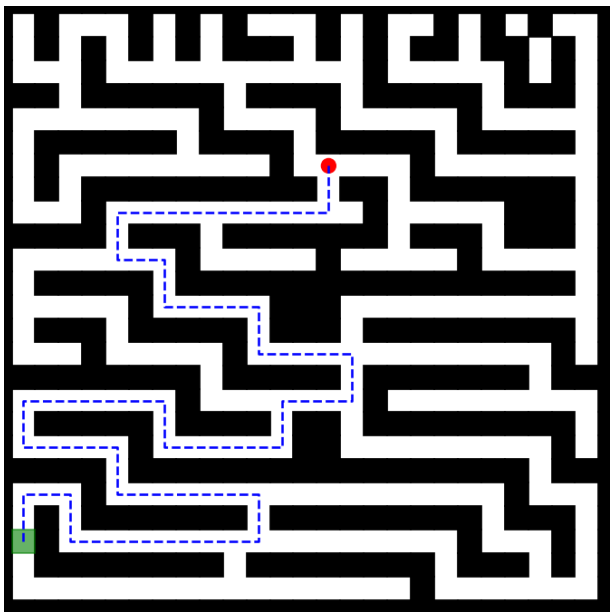


Figure 89: Grid #8: 25x25 sized grid showing the optimal path from start (green square) to end (red circle)

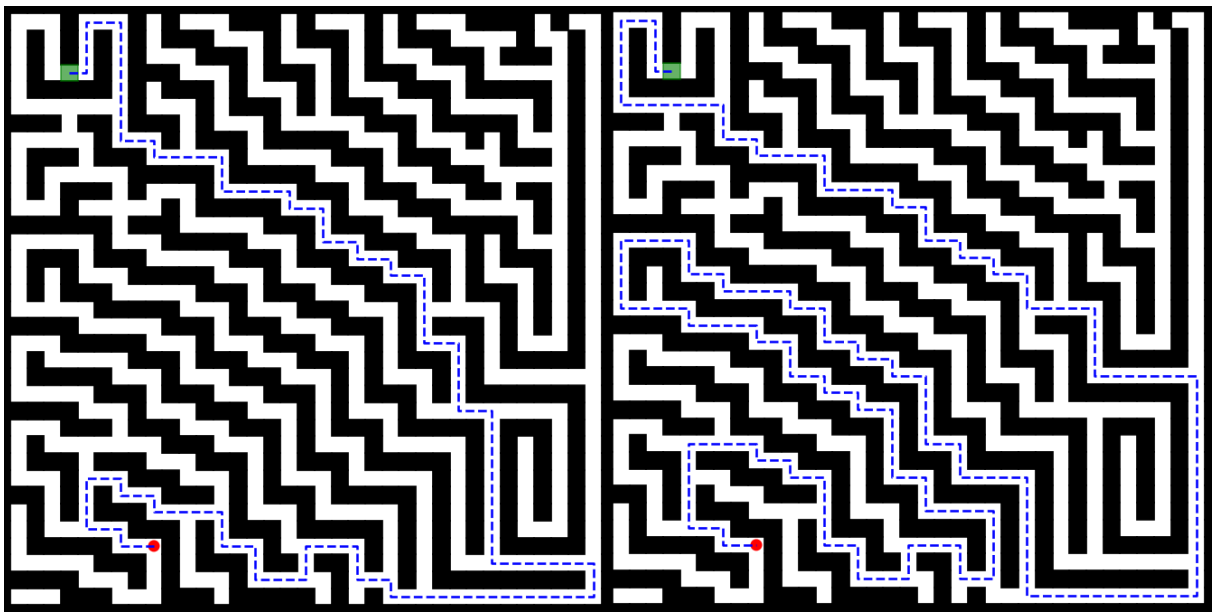


Figure 90: Grid #9: 35x35 sized grid showing the optimal path on the left side and ACO path on the right side from start (green square) to end (red circle)

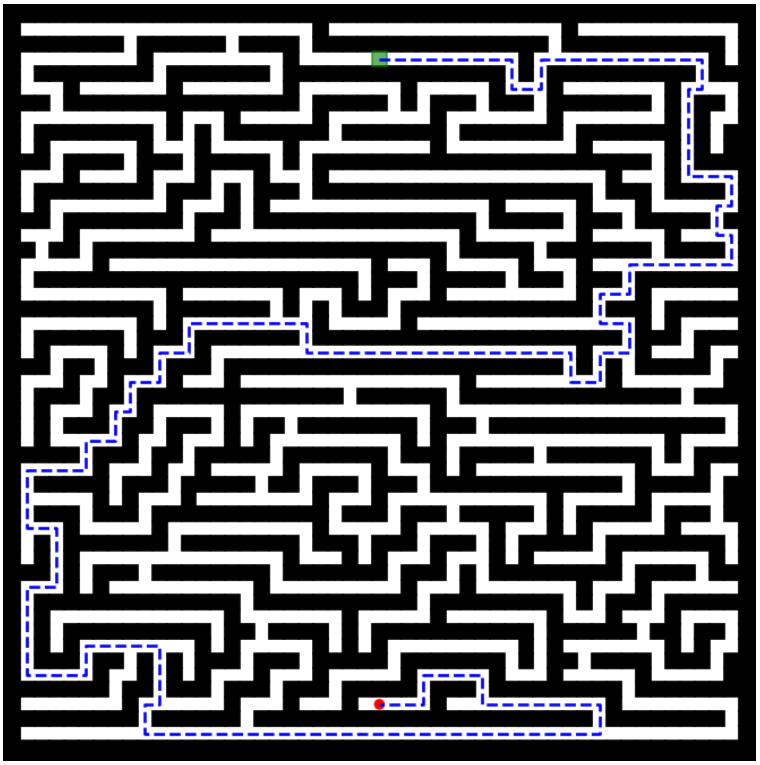


Figure 91: Grid #10: 50x50 sized grid showing the optimal path from start (green square) to end (red circle)

APPENDIX VI - STRUCTURED DOUBLE CELL PATH GRIDS

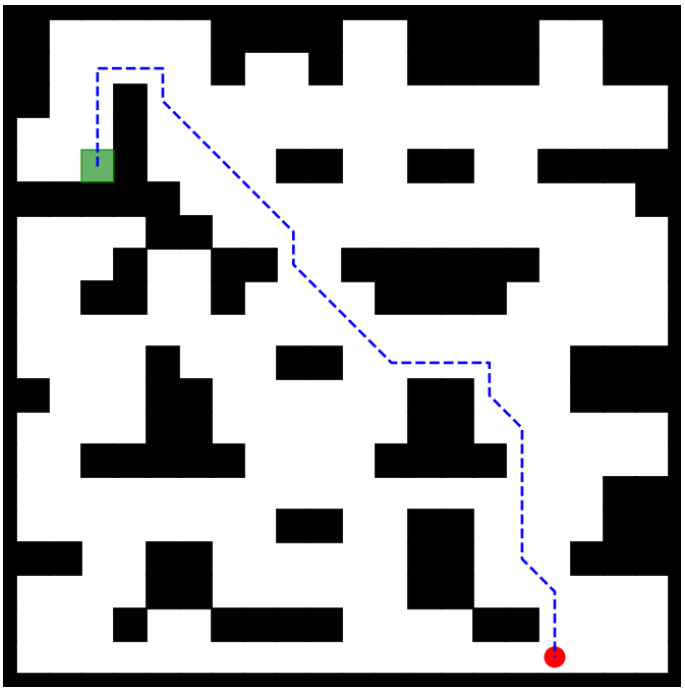


Figure 92: Grid #12: 20x20 sized grid showing the optimal path from start (green square) to end (red circle)

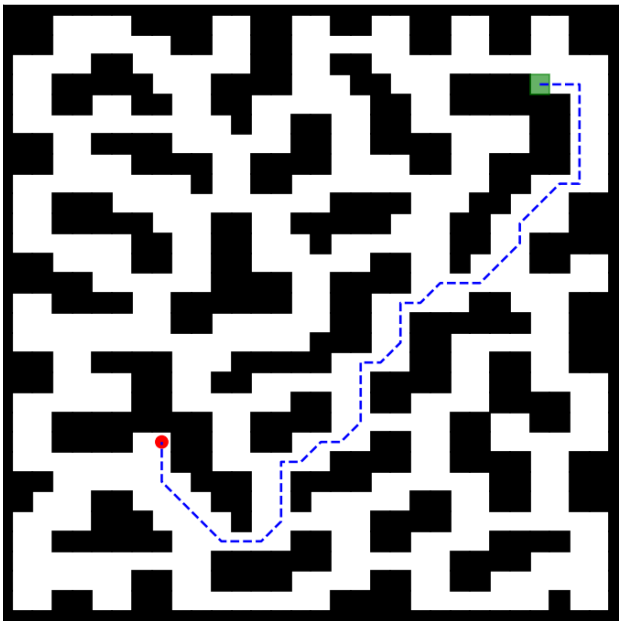


Figure 93: Grid #13: 30x30 sized grid showing the optimal path from start (green square) to end (red circle)

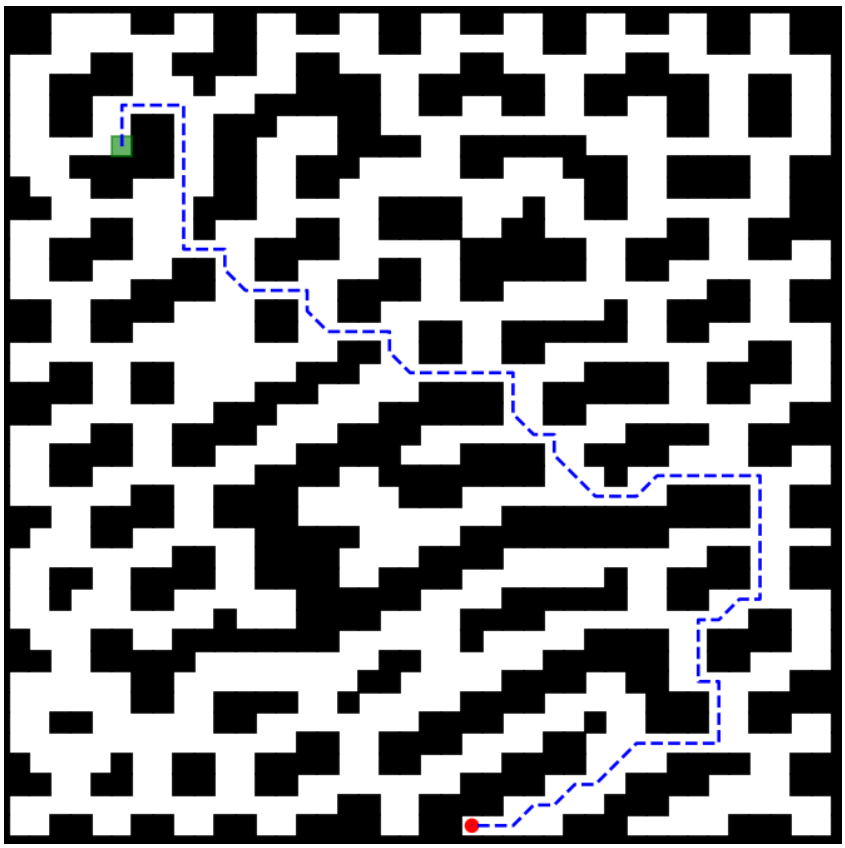


Figure 94: Grid #14: 40x40 sized grid showing the optimal path from start (green square) to end (red circle)

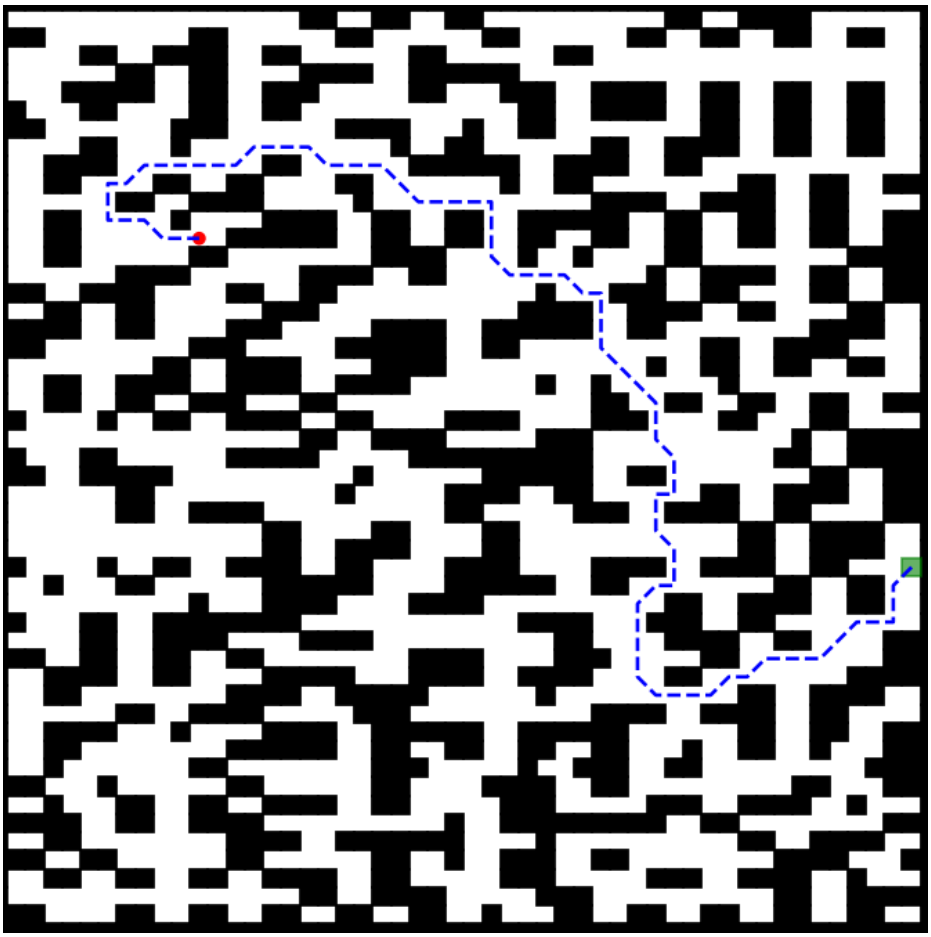


Figure 95: Grid #15: 50x50 sized grid showing the optimal path from start (green square) to end (red circle)

APPENDIX VII - UNSTRUCTURED GRIDS AND CORRESPONDING HEATMAPS

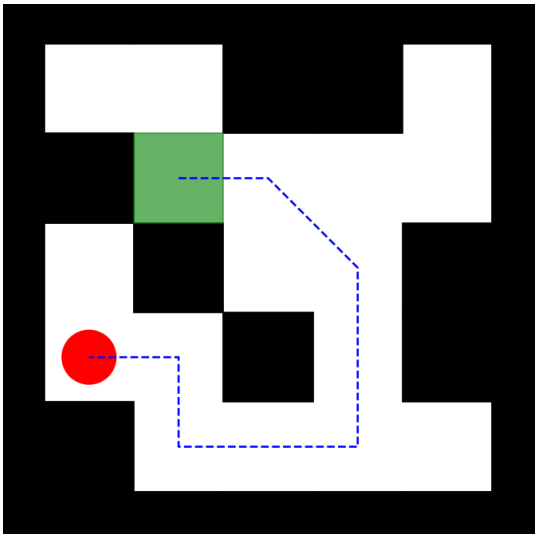


Figure 96: Grid #16: 5x5 sized grid showing the optimal path from start (green square) to end (red circle)

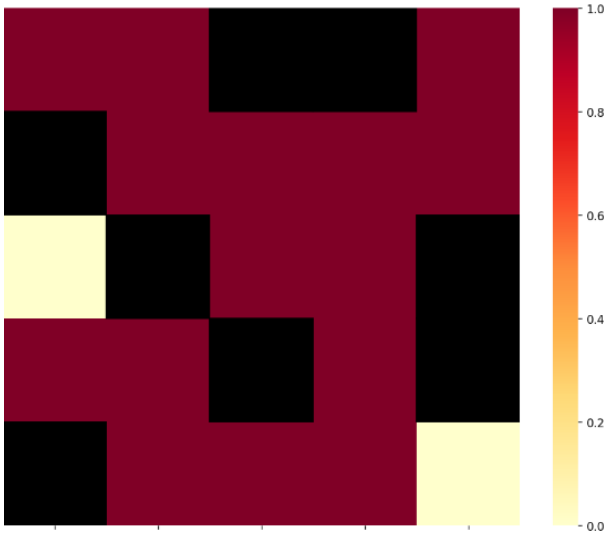


Figure 97: Grid #16: Exploration heatmap of A for a 5x5 sized grid*

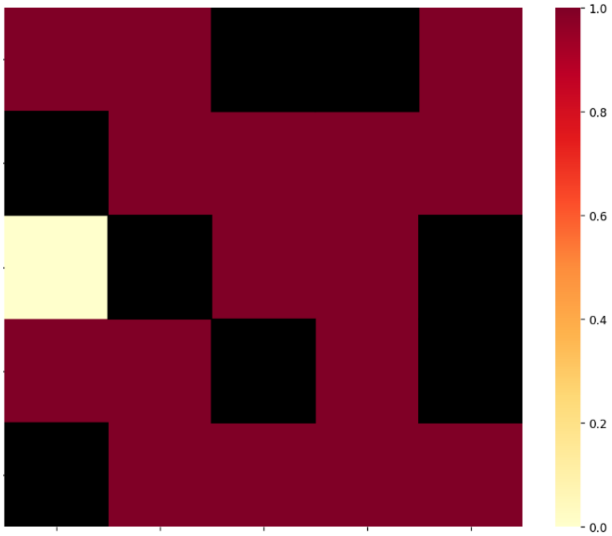


Figure 98: Grid #16: Exploration heatmap of Dijkstra for a 5x5 sized grid

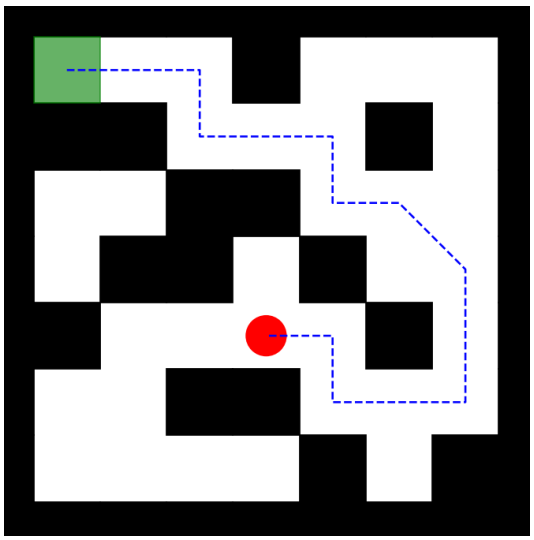


Figure 99: Grid #17: 7x7 sized grid showing the optimal path from start (green square) to end (red circle)

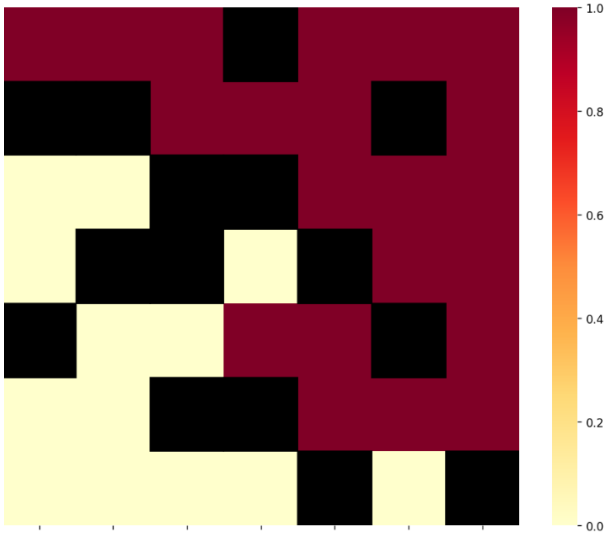


Figure 100: Grid #17: Exploration heatmap of A for a 7x7 sized grid*

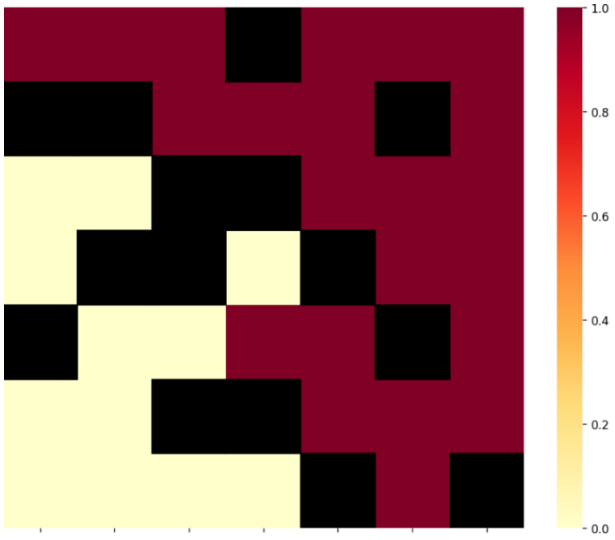


Figure 101: Grid #17: Exploration heatmap of Dijkstra for a 7x7 sized grid

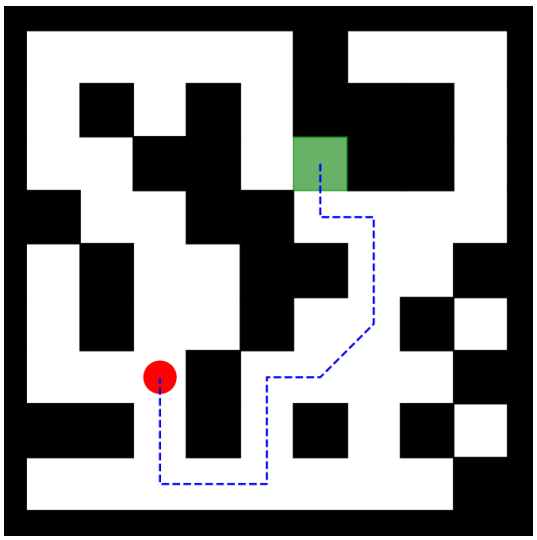


Figure 102: Grid #18: 9x9 sized grid showing the optimal path from start (green square) to end (red circle)

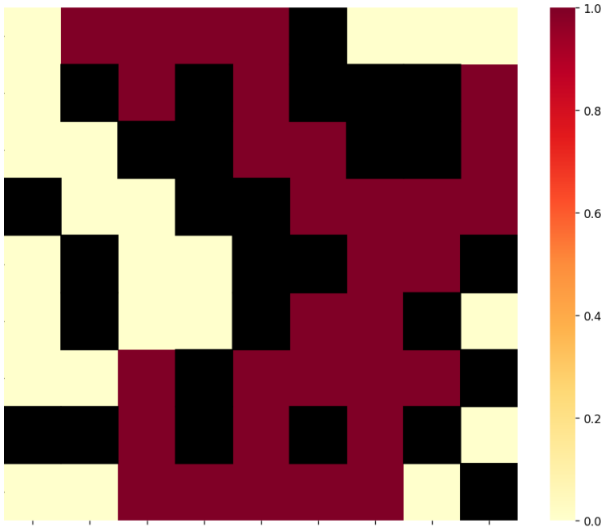


Figure 103: Grid #18: Exploration heatmap of A for a 9x9 sized grid*

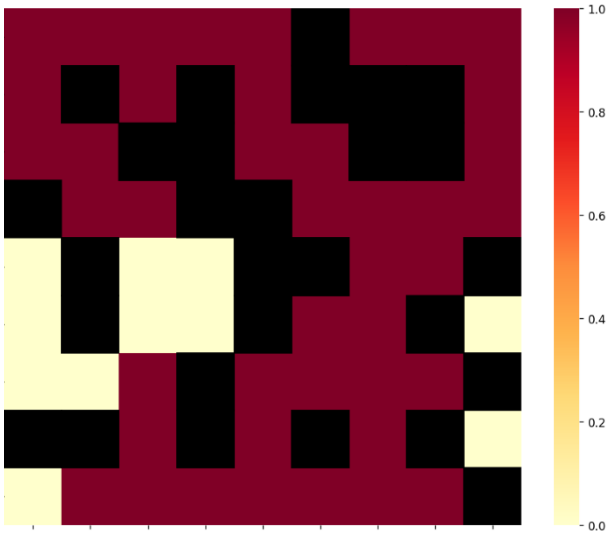


Figure 104: Grid #18: Exploration heatmap of Dijkstra for a 9x9 sized grid

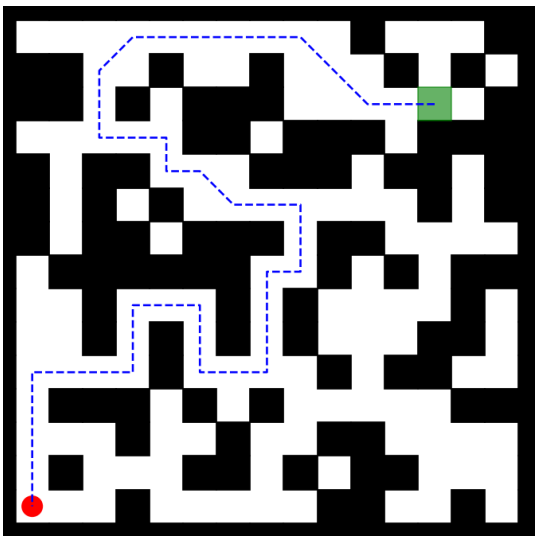


Figure 105: Grid #20: 15x15 sized grid showing the optimal path from start (green square) to end (red circle)

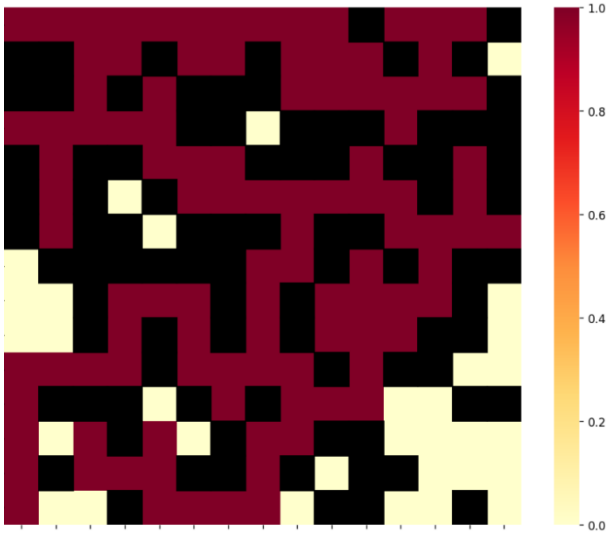


Figure 106: Grid #20: Exploration heatmap of A for a 15x15 sized grid*

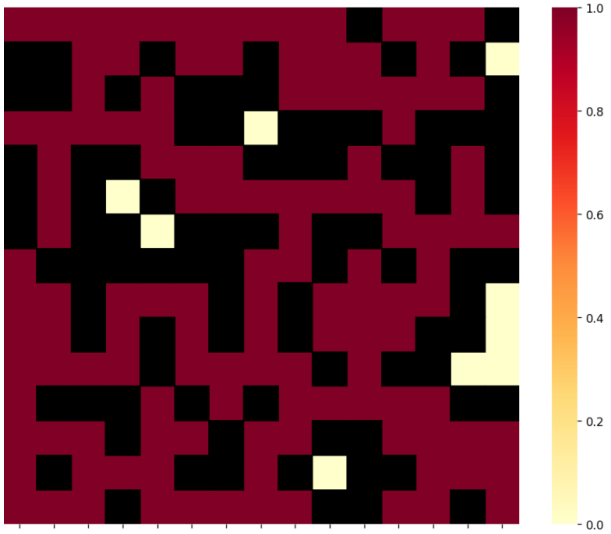


Figure 107: Grid #20: Exploration heatmap of Dijkstra for a 15x15 sized grid

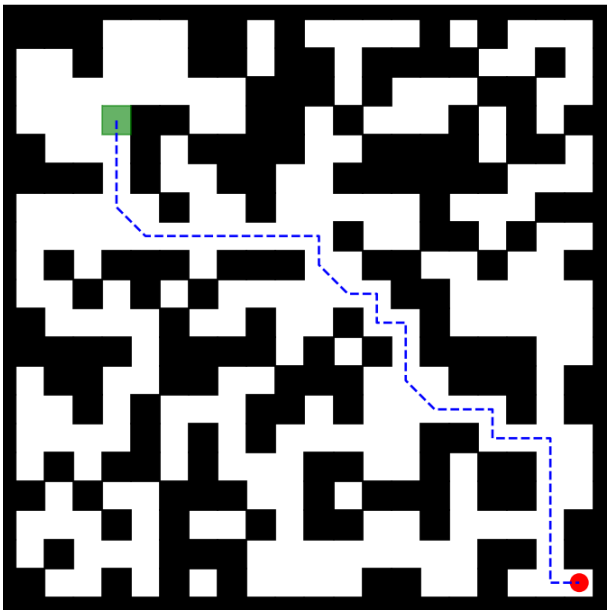


Figure 108: Grid #21: 20x20 sized grid showing the optimal path from start (green square) to end (red circle)

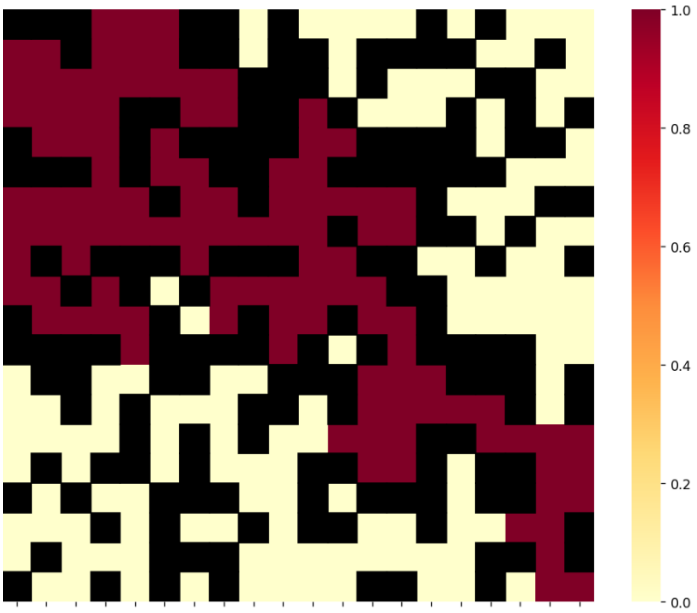


Figure 109: Grid #21: Exploration heatmap of A for a 20x20 sized grid*

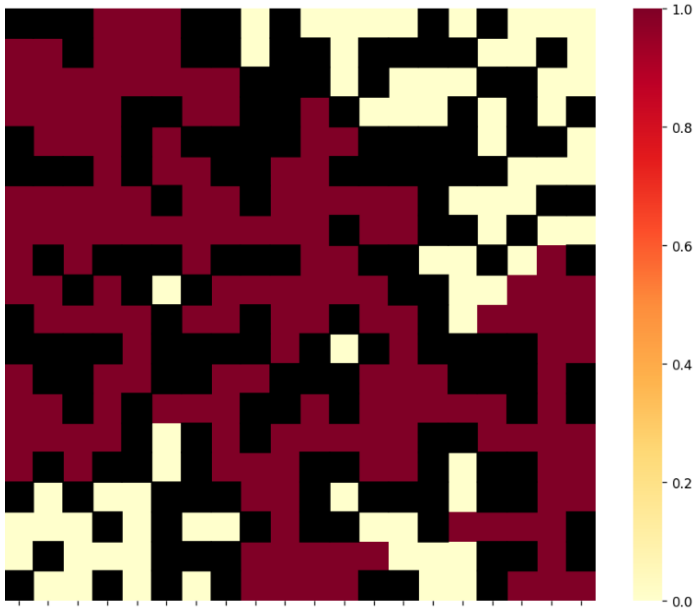


Figure 110: Grid #21: Exploration heatmap of Dijkstra for a 20x20 sized grid

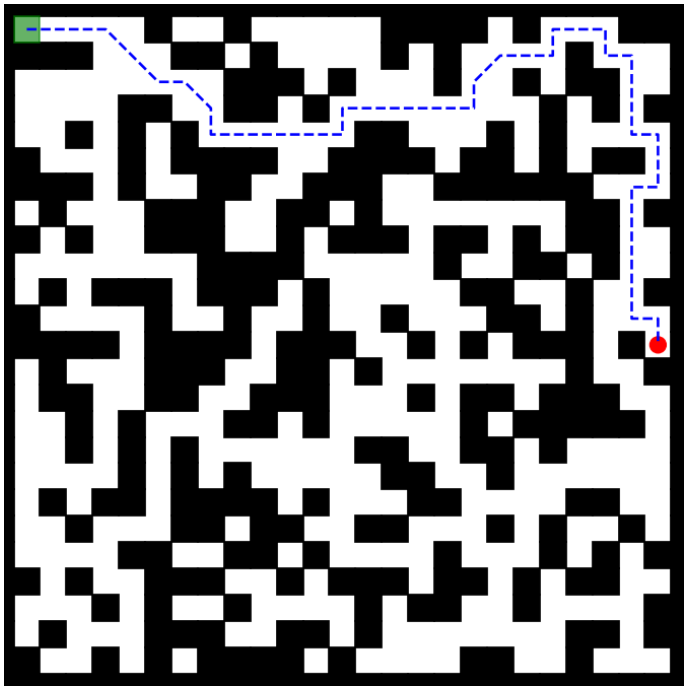


Figure 111: Grid #22: 25x25 sized grid showing the optimal path from start (green square) to end (red circle)

Figure 112: Grid #22: Exploration heatmap of A for a 25x25 sized grid*

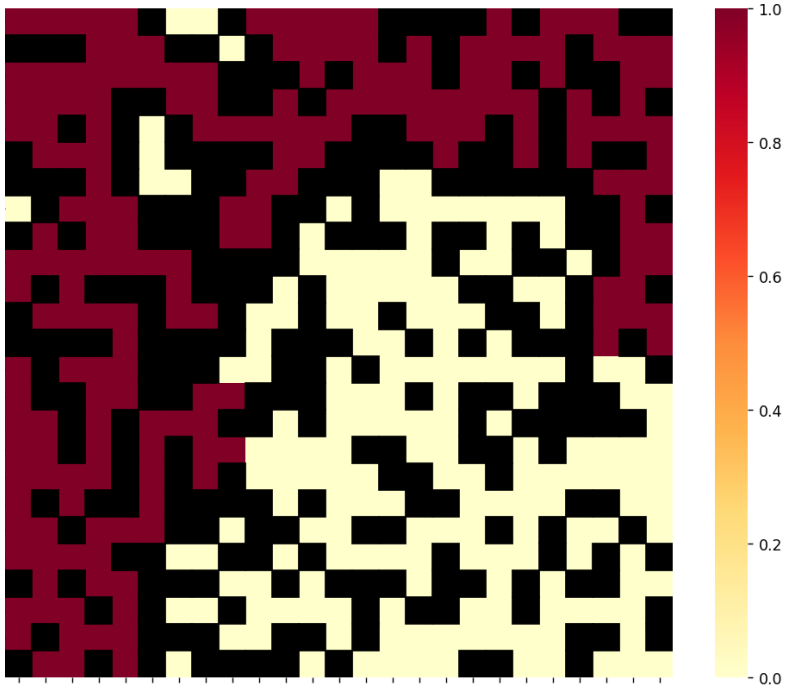


Figure 113: Grid #22: Exploration heatmap of Dijkstra for a 25x25 sized grid

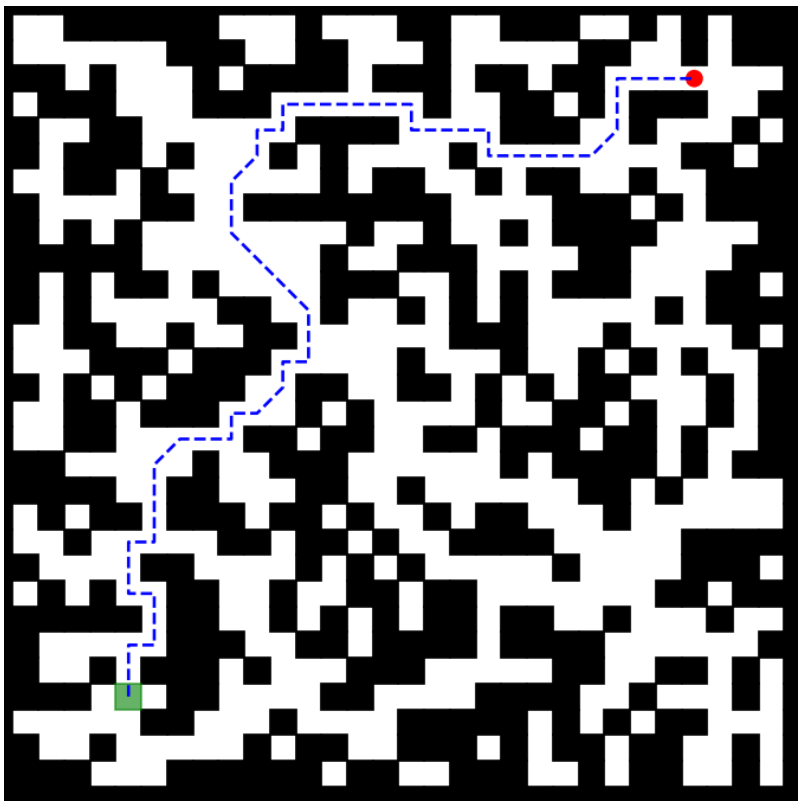


Figure 114: Grid #23: 30x30 sized grid showing the optimal path from start (green square) to end (red circle)

Figure 115: Grid #23: Exploration heatmap of A for a 30x30 sized grid*

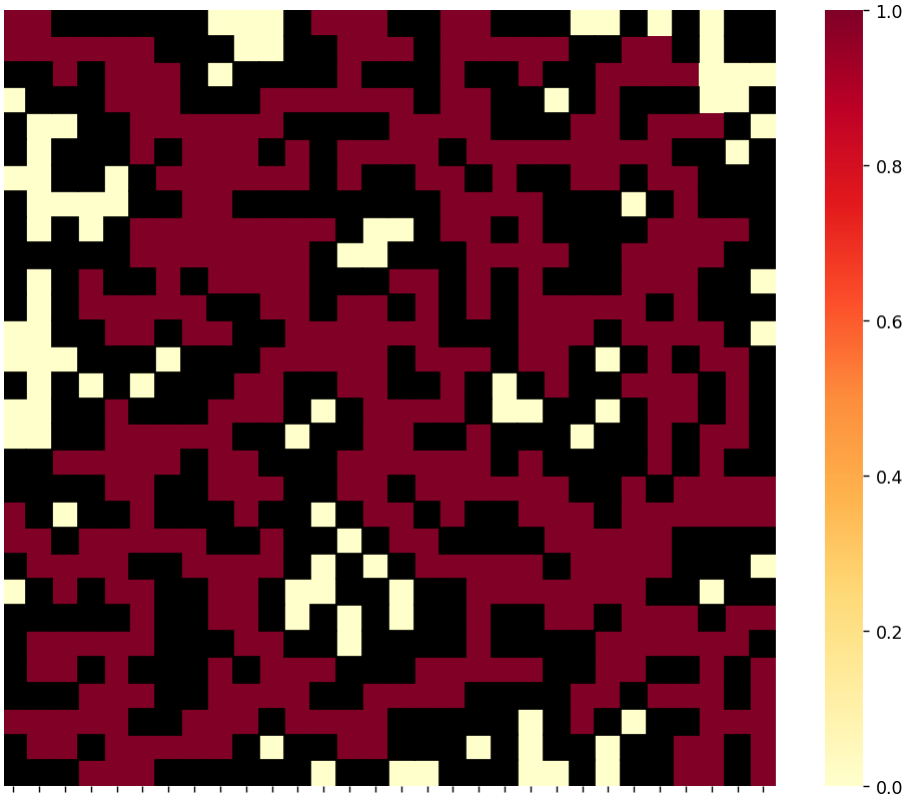


Figure 116: Grid #23: Exploration heatmap of Dijkstra for a 30x30 sized grid

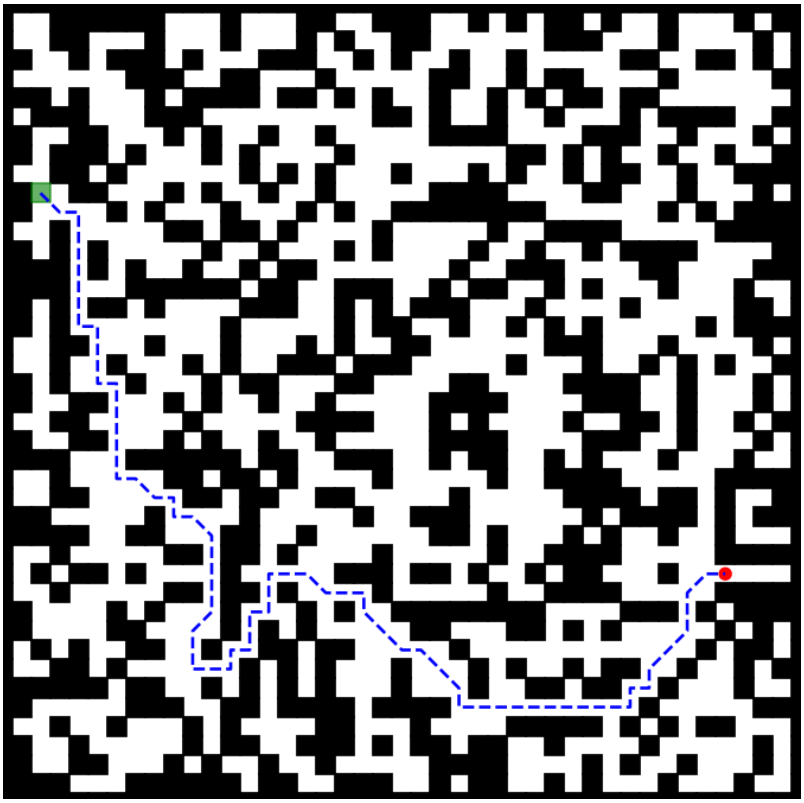


Figure 117: Grid #24: 40x40 sized grid showing the optimal path from start (green square) to end (red circle)

Figure 118: Grid #24: Exploration heatmap of A for a 40x40 sized grid*

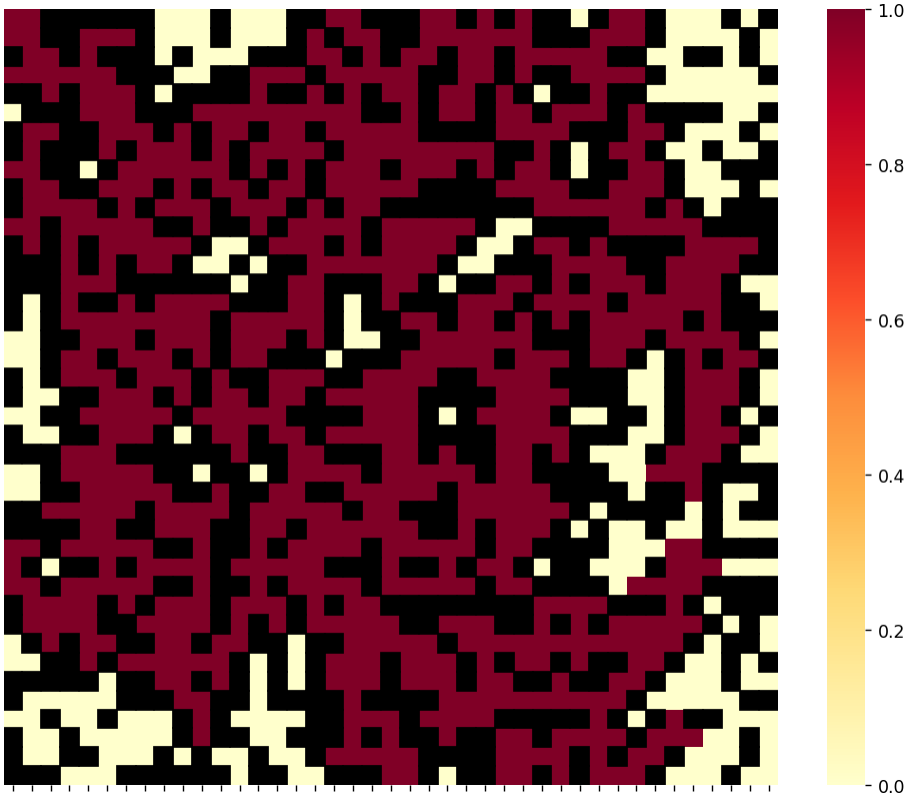


Figure 119: Grid #24: Exploration heatmap of Dijkstra for a 40x40 sized grid

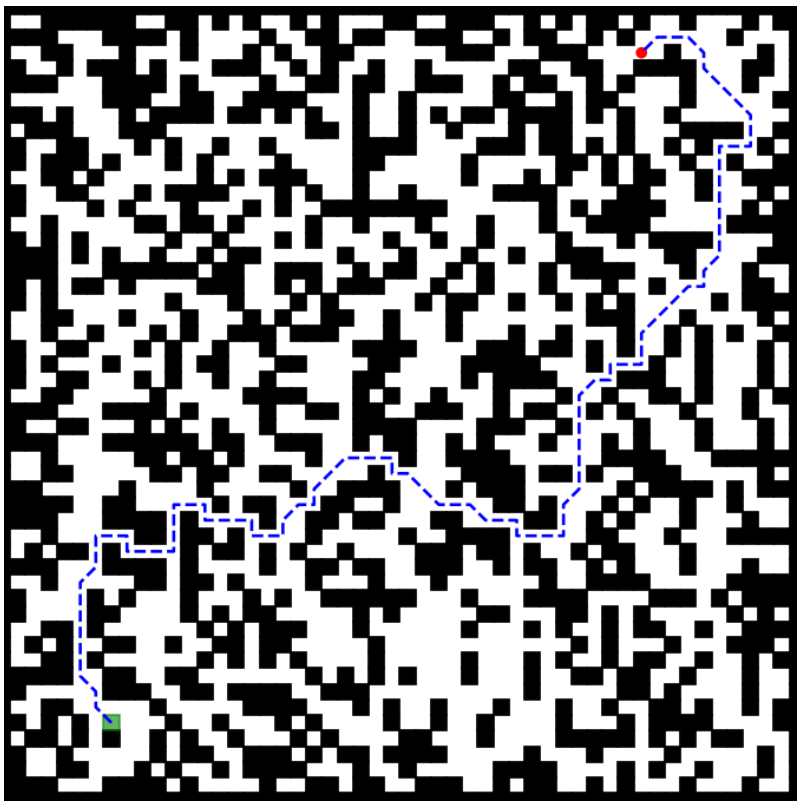


Figure 120: Grid #25: 50x50 sized grid showing the optimal path from start (green square) to end (red circle)

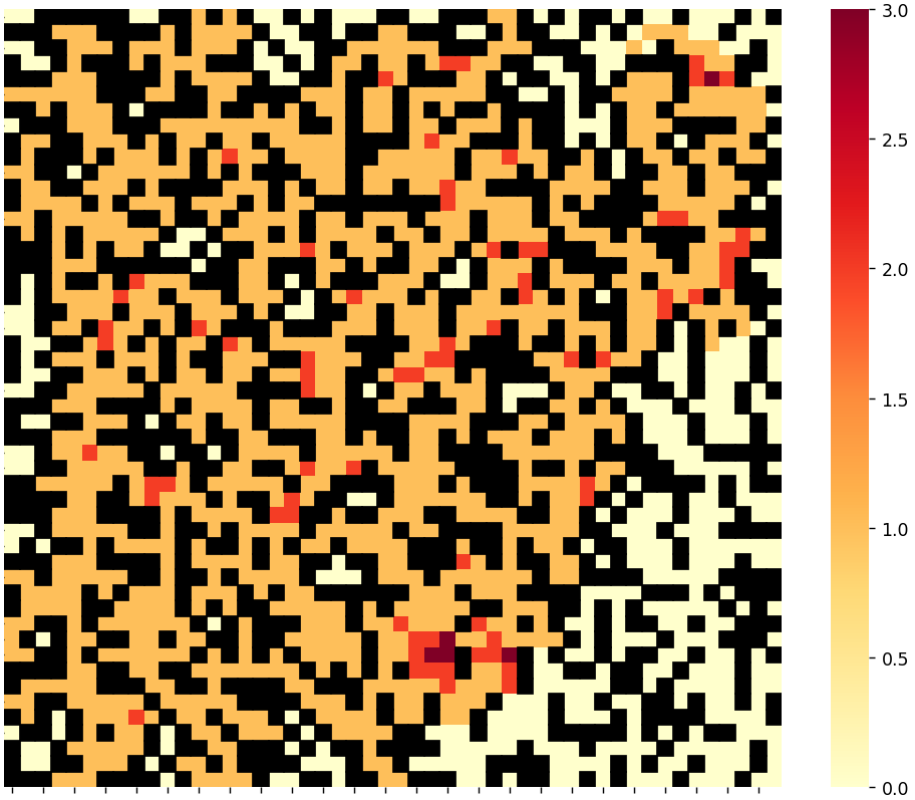


Figure 121: Grid #25: Exploration heatmap of A for a 50x50 sized grid*

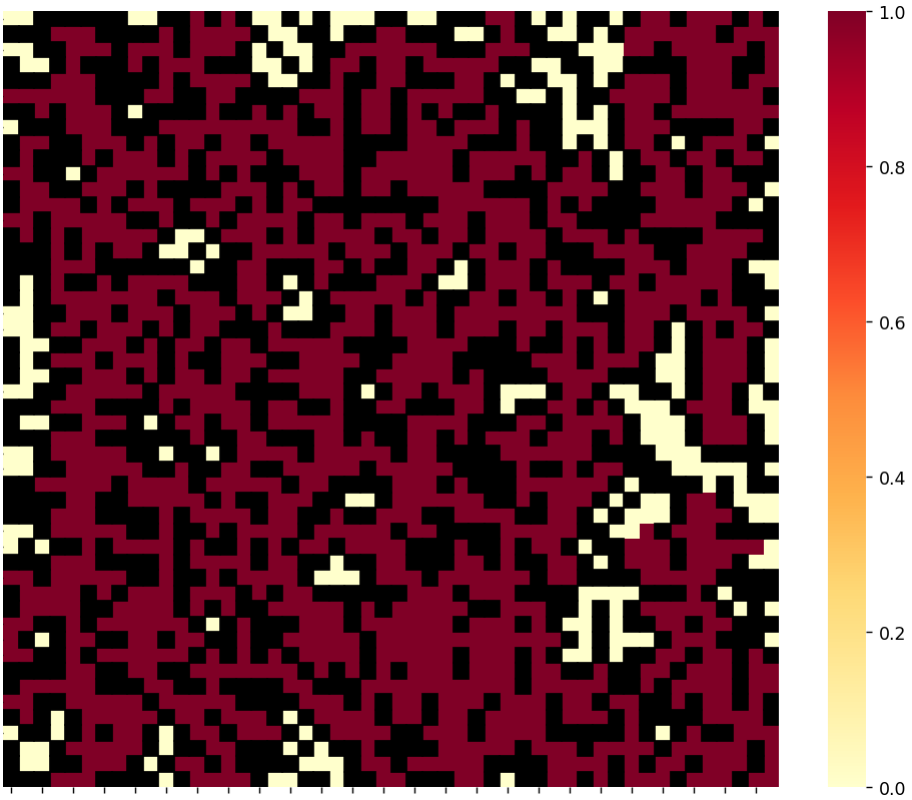


Figure 122: Grid #25: Exploration heatmap of Dijkstra for a 50x50 sized grid

APPENDIX VIII - REAL-WORLD GRIDS

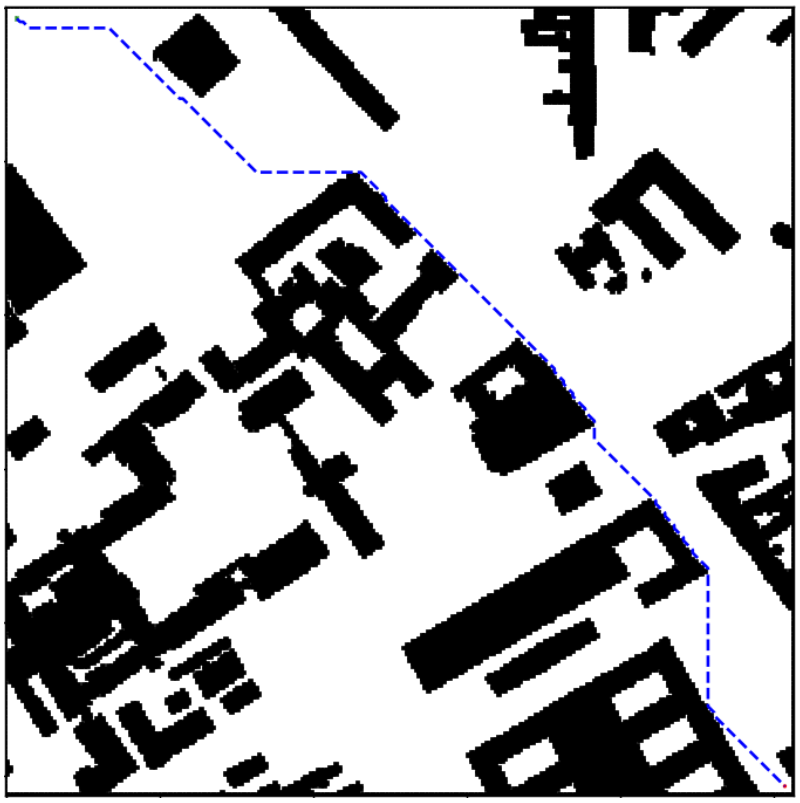


Figure 123: Grid #26: 256x256 sized grid representing Berlin scenario 0 showing the optimal path from start (top-left) to end (bottom-right) [21]

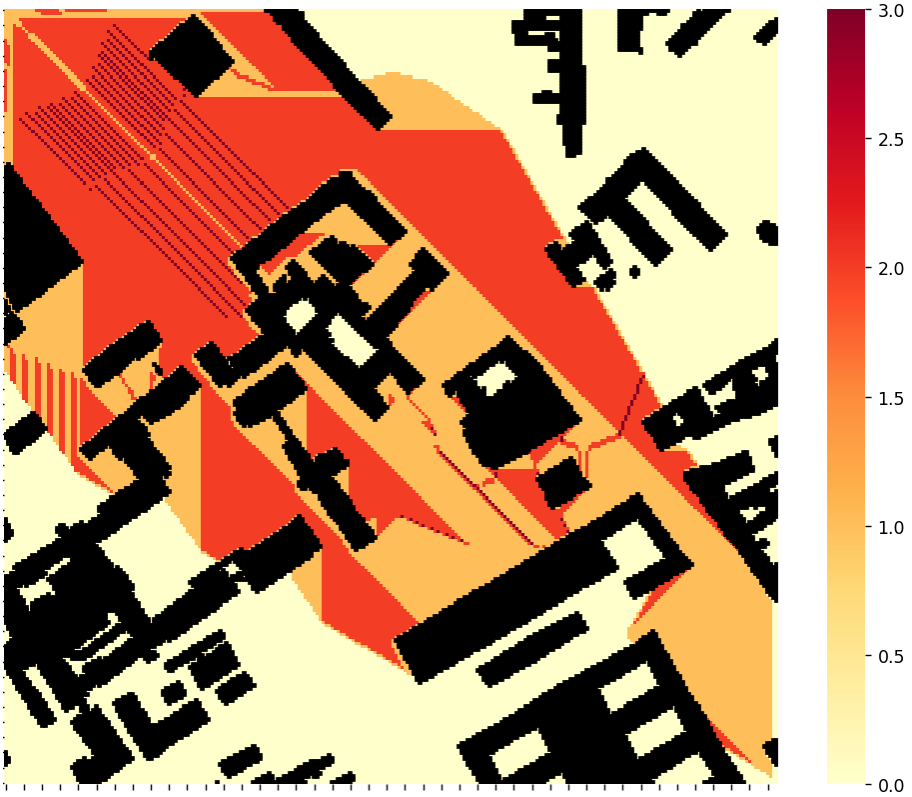


Figure 124: Grid #26: Heatmap of A for a 256x256 sized grid representing Berlin scenario 0 with optimal path from start (top-left) to end (bottom-right) [21]*

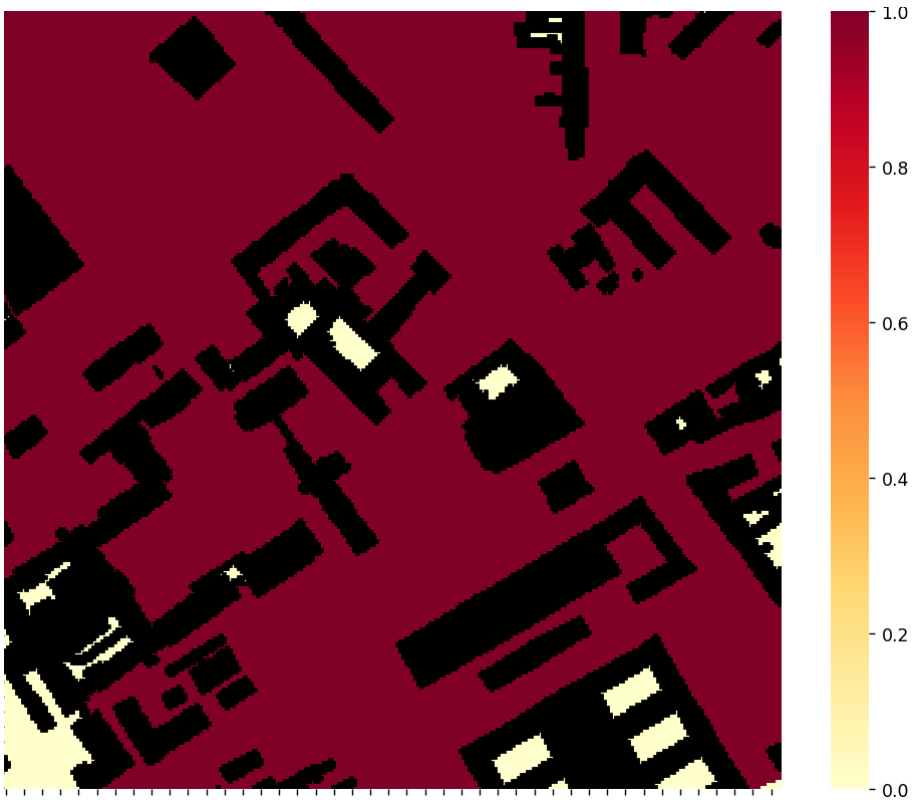


Figure 125: Grid #26: Heatmap of Dijkstra for a 256x256 sized grid representing Berlin scenario 0 with optimal path from start (top-left) to end (bottom-right) [21]

Figure 126: Grid #27: 256x256 sized grid representing London scenario 0 showing the optimal path from start (top-left) to end (middle-right) [21]

Figure 127: Grid #27: Heatmap of A for a 256x256 sized grid representing London scenario 0 with optimal path from start (top-left) to end (middle-right) [21]*

Figure 128: Grid #27: Heatmap of Dijkstra for a 256x256 sized grid representing London scenario 0 with optimal path from start (top-left) to end (middle-right) [21]

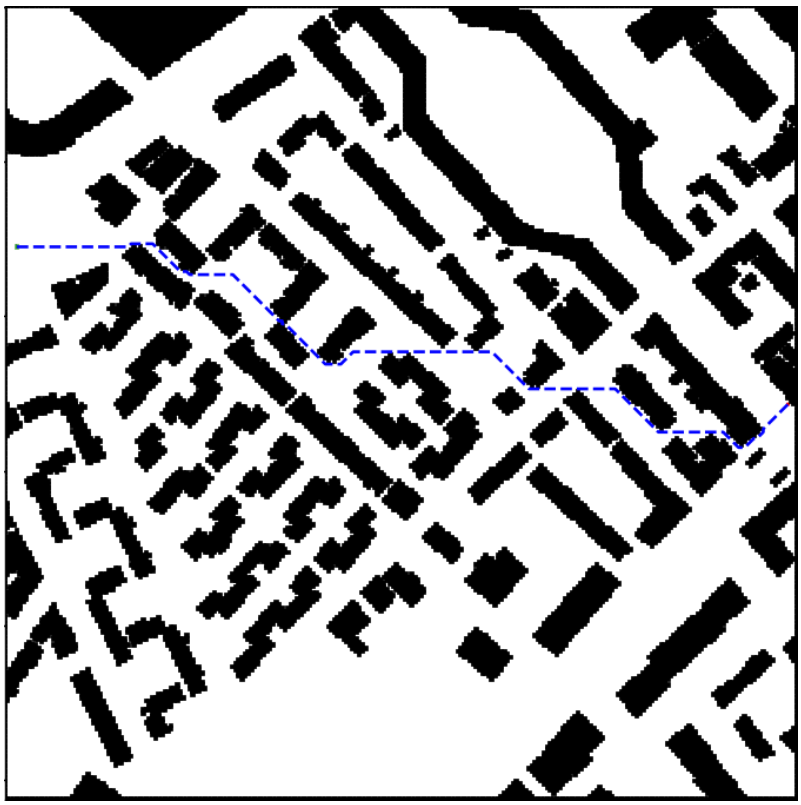


Figure 129: Grid #28: 256x256 sized grid representing Boston scenario 0 showing the optimal path from start (middle-left) to end (middle-right) [21]

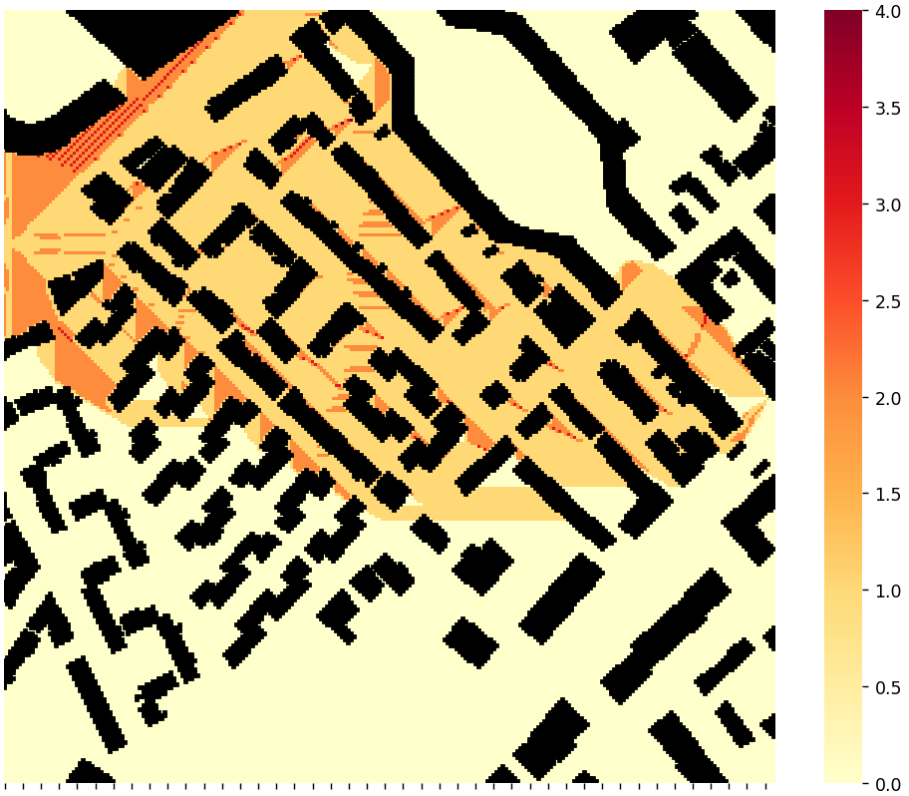


Figure 130: Grid #28: Heatmap of A for a 256x256 sized grid representing Boston scenario 0 with optimal path from start (middle-left) to end (middle-right) [21]*

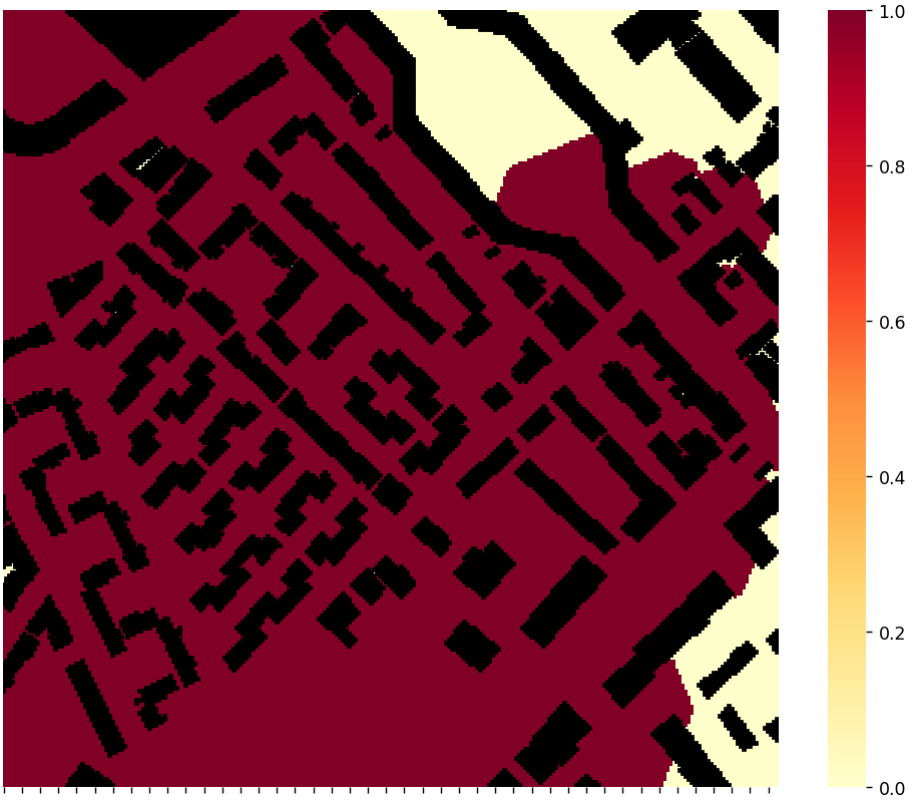


Figure 131: Grid #28: Heatmap of Dijkstra for a 256x256 sized grid representing Boston scenario 0 with optimal path from start (middle-left) to end (middle-right) [21]

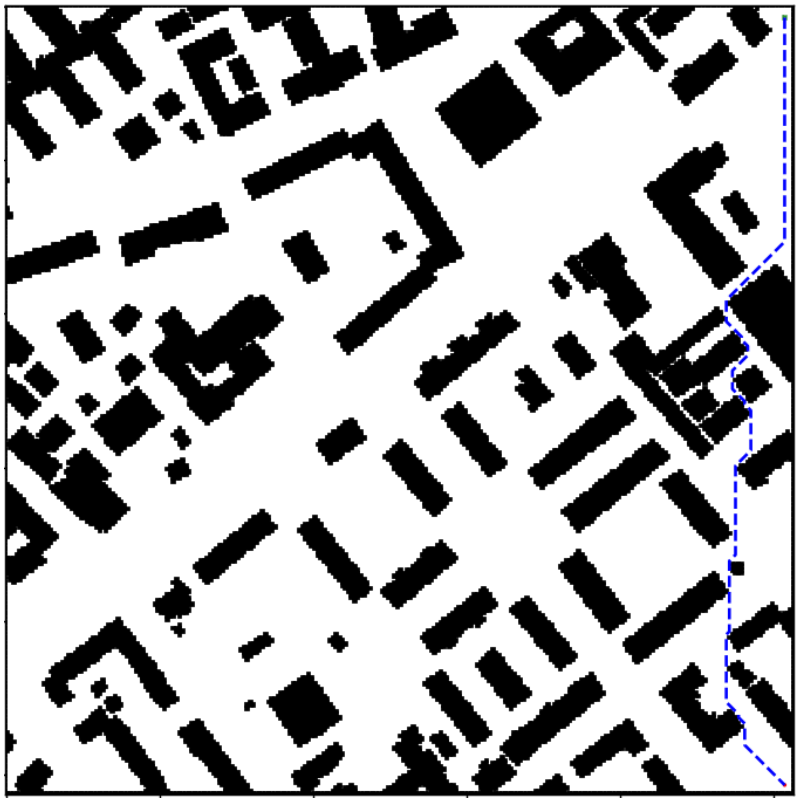


Figure 132: Grid #30: 256x256 sized grid representing Moscow scenario 0 showing the optimal path from start (top-right) to end (bottom-right) [21]

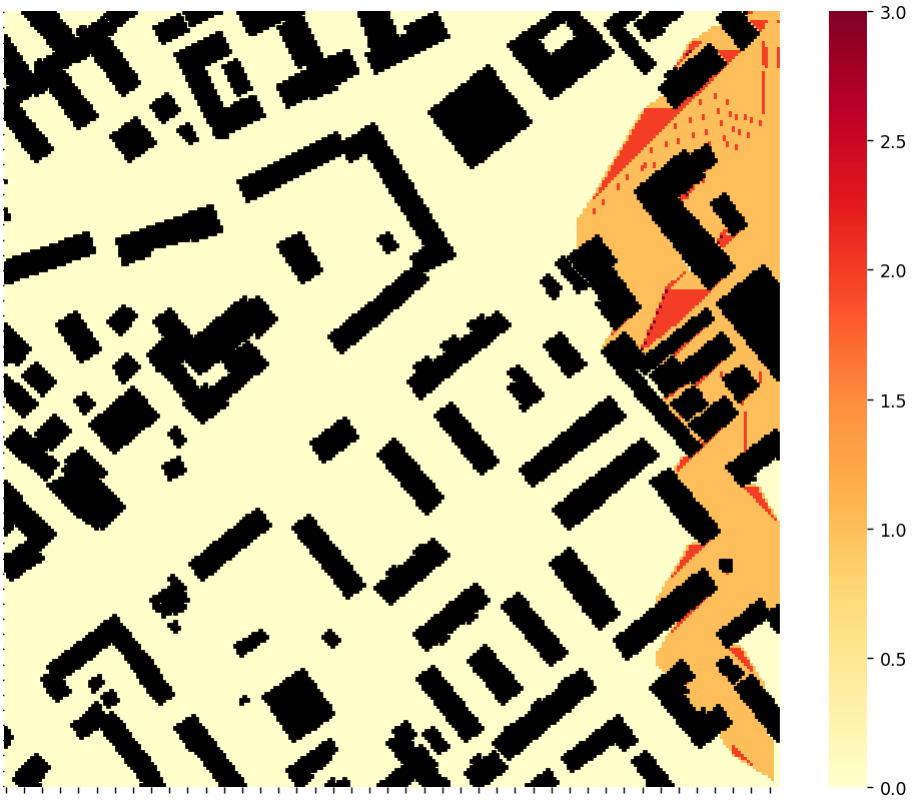


Figure 133: Grid #30: Heatmap of A for a 256x256 sized grid representing Moscow scenario 0 with optimal path from start (top-right) to end (bottom-right) [21]*

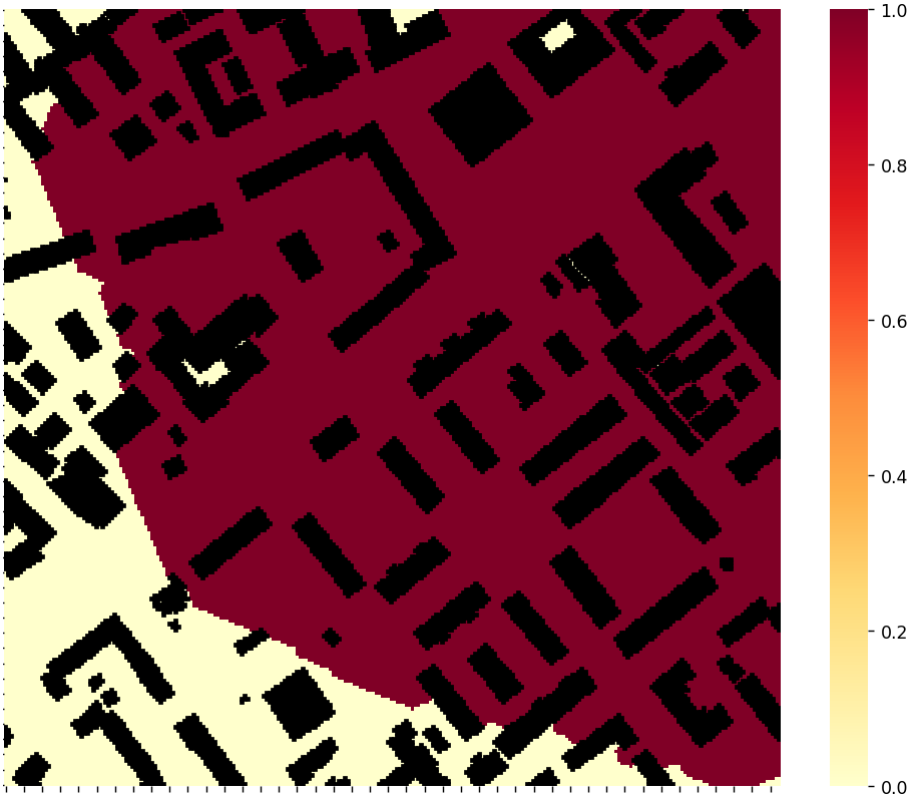


Figure 134: Grid #30: Heatmap of Dijkstra for a 256x256 sized grid representing Moscow scenario 0 with optimal path from start (top-right) to end (bottom-right) [21]

Figure 135: Grid #31: 512x512 sized grid representing Berlin scenario 1 showing the optimal path from start (top-right) to end (bottom-left) [21]

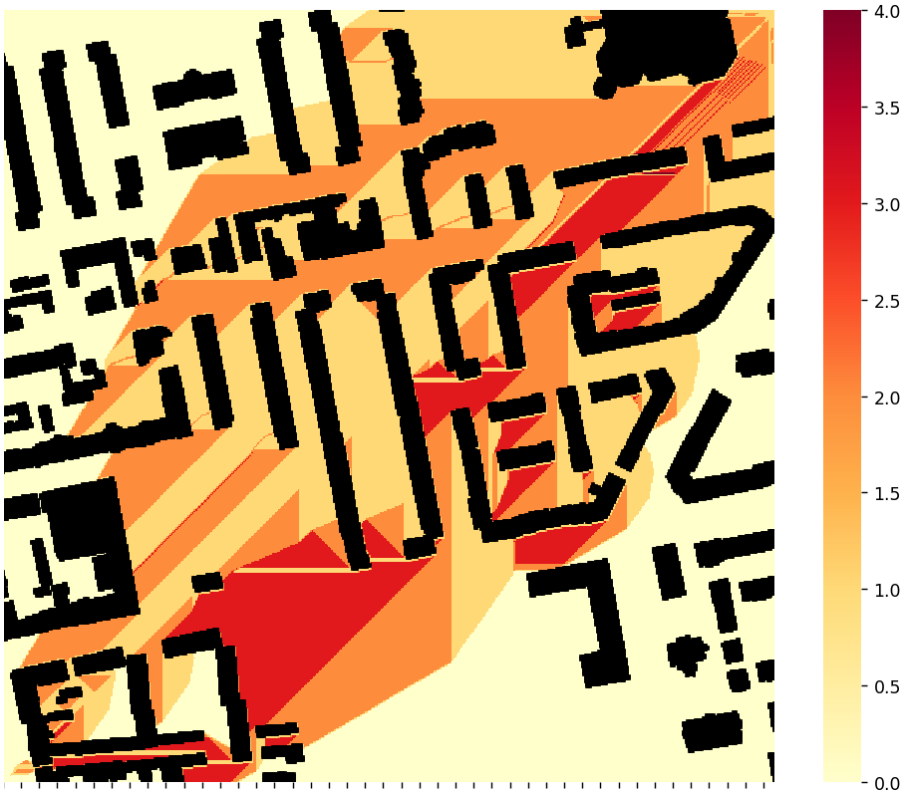


Figure 136: Grid #31: Heatmap of A for a 512x512 sized grid representing Berlin scenario 1 with optimal path from start (top-right) to end (bottom-left) [21]*

Figure 137: Grid #31: Heatmap of Dijkstra for a 512x512 sized grid representing Berlin scenario 1 with optimal path from start (top-right) to end (bottom-left) [21]

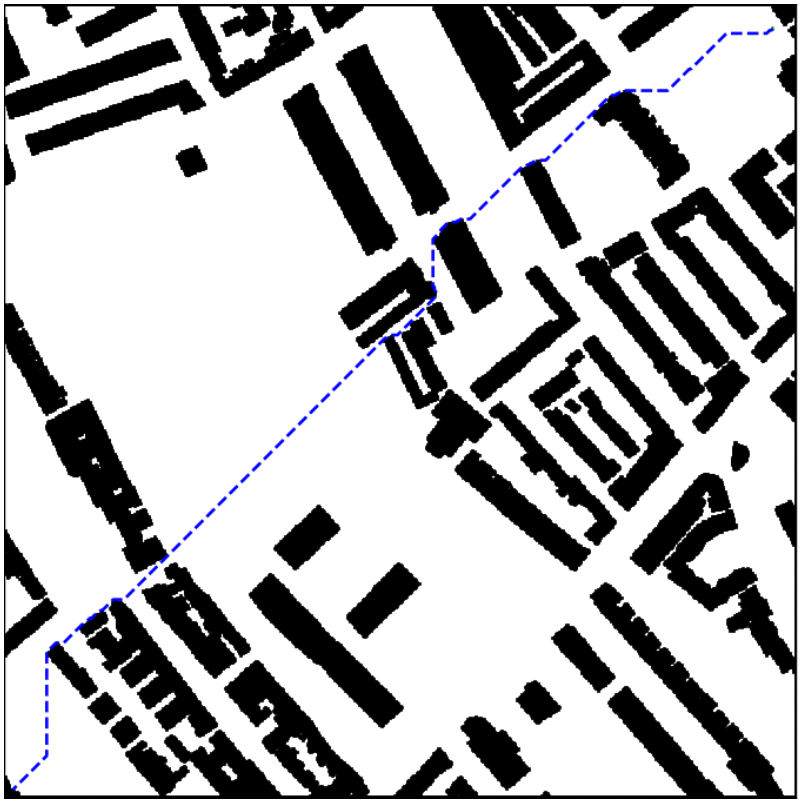


Figure 138: Grid #32: 512x512 sized grid representing London scenario 1 showing the optimal path from start (top-right) to end (bottom-left) [21]

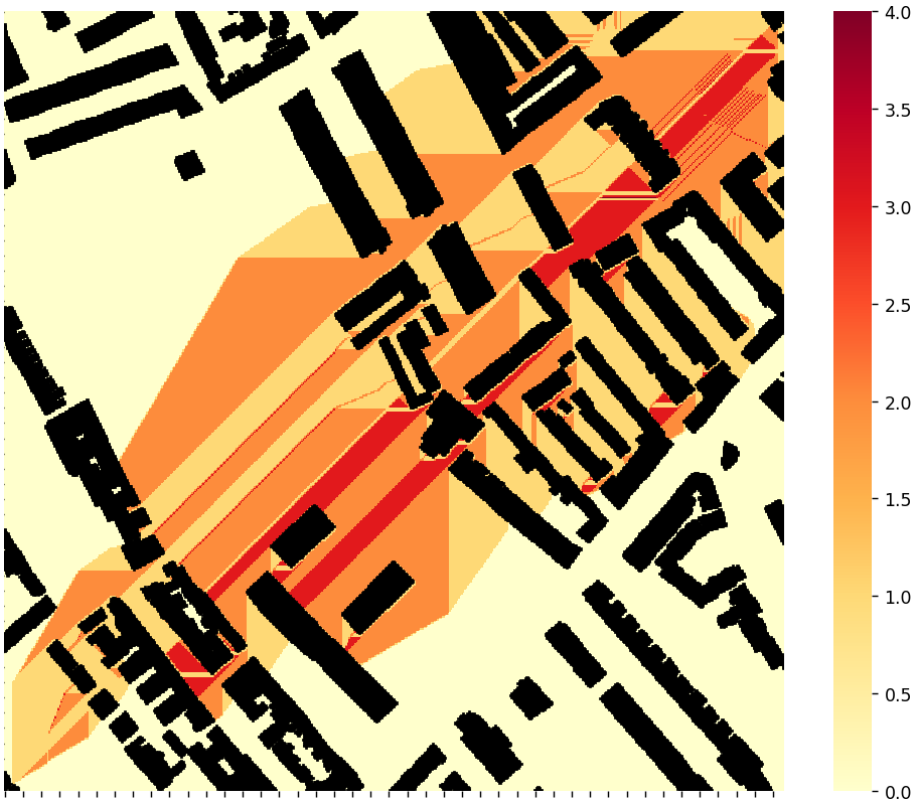


Figure 139: Grid #32: Heatmap of A for a 512x512 sized grid representing London scenario 1 with optimal path from start (top-right) to end (bottom-left) [21]*

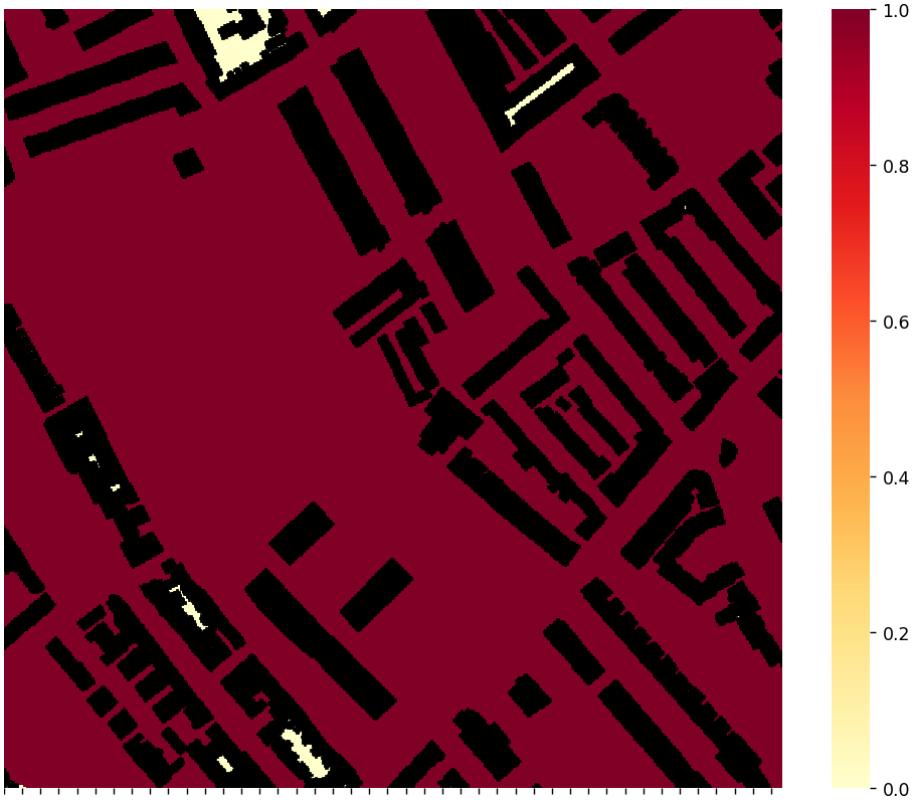


Figure 140: Grid #32: Heatmap of Dijkstra for a 512x512 sized grid representing London scenario 1 with optimal path from start (top-right) to end (bottom-left) [21]

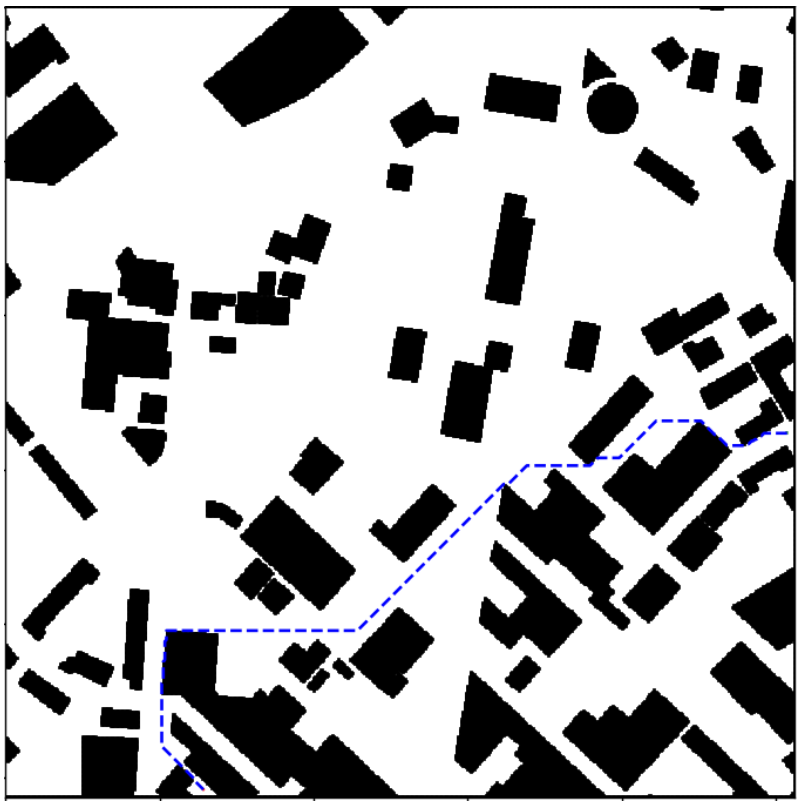


Figure 141: Grid #33: 512x512 sized grid representing Boston scenario 1 showing the optimal path from start (right-middle) to end (bottom-middle) [21]

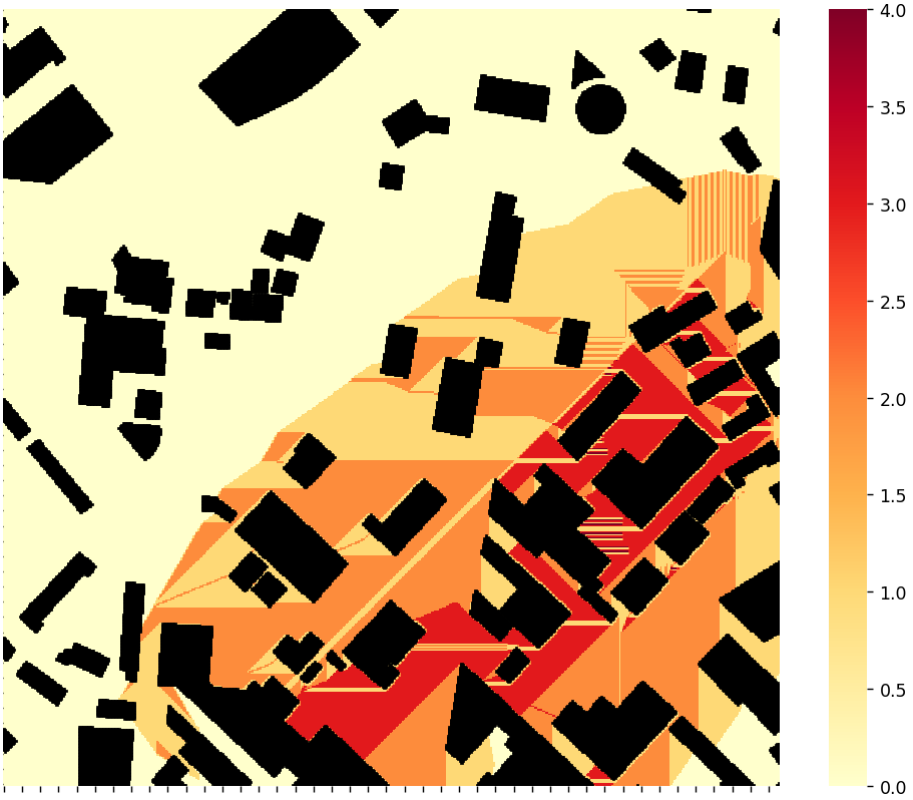


Figure 142: Grid #33: Heatmap of A for a 512x512 sized grid representing Boston scenario 1 with optimal path from start (right-middle) to end (bottom-middle) [21]*

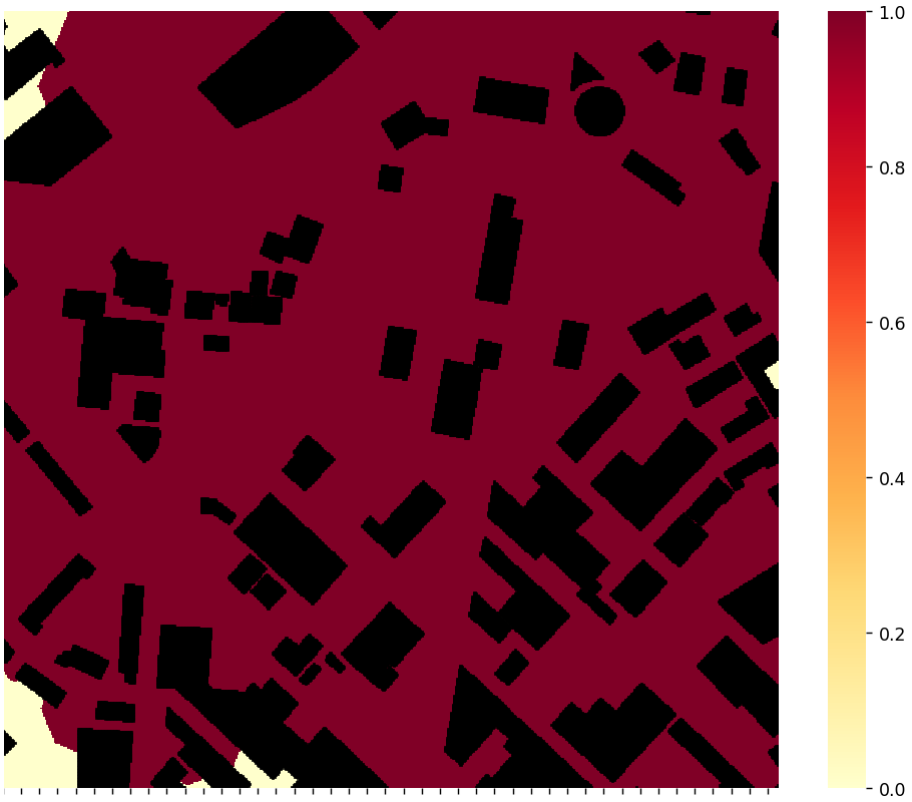


Figure 143: Grid #33: Heatmap of Dijkstra for a 512x512 sized grid representing Boston scenario 1 with optimal path from start (right-middle) to end (bottom-middle) [21]

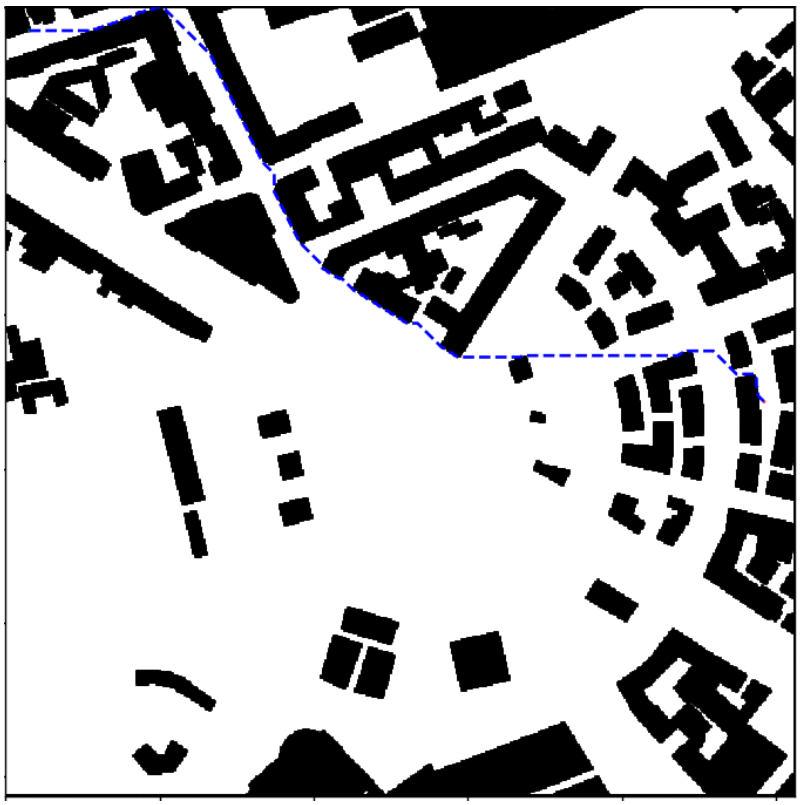


Figure 144: Grid #34: 512x512 sized grid representing Milan scenario 1 showing the optimal path from start (top-left) to end (middle-right) [21]

Figure 145: Grid #34: Heatmap of A for a 512x512 sized grid representing Milan scenario 1 with optimal path from start (top-left) to end (middle-right) [21]*

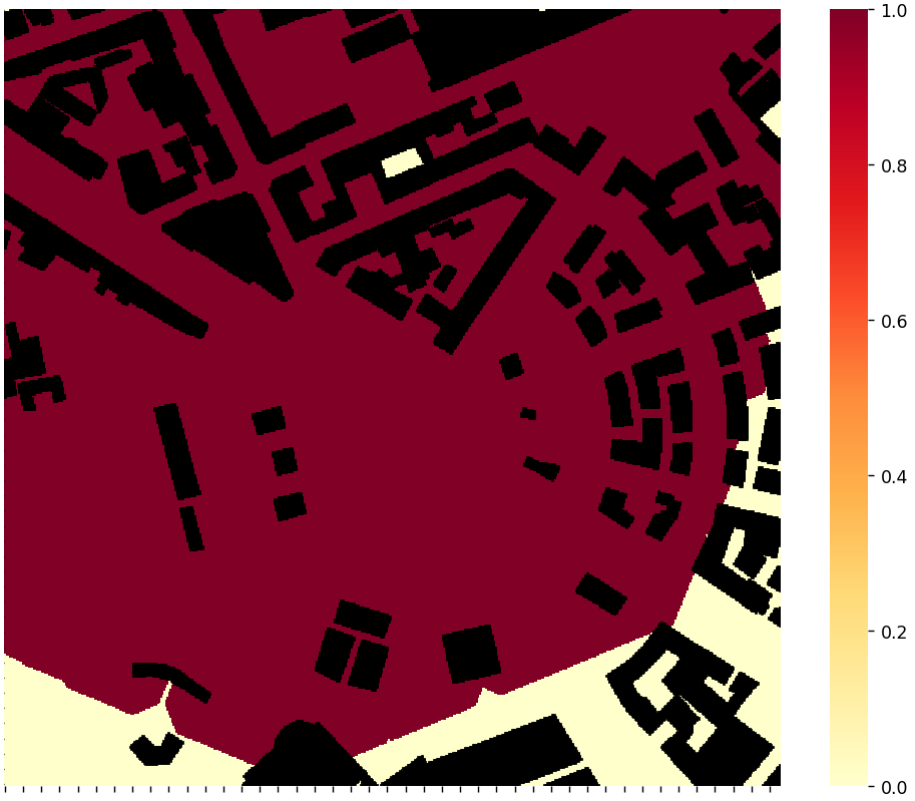


Figure 146: Grid #34: Heatmap of Dijkstra for a 512x512 sized grid representing Milan scenario 1 with optimal path from start (top-left) to end (middle-right) [21]

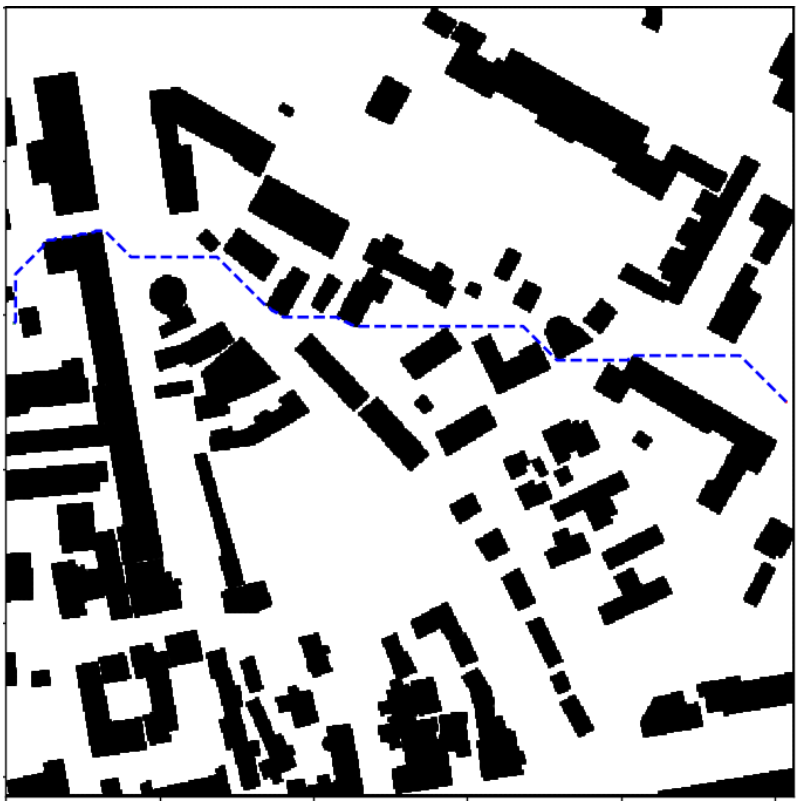


Figure 147: Grid #35: 512x512 sized grid representing Moscow scenario 1 showing the optimal path from start (middle-left) to end (middle-right) [21]

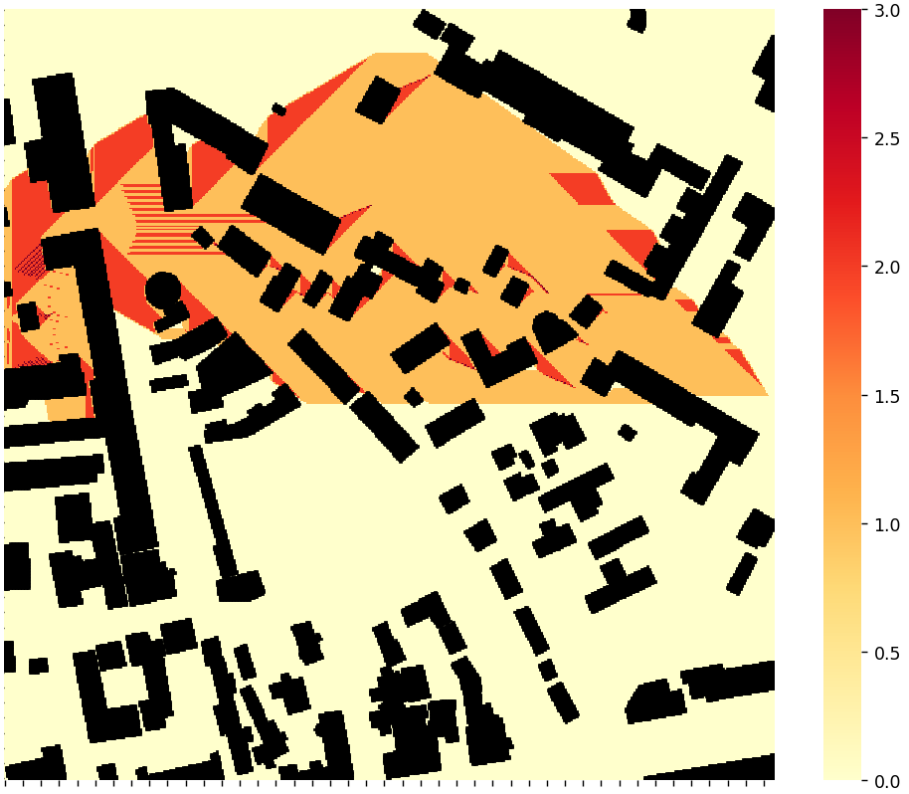


Figure 148: Grid #35: Heatmap of A for a 512x512 sized grid representing Moscow scenario 1 with optimal path from start (middle-left) to end (bottom-right) [21]*

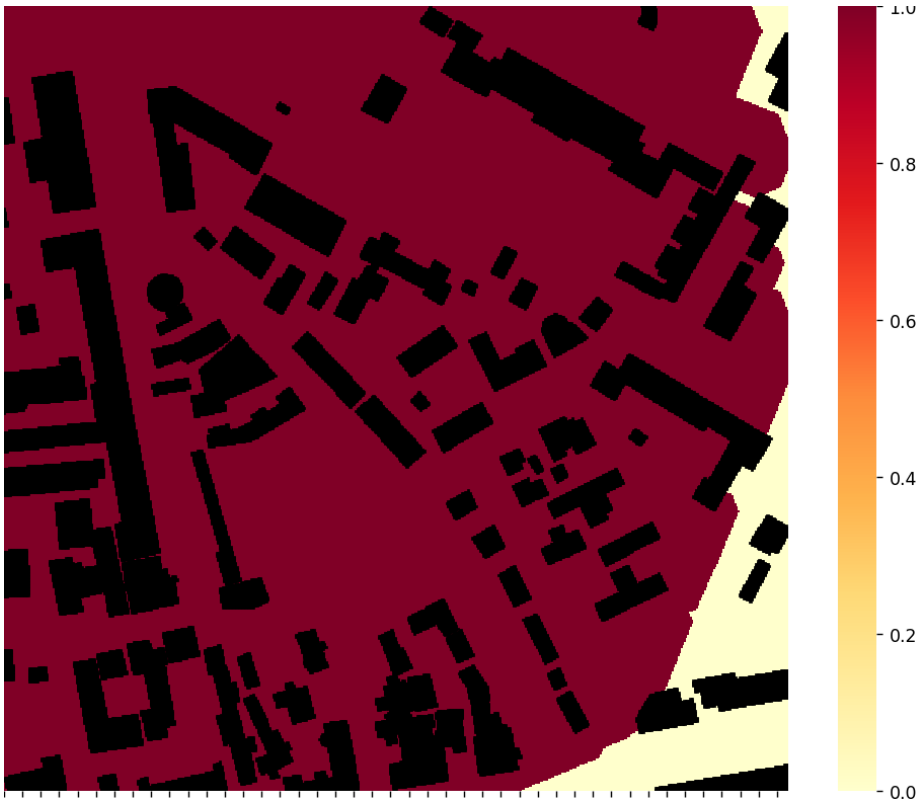


Figure 149: Grid #35: Heatmap of Dijkstra for a 512x512 sized grid representing Moscow scenario 1 with optimal path from start (middle-left) to end (bottom-right) [21]

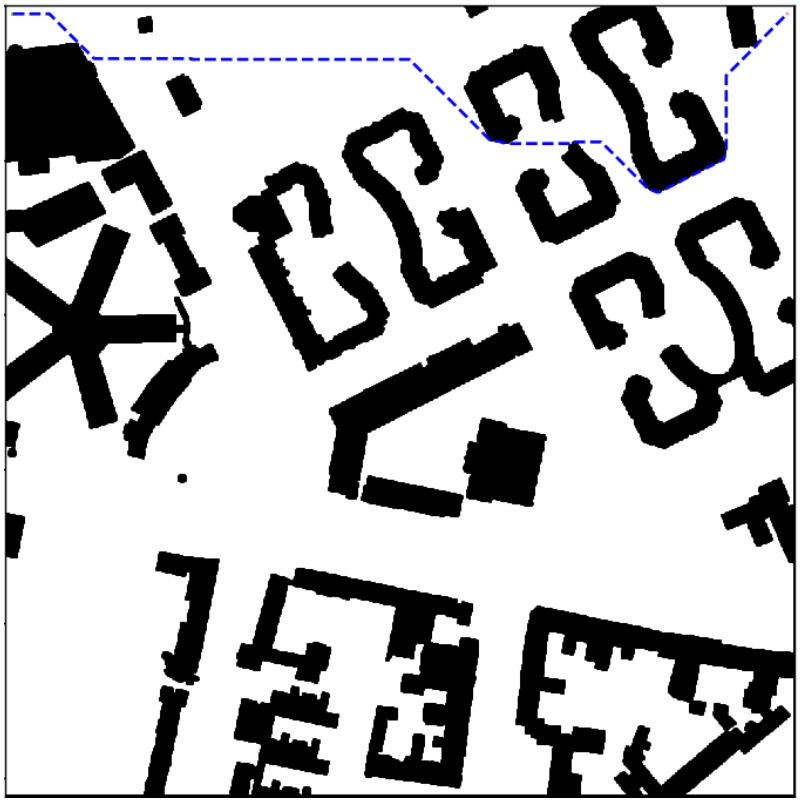


Figure 150: Grid #36: 1024x1024 sized grid representing Berlin scenario 2 showing the optimal path from start (top-left) to end (top-right) [21]

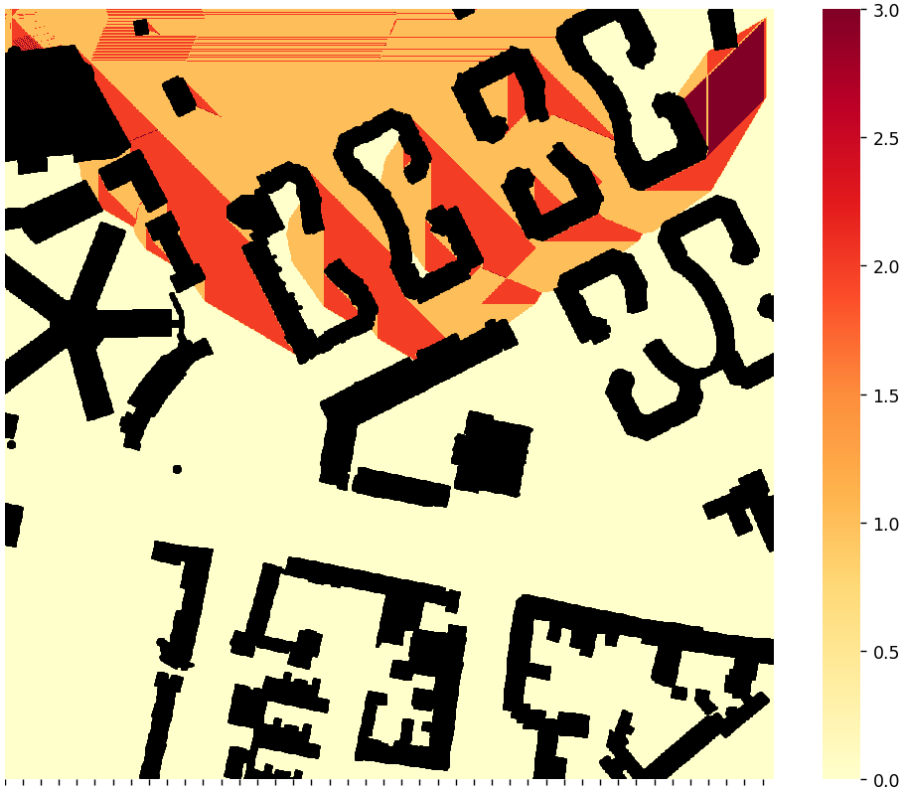


Figure 151: Grid #36: Heatmap of A for a 1024x1024 sized grid representing Berlin scenario 2 with optimal path from start (top-left) to end (top-right) [21]*

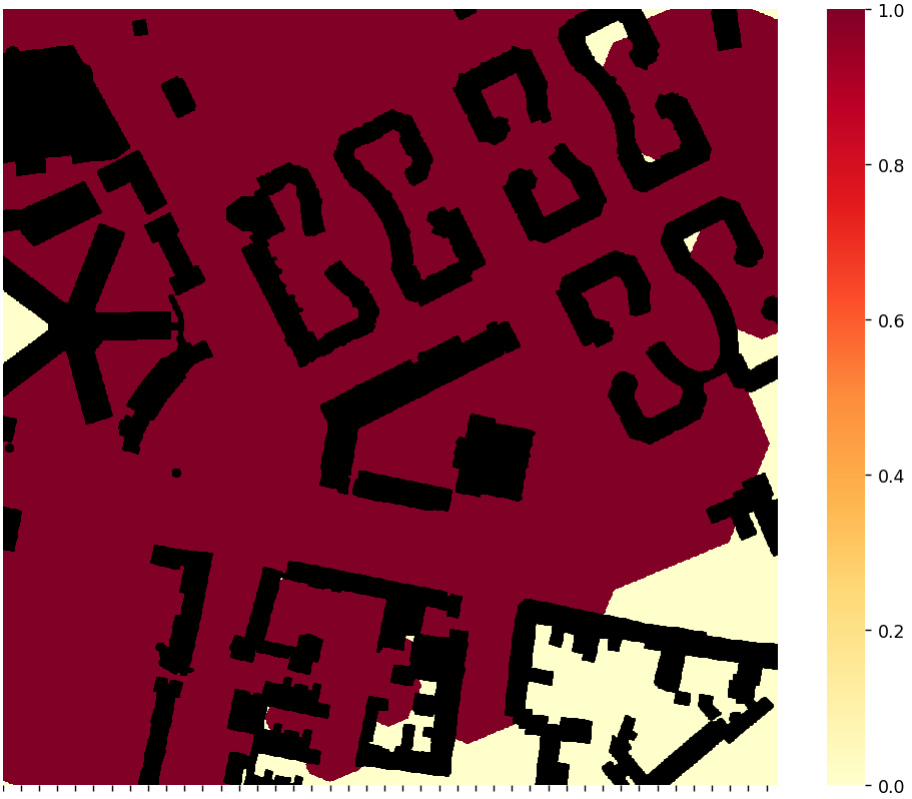


Figure 152: Grid #36: Heatmap of Dijkstra for a 1024x1024 sized grid representing Berlin scenario 2 with optimal path from start (top-left) to end (top-right) [21]

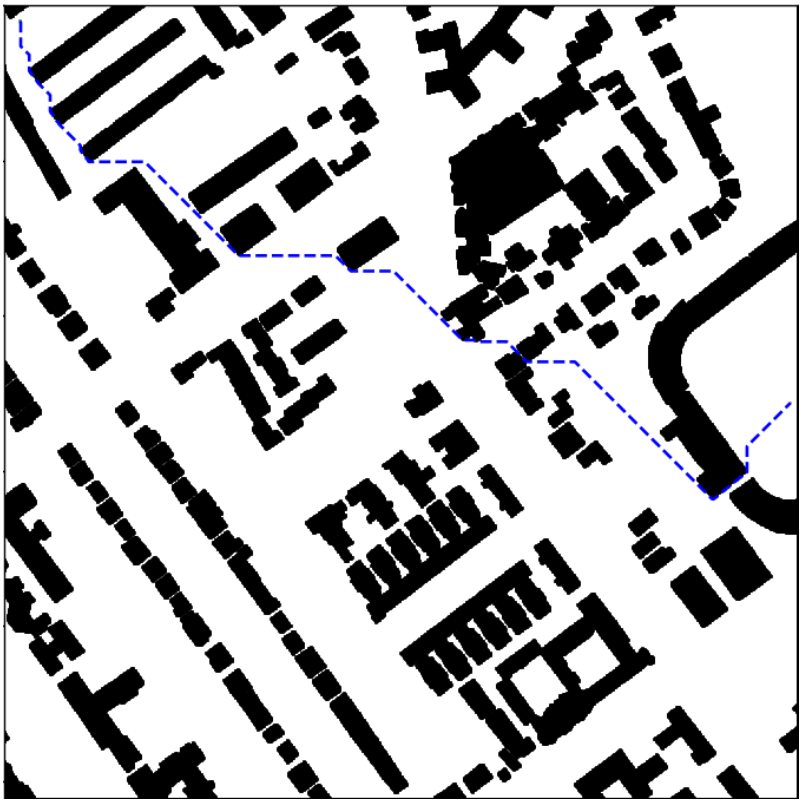


Figure 153: Grid #37: 1024x1024 sized grid representing London scenario 2 showing the optimal path from start (top-left) to end (middle-right) [21]

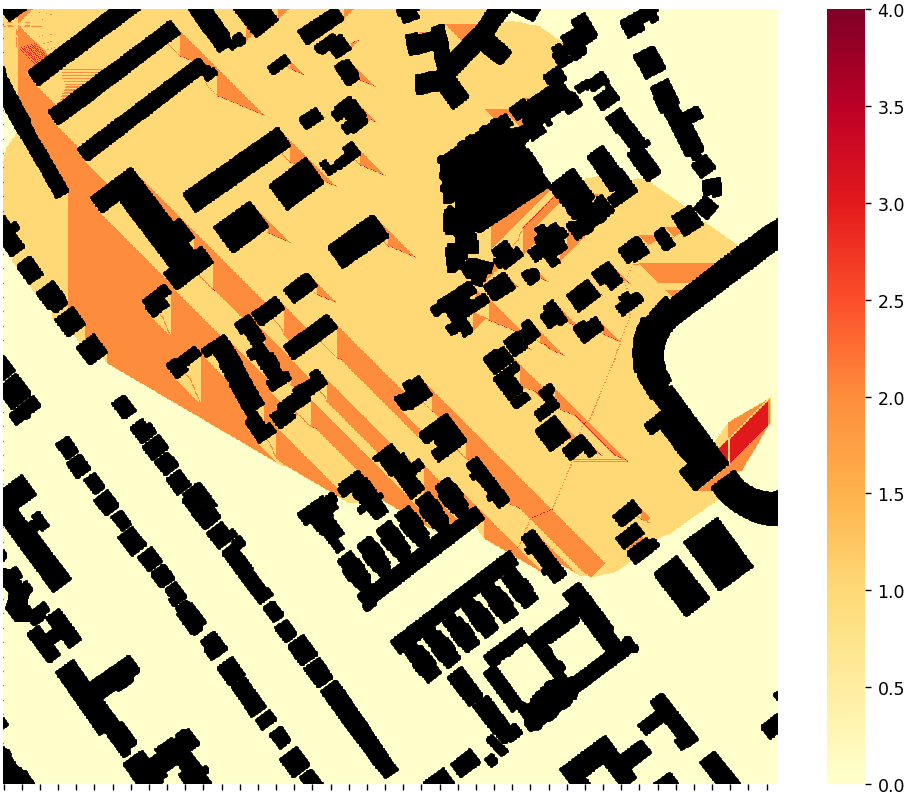


Figure 154: Grid #37: Heatmap of A for a 1024x1024 sized grid representing London scenario 2 with optimal path from start (top-left) to end (top-right) [21]*

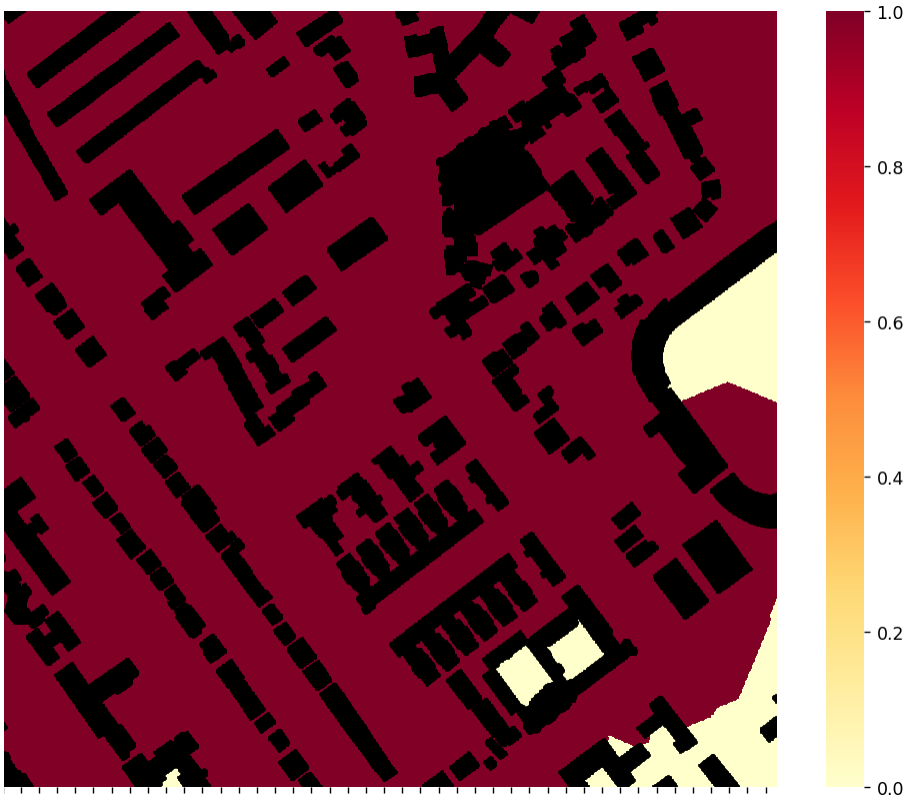


Figure 155: Grid #37: Heatmap of Dijkstra for a 1024x1024 sized grid representing London scenario 2 with optimal path from start (top-left) to end (top-right) [21]

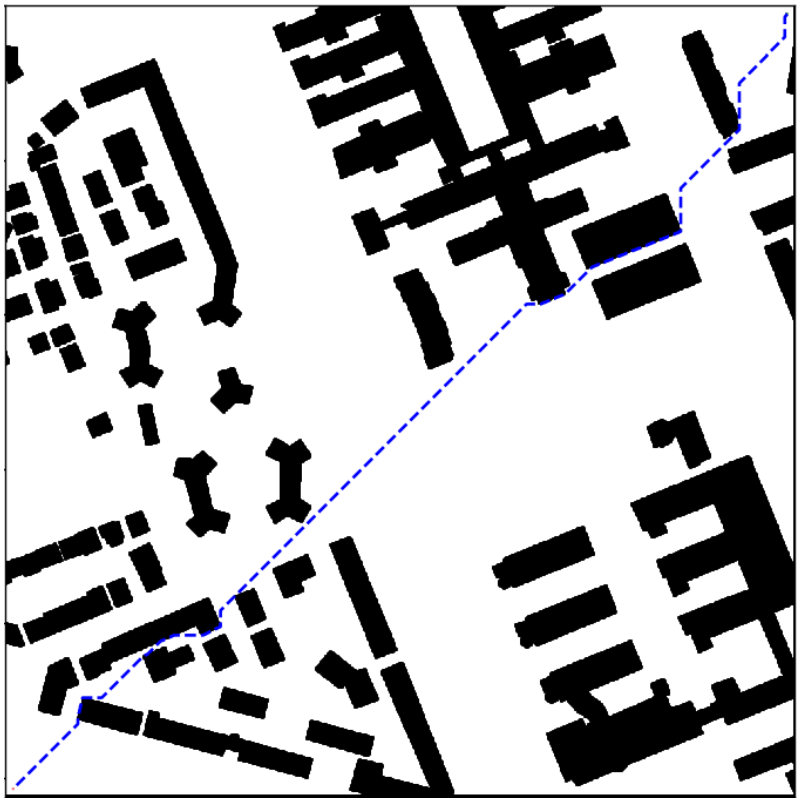


Figure 156: Grid #39: 1024x1024 sized grid representing Milan scenario 2 showing the optimal path from start (top-right) to end (bottom-left) [21]

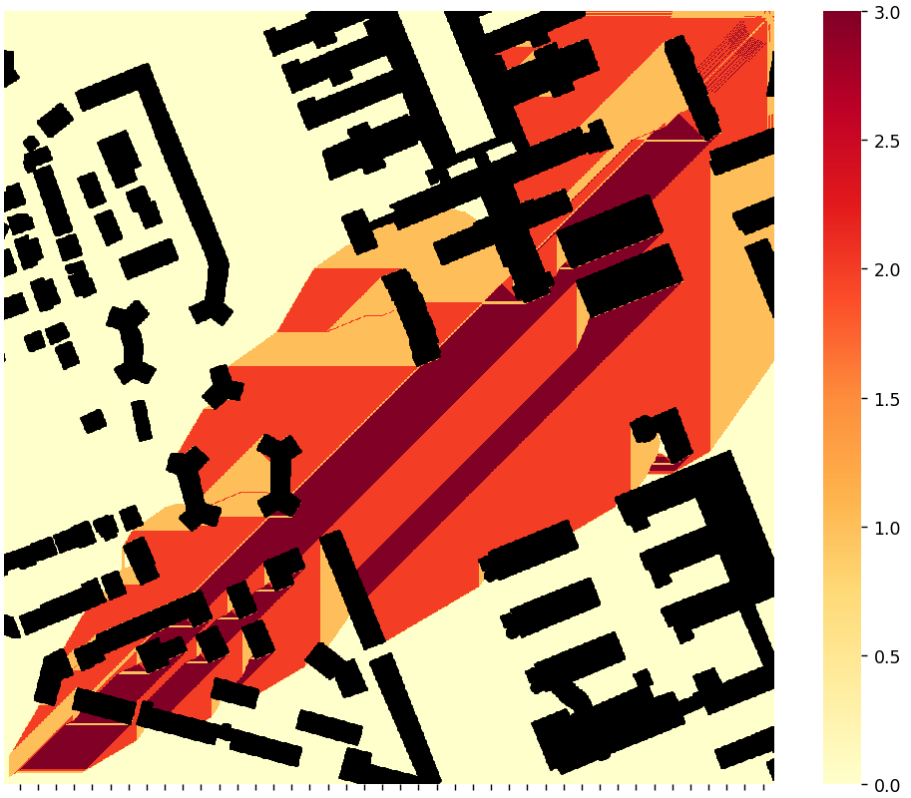


Figure 157: Grid #39: Heatmap of A for a 1024x1024 sized grid representing Milan scenario 2 with optimal path from start (top-right) to end (bottom-left) [21]*

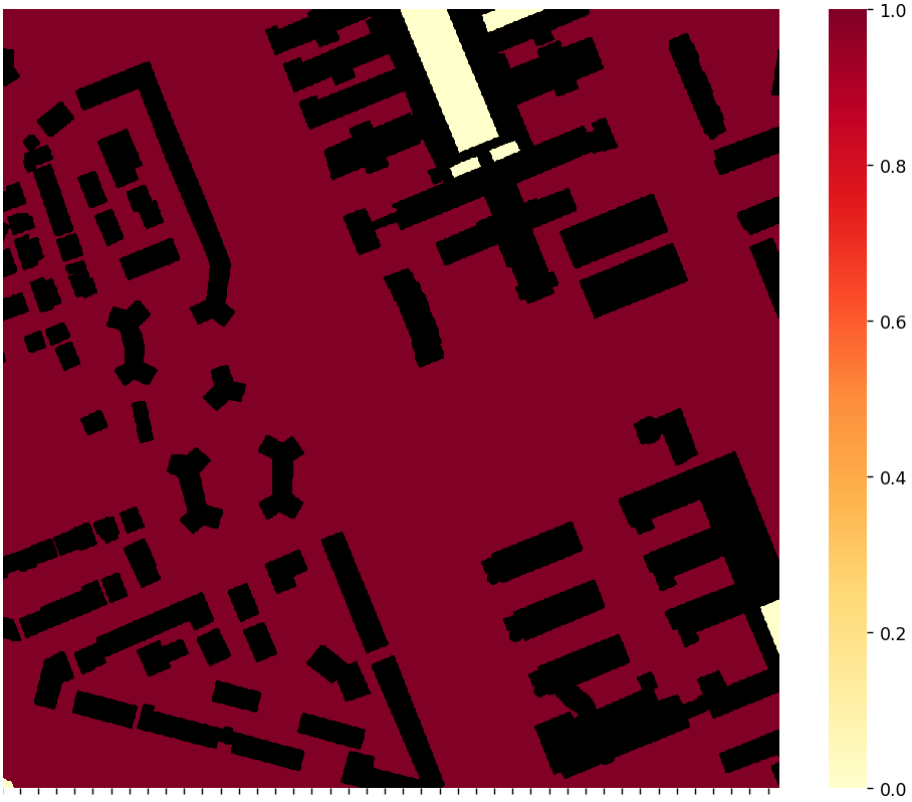


Figure 158: Grid #39: Heatmap of Dijkstra for a 1024x1024 sized grid representing Milan scenario 2 with optimal path from start (top-right) to end (bottom-left) [21]

Figure 159: Grid #40: 1024x1024 sized grid representing Moscow scenario 2 showing the optimal path from start (top-left) to end (bottom-right) [21]

Figure 160: Grid #40: Heatmap of A for a 1024x1024 sized grid representing Milan scenario 2 with optimal path from start (top-left) to end (bottom-right) [21]*

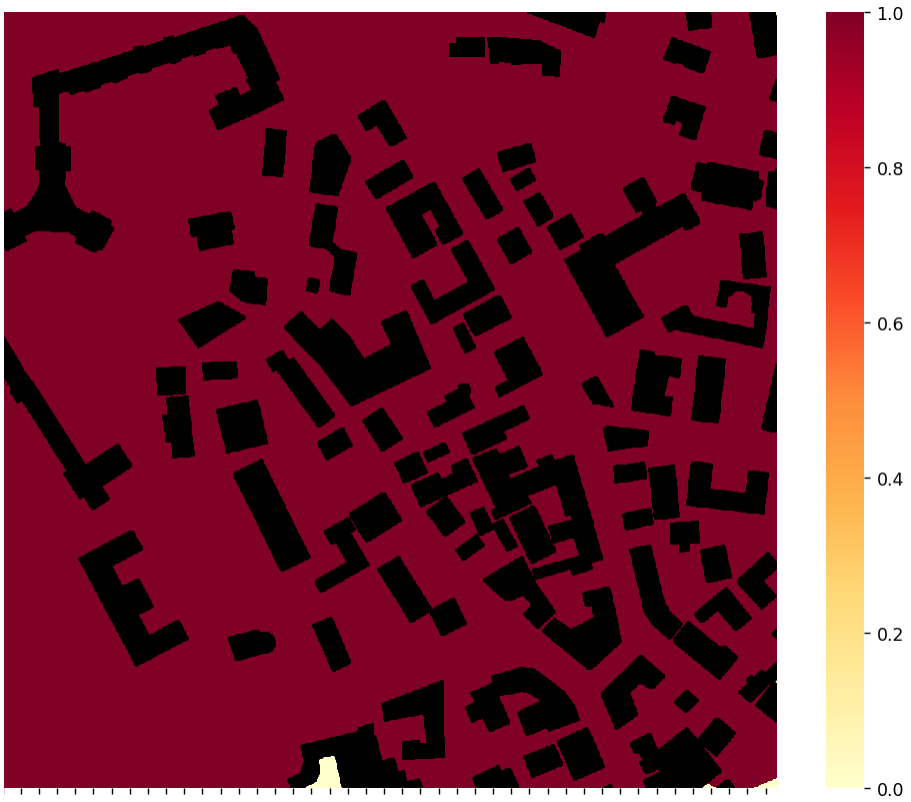


Figure 161: Grid #40: Heatmap of Dijkstra for a 1024x1024 sized grid representing Milan scenario 2 with optimal path from start (top-left) to end (bottom-right) [21]