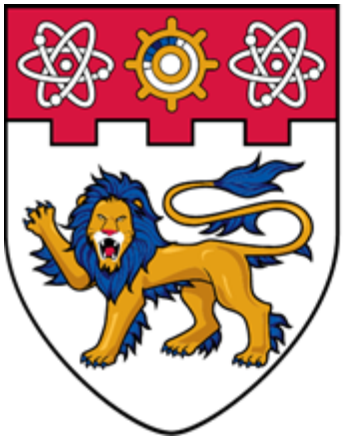


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SCSE21-0115

Cloud Driven Implementation for Multi-Agent Path Finding

Authored by

Anusha Datta

Supervised by

Assoc. Prof. Xueyan Tang

Submitted in Partial Fulfilment of the Requirements for the Degree of Bachelor of Engineering (Computer Science) from Nanyang Technological University

School of Computer Science and Engineering (SCSE)

Abstract

Multi-Agent Path Finding (MAPF) is the computational problem of constructing collision-free paths for a set of agents from their respective start to goal positions within a given maze. In recent years, MAPF has gained increasing importance as it is central to many large-scale robotic applications, from logistic distribution systems to simultaneous localization and mapping. Over time, numerous approaches to MAPF have emerged, one of which is the dual level Conflict Based Search (CBS) Algorithm. At the high level, CBS performs search on a binary constraint tree. While at the lower level, it performs a search for a single agent at a time. In most cases, this reformulation enables CBS to examine fewer states than a global A* based approached, while still maintaining optimality. Hence, this project explores Conflict Based Search for optimal Multi-Agent Path Finding. These findings are augmented with additional experimentation on search performances of different lower level search heuristics. Furthermore, this project also includes the design, development and deployment of a cloud driven MAPF application. This application aims to provide an intuitive user experience to interact with the MAPF algorithm, visualise the traversal of the path finding solution and record statistical navigation parameters such as execution cost and execution time of the same. Finally, a navigation statistics pipeline is also established to produce strong predictive insights and navigation trends which subsequently facilitate intelligent business decisions.

Acknowledgement

First and foremost, I wish to express my sincere gratitude to my supervisor, Associate Professor Xueyan Tang, for providing me with the opportunity to embark on this meaningful project abundant with diverse learning opportunities. Under the insightful guidance of Professor Xueyan, I was able to truly interject my creativity into this project and fuel my spark for innovation.

Next, I would like to thank my examiner, Associate Professor Anh Tuan Luu, for his perceptive feedback and time in grading this project.

I would also like to express my gratitude to Nanyang Technological University and the School of Computer Science and Engineering (SCSE) for the opportunity to practice and hone the knowledge amassed during my candidature.

Lastly, I wish to thank my family and friends for their continuous support throughout this project and the entirety of my academic journey. This project is entirely dedicated to my parents, who have always been the wind beneath my wings.

Table of Contents

Acronyms

A brief note on the acronym notations abided by this report:

MAPF	**M**ulti-**A**gent **P**ath **F**inding
AI	**A**rtificial **I**ntelligence
CBS	**C**onflict **B**ased **S**earch
CAT	**C**onflict **A**voidance **T**able
CT	**C**onflict **T**ree
UCS	**U**niform **C**ost **S**earch
DB	**D**ata**b**ase
ER	**E**ntity **R**elationship
BCNF	**B**oyce-**C**odd **N**ormal **F**orm
AWS	**A**mazon **W**eb **S**ervices
AWS S3	AWS **S**imple **S**torage **S**ervice
SAM	**S**erverless **A**pplication **M**anager
SDLC	**S**oftware **D**evelopment **L**ife **C**ycle
CLI	**C**ommand **L**ine **I**nterface
API	**A**pplication **P**rogramming **I**nterface
CI/CD	**C**ontinuous **I**ntegration/**C**ontinuous **D**eployment
CORS	**C**ross-**O**rigin **R**esource **S**haring
CDN	**C**ontent **D**elivery **N**etwork
DoS	**D**enial **o**f **S**ervice
JWT	**J**ava **W**eb **T**oken
UAT	**U**ser **A**cceptance **T**esting
GUI	**G**raphical **U**ser **I**nterface
L to R	**L**eft to **R**ight

List of Figures

1. Introduction

In recent years, evolution of the technological landscape has evoked a growing interest in Multi-Agent Path Finding in the AI research community. MAPF is the computational problem of constructing collision-free paths for a set of agents from their respective start to goal positions within a given maze. In other words, it directs path planning for multiple agents with the key constraint that agents must traverse these paths concurrently without colliding with one another. MAPF has a multitude of applications in diverse fields, some of which include autonomous vehicles, automated warehouse systems, multi-robot teams, aircraft management and game characters in video games. As the prevalence of these agents grows, so does the need for strong navigation and path finding algorithms, so the agents may function with minimum human instruction. The importance of these algorithms is significant as robotic automation has numerous benefits including higher speed and efficiency, time and cost savings, safety in hazardous environments and mitigation of human error whilst performing repetitive tasks [1].

Over time, diverse approaches to MAPF have emerged, one of which is the dual level Conflict Based Search (CBS) Algorithm [2]. At the high level, CBS performs search on a binary constraint tree. While at the lower level, it performs a search for a single agent at a time. In most cases, this reformulation enables CBS to examine fewer states than a global A*-based approached, while still maintaining optimality. On that note, this project explores Conflict Based Search for optimal Multi-Agent Path Finding, with A* Search as the chosen lower level solver.

There is immense scope for exploring various lower level solver (A* Search) heuristics used within CBS. Hence, the findings above are augmented with additional experimentation on the effect of different lower level search heuristics on search performances. The heuristics considered include Manhattan Distance, Chebyshev Distance and Euclidian Distance. Furthermore, this project also includes the design, development and deployment of a cloud driven MAPF application. This application aims to provide an intuitive user experience to interact with the MAPF algorithm, visualise the traversal of the path finding solution and record statistical navigation parameters such as execution cost and execution time of the same. Subsequently, a navigation statistics pipeline is also established with the intention to produce strong predictive insights and perceptible navigation trends. Conclusively, this platform aims to aid organisations collectively monitor, control and measure the performance metrics of their fleet of agents effectively, to aid intelligent business decision making.

Figure 1: MAPF Application by Anusha Datta

2. Literature Review

The first step to embark on this project entailed gaining proficiency in previous research conducted in the domain of Multi-Agent Path Finding. This was achieved by studying a diversified set of related works and advancements in the field.

As mentioned above, a recent study [2] proposed Conflict Based Search (CBS), which is now one of the most successful MAPF algorithms. Following this research, multiple improvements have since been proposed such as CBS-D*-lite [3]. CBS-D*-lite is built upon CBS and avoids re-planning for agents that are not affected by the environmental changes. To achieve this, CBS-D*-lite employs D*-lite, an incremental single-agent pathfinding algorithm as the lower-level search method in CBS. On the same note, another instance of improvement is Multi-Objective CBS, or MOCBS[4], which accounts for optimisation of multiple objectives concurrently for each agent. Additionally, another example of an optimised variant is Meta-Agent CBS [5], or MA-CBS, which merges small groups of agents into meta-agents when beneficial. It may be noted that these improvements are just select examples of the numerous CBS variants at present.

However, despite extensive investigation, these previous works do not explore the effects on search performance upon introduction of varying lower level search heuristics. Hence, further research in this context is a key focus of this project.

2.1 A* Search Algorithm

A* Search is a state of the art algorithm for shortest pathfinding and graph traversal problems [6]. It is commonly utilized for a diverse set of applications such as video games, satellite navigation systems, or even informational search during online learning [7]. However, the application specific problem statement goal is to derive the least cost collision free path from a given start position to a given goal position in an obstacle encompassing grid maze. In this context, the least cost path is defined as the shortest path in terms of maze distance.

A* Search accomplishes this by maintaining a tree of paths originating at the start node, which it expands one edge at a time until its termination criterion is satisfied. This edge expansion, or exploration of state space, is performed by generating successors of previously explored states. In our case, that refers to the collective set of unvisited neighbours of each visited grid cell. Effectively, that results in two mutually exclusive sets of nodes- the Explored Set consisting of visited nodes, and the Frontier Set consisting of candidate notes for expansion.

At each iteration of the main loop, A* is required to determine which of its paths to extend. This decision is made based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Quantitatively, this is defined as the minimisation of the Evaluation Function *f(x)*, which is the sum of the Cost Function *g(x)* and the Heuristic Function *h(x)* as described below:

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

where

***x** is the current node on the path,*
***f(x)** is the A* evaluation function,*
***g(x)** is the cost function that computes the path cost from the start node to x,*
***h(x)** is a heuristic function that estimates path cost from x to the goal node.*

Figure 2: A Search Cost Evaluation Function [6]*

The termination criteria for A* Search consists of two primary cases. The first case is when the path it chooses to expand is a solution path from start to goal position. On the other hand, the second case arises if there are no paths eligible for expansion. Encountering either of these two cases will result in termination of program execution, with only the first termination case resulting in a path solution. The details of this program flow, including data structures and implementation nuances, may be observed via the detailed pseudocode below:

Algorithm 1: A* Algorithm

```
Input: graph
Input: startNode
Input: targetNode
Output: Path
for node in graph do
    node score := Inf;
    node.heuristicScore := Inf;
    node.visited := false;
end
startNode.score := 0;
startNode.heuristicScore := 0;
while true do
    currentNode := nodeWithLowestScore(graph);
    currentNode.visited := true;
    for nextNode in currentNode.neighbors do
        if nextNode.visited == false then
            newScore := calculateScore(currentNode, nextNode);
            if newScore < nextNode.score then
                nextNode.score := newScore;
                nextNode.heuristicScore := newScore +
                  calculateHeuristicScore(nextNode, targetNode);
                nextNode.routeToNode := currentNode;
            end
        end
    end
    if currentNode == targetNode then
        return buildPath(targetNode);
    end
    if nodeWithLowestScore(graph).score == Inf then
        throw NoPathFound;
    end
end
```

Figure 3: A Search Pseudocode [7]*

A map structure is maintained throughout the search which records each node's references to its respective parents. These references can be used to reconstruct and recover the solution path at the end of the search.

Unlike its well-known counter-parts, A* Search uses an informed approach by accounting for heuristics costs. Uniform Cost Search (UCS) is a special case of A* Search that only considers path costs *g(n)*, effectively setting *h(x) = 0* for all nodes. On the other hand, Greedy Algorithms only consider heuristic costs *h(x)* during the search traversal. Hence, A* Search tends to perform comparatively better as it combines the benefits of each of the two above mentioned approaches. It achieves this by including two separate path finding functions in its algorithm, which take into account both the cost from the root node to the current node and the estimated path cost from the current node to the goal node [8].

Moreover, it may be noted that UCS is optimal and complete but inefficient. Conversely, Greedy Algorithms are neither optimal or complete, but offer high efficiency by reducing the search space considerably. As a result of their union, A* Search does not always produce the shortest path, and hence does not guarantee optimality or completeness. However, A* Search achieves efficiency through intelligent approximation using heuristics. This trade-off between computation time and solution optimality is highly useful for applications such as games or web based maps, wherein speed is essential and an approximate solution is satisfactory [9].

In some practical travel-routing systems, it is generally outperformed by algorithms which can pre-process the graph to attain better performance, as well as memory-bounded approaches; however, A* is still the best solution in many cases [6].

2.1.1 Algorithmic Complexity

Algorithmic Complexity provides an estimate of the computing resources required for an algorithm to handle input data of various sizes or values, particularly in terms of time and space (memory or storage). This is denoted by the Big-O Notation, which is a mathematical representation of algorithmic complexity as a function of the size of the input to an algorithm.

2.1.2 Time Complexity

Considering a graph, A* Search may be required to traverse all edges before reaching from source to destination cell. Hence, the worst case time complexity is O(|E|), where E is the number of edges in the graph. In an unbounded search space, the number of nodes expanded is exponential in the depth of the solution. As a result, this may also be denoted as $O(b^d)$, where b is the branching factor (average number of successors per state) and d is the depth of the search tree.

2.1.3 Space Complexity

A* Search stores all generated candidate nodes in memory. Thus, in the required auxiliary space in worst case is O(|V|), where V is the total number of vertices. This

may also be denoted as $O(b^d)$, where b is the branching factor and d is the depth of the search tree. In practice, this turns out to be the biggest drawback of A* Search, leading to the development of memory-bounded heuristic searches, such as Iterative deepening A*, memory bounded A*, and SMA* [6].

Worst Case Time Complexity	$O(\|E\|) = O(b^d)$
Worst Case Space Complexity	$O(\|V\|) = O(b^d)$
where **b** is branching factor, **d** is depth of search tree	

Figure 4: A Search Algorithmic Complexity [6]*

2.2 Search Heuristics

The heuristic function *h(x)* has a significant effect on the practical performance of A* Search [10]. The reason for this being that a good heuristic allows A* to prune away many of the b^d nodes that an uninformed search would expand. Hence, it may be concluded that strong heuristics are those with low effective branching factor (the optimal being b* = 1). On another note, A* Search requires its heuristics to be admissible, meaning that the heuristic does not overestimate the path cost from start to goal node. In other words, admissible heuristic functions need to estimate path costs lesser than or equal to the actual costs [13].

As mentioned above, the A* evaluation cost function is given by: $f(x) = g(x) + h(x)$. The cost from start node to current node, *g(x)*, is computed incrementally during the A* Search. However, the next matter of consideration is to determine how to estimate the costs from the current node to the goal node, *h(x).* There are two primary approaches for this – either computing exact costs or approximating the cost based on some heuristics.

The former method refers to Exact Heuristics , which is generally very time consuming and computationally expensive. This may be achieved by pre-computing the distance between each pair of cells before employing A* Search. In the case of no cell obstacles, then the exact value of *h(x)* can be simply derived by using the Euclidean Distance formula. However, that is not the case in this specific application so it is not applicable. On the other hand, the latter, Approximate Heuristics, often prove to be more computationally efficient [12]. Hence, for the purpose of our MAPF application, we focus on exploring and comparing the following Approximate Heuristics:

2.2.1 Manhattan Distance

Manhattan Distance, or also commonly referred to as Taxi Cab Geometry, is the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively.

```
manhattan_distance = abs(x2 - x1) + abs(y2 - y1)
```

Figure 5: Manhattan Distance Formula [12]

This heuristic is most appropriate when the agent is allowed to move in four directions (right, left, top, bottom). An illustration of the Manhattan Distance Heuristic may be observed below (assume the black dot as start cell and red dot as goal cell).

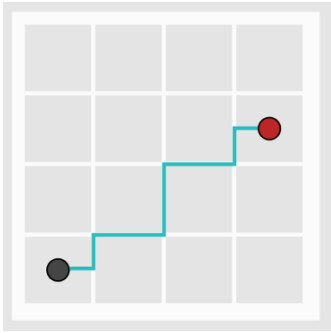


Figure 6: Manhattan Distance Heuristic

2.2.2 Chebyshev Distance

Chebyshev Distance, or also commonly referred to as Chessboard Distance, is the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively.

```
chebyshev_distance = max(abs(x2 - x1),abs(y2 - y1))
```

Figure 7: Chebyshev Distance Formula [11]

This heuristic is most appropriate when the agent is allowed to move in eight directions, similar to the allowable moves of a King in Chess. An illustration of the Chebyshev Distance Heuristic may be observed below (assume the black dot as start cell and red dot as goal cell).

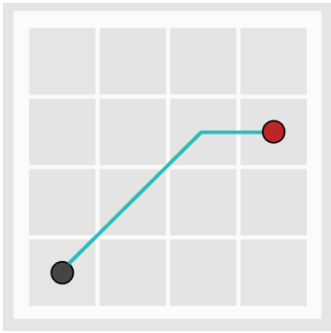


Figure 8: Chebyshev Distance Heuristic

2.2.3 Euclidian Distance

Euclidian Distance, or also commonly referred to as Straight Line Distance, is the distance between the current cell and the goal cell using the distance formula, which is essentially the Pythagorean Formula.

```
euclidian_distance = sqrt((x2 - x1)^2 + (y2 - y1)^2)
```

Figure 9: Euclidian Distance Formula [12]

This heuristic is most appropriate when the agent is allowed to move in any direction. An illustration of the Euclidian Distance Heuristic may be observed below (assume the black dot as start cell and red dot as goal cell).

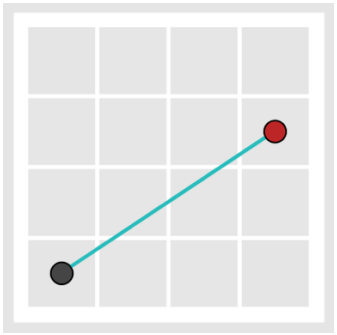


Figure 10: Euclidian Distance Heuristic

To conclude, a comprehensive overview of the heuristics and their cost distributions has been constructed in line with the heuristic definitions outlined above. This may be observed via the illustrations below (assume blue dot is agent, arrows indicate direction of movement and numeric labelling represents cost to cell pointed by arrow):

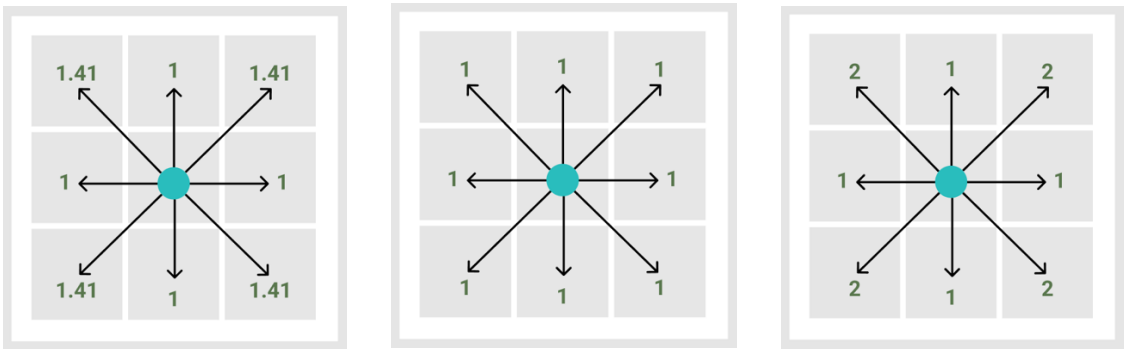


Figure 11: Heuristics Cost Comparison (L to R: Euclidian, Chebyshev, Manhattan)

2.3 Multi-Agent Path Finding

Formally, the Multi-Agent Path Finding problem is specified by a graph G = (V, E) and a set of k agents $\{a_1 \ldots a_k\}$, where agent a_i has start location $s_i \in V$ and goal location $g_i \in V$. Time is discretized into time steps, and agent a_i is in location s_i at time step t_0. Between successive time steps, each agent can either move to an adjacent empty location or wait in its current location. Both move and wait actions incur a cost of 1. A path for agent a_i is a sequence of move and wait actions that lead agent a_i from location s_i to location g_i. A conflict between two paths is a tuple (a_i , a_j, v, t_i), meaning that agents a_i and a_j both occupy the same vertex v at the same time step t. The objective is to find a conflict-free solution in the form of a solution set of k paths, one for each agent [14].

This objective may be achieved by using either a coupled or decoupled approach. A coupled approach is the optimal approach, as it considers all agents during solution formulation to produce a set of fastest collision free routes. On the other hand, a decoupled approach considers agents individually and finds optimal routes for each agent regardless of any potential collisions.

The decoupled approach is generally faster than the coupled approach, but since it does not consider all agents collectively it is non-optimal and may result in collisions. Not only does the decoupled approach not guarantee optimality, but it also does not guarantee completeness. Given the advantages of each approach, researchers were incentivised to combine optimality of the coupled approach and speed of the decoupled approach. Hence, this led to the formulation of Conflict Based Search.

2.4 Conflict Based Search Algorithm

Conflict Based Search (CBS) Algorithm is an approach to solve the Multi-Agent Path Finding problem [2]. CBS is a two-level algorithm.

- **Lower Level Algorithm**

  At the low level, a search is performed only for a single agent at a time, to produce the optimal route for each individual agent from start to goal node. All these solution paths are combined and passed to the higher level algorithm.

- **Higher Level Algorithm**

  The higher level algorithm performs a tree based search to identify conflicts between agents. It checks the collective solution paths for all agents to discover any conflicts and resolve them.

An overview of this Conflict Based Search Multi-Agent Path Finding program execution may be visualised through the program flow diagram show below:

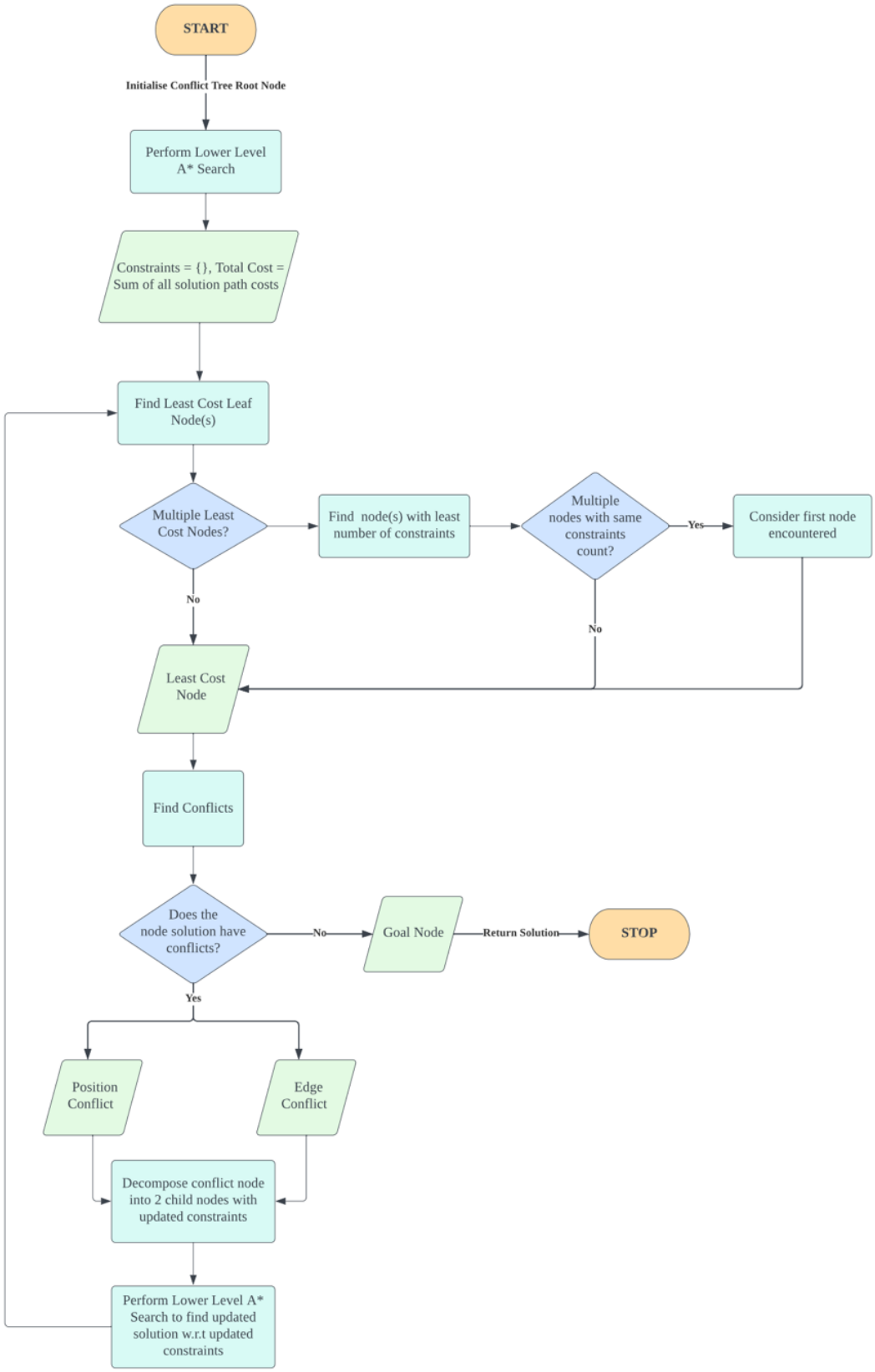


Figure 12: MAPF Program Flow Diagram [2]

The working of Conflict Based Search Multi-Agent Path Finding may be better understood through an example. Consider a 4x4 grid maze and 2 agents, Car 1 (Blue) and Car 2 (Green) respectively. The start and goal positions of these agents are denoted in the illustration above, with the goal house being the same colour as the agent car. Also it may be noted that the optimal routes from start to goal positions need to avoid the maze obstacles, in this case the hills, and must be collision free from other agents as well.

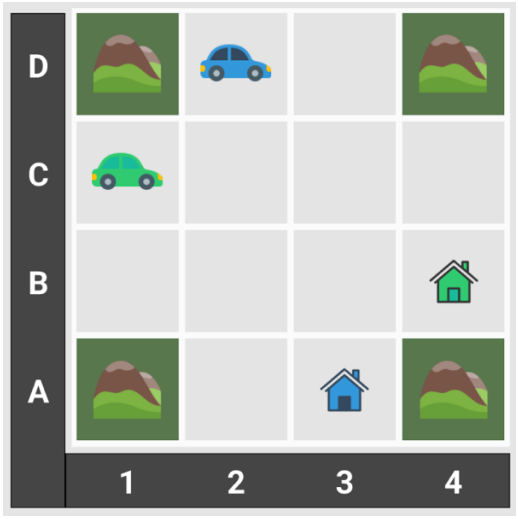


Figure 13: Conflict Based Search Maze

As the execution of the MAPF algorithm commences, a conflict tree is instantiated, as shown in Figure 14. The root node is initialised with no constraints {} and the solution is found using the low level search (A* Search in this case) to find optimal paths for each individual agent. The total cost is also computed by summing the cost of all paths in the solution set, with each action (move/wait) having a unit cost of 1. As observable, the root node has a collective cost of 8, as it is the sum of path costs of Car 1 and Car 2, which have a cost of 4 each. Once the root node is instantiated with all these details, it is passed up to the high level search algorithm for subsequent steps.

As mentioned above, the high level search checks the positions of all agents at each timestamp to detect conflicts. As such, upon analysing the root node, a conflict is detected at location B3 at timestamp 4. When a conflict is encountered, it will split the node into two child nodes, wherein each node contains a constraint for each of the agents involved in the conflict. As shown, the left child node has constraint {(1,B3,4)} which states that agent 1 cannot be at location B4 at timestamp 4. Conversely, the right child node has constraint {(2,B3,4)} which states the agent 2 cannot be at location B3 at timestamp 4.

As a next step, in an attempt to avoid conflicts in terms of agent collision, updated optimal routes are calculated for each agent with respect to all the constraints. Following this, the lowest cost leaf nodes of the tree are analysed by the high level search, which finds a conflict for node 2 at location B2 at timestamp 3. Similar to before, the conflict node is split into 2 child nodes each with an additional constraint for the conflicting agents. It may be noted that a child node will inherit all the constraints of its parent node, and simply append the new constraint to it.

This information of updated constraints is passed down to the low level search and updated optimal paths are generated with respect to both the previously inherited and newly added constraints. Next, the high level search analyses the lowest cost leaf node (either node 5 or node 3, both have least cost of 8 each). The algorithm uses a Conflict Avoidance Table (CAT), which states that if there are two least cost nodes, it will have a preference for the one with the least number of constraints. This ensures that the final solution returned in the end has the fewest constraints possible. Hence, it prefers node 3 to node 5 as it has 1 fewer constraint.

Upon analysing the preferred node 3, the high level search identifies a conflict, splits it into nodes 6 and 7, updates the constraints list and generates updated lower level search solutions accordingly, same as above. Next, the high level search looks to consider the lowest cost node. At this stage, that decision falls between node 5 and node 6. However, since they both have equal costs and equal number of constraints, the high level search simply considers the first node of the two.

The constraints of node 5's child nodes are such that the updated optimal solutions cannot be generated without having a wait action in them at locations D2 and C1 respectively. Next, the high level search considers node 6, which is at present the least cost leaf node, and splits it into child nodes 10 and 11.

At this stage, the least cost leaf nodes are node 4 and 7, both having an equal number of constraints. Hence, as per the reasoning outlined above, the high level search first processes node 4. Upon analysing node 4, no conflict is found, so it is returned as goal node containing optimal solution with a total cost of 9. This will conclude the algorithm execution.

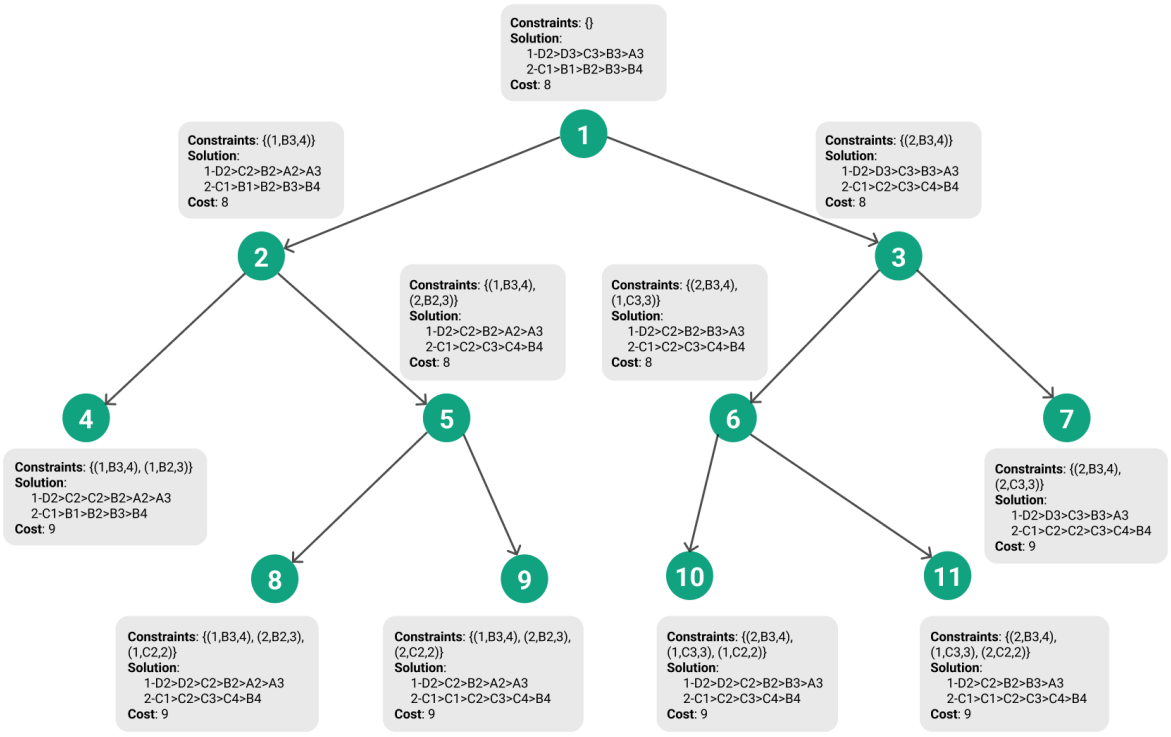


Figure 14: Conflict Based Search Tree

The high level pseudocode for the same is described below.

Algorithm 1: high-level of CBS

Input: MAPF instance
1 $R.constraints = \emptyset$
2 $R.solution$ = find individual paths using the low-level()
3 $R.cost = SIC(R.solution)$
4 insert R to OPEN
5 **while** OPEN *not empty* **do**
6 P ← best node from OPEN // *lowest solution cost*
7 Validate the paths in P until a conflict occurs.
8 **if** *P has no conflict* **then**
9 **return** P.solution // *P is goal*
10 C ← first conflict (a_i, a_j, v, t) in P
11 **foreach** *agent* a_i *in C* **do**
12 A ← new node
13 A.constriants ← P.constriants + (a_i, s, t)
14 A.solution ← P.solution.
15 Update A.solution by invoking low-level(a_i)
16 $A.cost = SIC(A.solution)$
17 Insert A to OPEN

Figure 15: Conflict Based Search Pseudocode [2]

2.4.1 Handling k-Agent Conflicts

There are two potential approaches to decomposing a conflict node, given a k-agent conflict is found for k > 2. In the first approach, k children can be generated, each of which add a constraint to k − 1 agents (i.e., each child allows only one agent to occupy the conflicting vertex v at time t). On the other hand, the second approach focuses on the first two agents that are found to conflict, and only branch according to their conflict. This leaves further conflicts for deeper levels of the tree.

As observable in the diagram below, the left side represents a k-way branching Conflict Tree (CT) for a problem that contains a 3-agent conflict at vertex v at time t. The right side tree presents a binary CT for the same problem. It may be noted that the size of the deepest layer in both trees is comparable. Hence, the complexity of both approaches is the same as they both will end up with k nodes each with k − 1 new constraints. In this context, we employ the second approach of handling k-Agent conflicts.

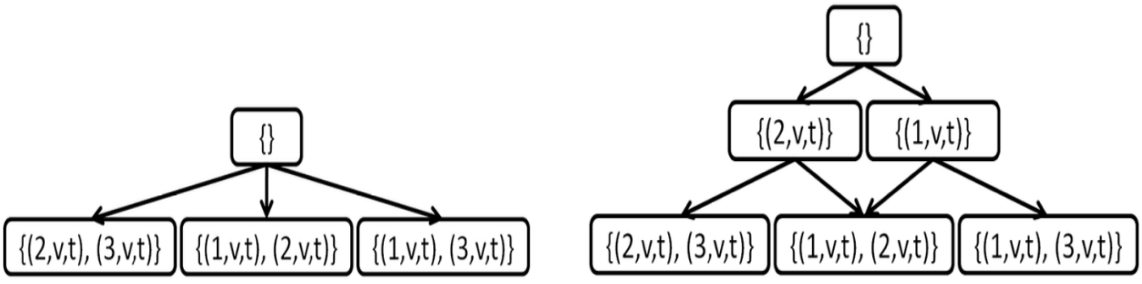


Figure 16: Conflict Tree Branching (L to R: k-way, binary) [2]

3. Project Objective & Scope

The primary objective of this project is to explore various lower level solver (A* Search) heuristics used within CBS for MAPF and deploy an interactive, intuitive graphical interface of the MAPF implementation powered by cloud services.

- Explore Conflict Based Search execution performances, through experimentation on the effect of various lower level search algorithm heuristics, such as Manhattan Distance, Chebyshev Distance and Euclidian Distance
- Design, develop and deploy a full stack MAPF application, which implements these algorithmic variations, to enable users to visualise the execution of the MAPF algorithm
- Set up a pipeline for navigation statistics collection, visualisation and analysis to facilitate intelligent data driven decisions
- Utilise cloud services for algorithmic computation, data storage and application deployment for a scalable implementation
- Implement a user authentication system to facilitate a personalised experience for users interacting with the MAPF application

4. Software Development Life Cycle

The Software Development Lifecycle (SDLC) adopted for the purpose of this project is the Waterfall Model [15]. According to this model, the SDLC phases are conducted in a precisely sequential manner as follows:

4.1 Requirements Elicitation

This phase entails enumerating and capturing all functional and non-functional requirements of the application.

4.2 System Design

This phase entails studying the system requirements obtained previously to facilitate system design and craft a comprehensive system architecture.

4.3 Implementation

This phase entails development of the application aligned with the system requirements and system design architecture determined in the previous phases.

4.4 Verification & Validation

This phase entails thorough software testing in form of verification and validation following the successful implementation of the application.

4.5 Maintenance

This phase entails the process of changing, modifying, and updating software to keep pace with dynamic customer needs.

Following the SDLC guideline above, a comprehensive project schedule was created to track progress against the projected timeline. This helped further refine and reiterate the project scope and objectives for seamless development.

5. Requirements Elicitation

In accordance to the project objectives described above, the following functional and non-functional requirements have been derived:

5.1 Functional Requirements

5.1.1 Authentication

User Authentication is essential to facilitate a personalised user experience and enforce features of accountability and traceability.

- User must be able to sign up on the platform
- User must be able to sign in/out of the platform with user ID and password
- User must be able to change password
- User must be able to retrieve access to account in case of forgotten credentials
- User must be able to delete account
    - Right to Erasure: Under Article 17 of the United Kingdom General Data Protection Regulation (GDPR) individuals have the right to have personal data erased [16]
- User must be able to access all data records associated with his/her account only
- User must not be able to access any MAPF application features until successfully authenticated

5.1.2 Maze Generation

Maze layout generation features constitute a core element of the MAPF application.

- User must be able to generate random mazes directed by the following maze generation algorithms:
    - Randomised Verticals
    - Randomised Horizontals
    - Recursive Division

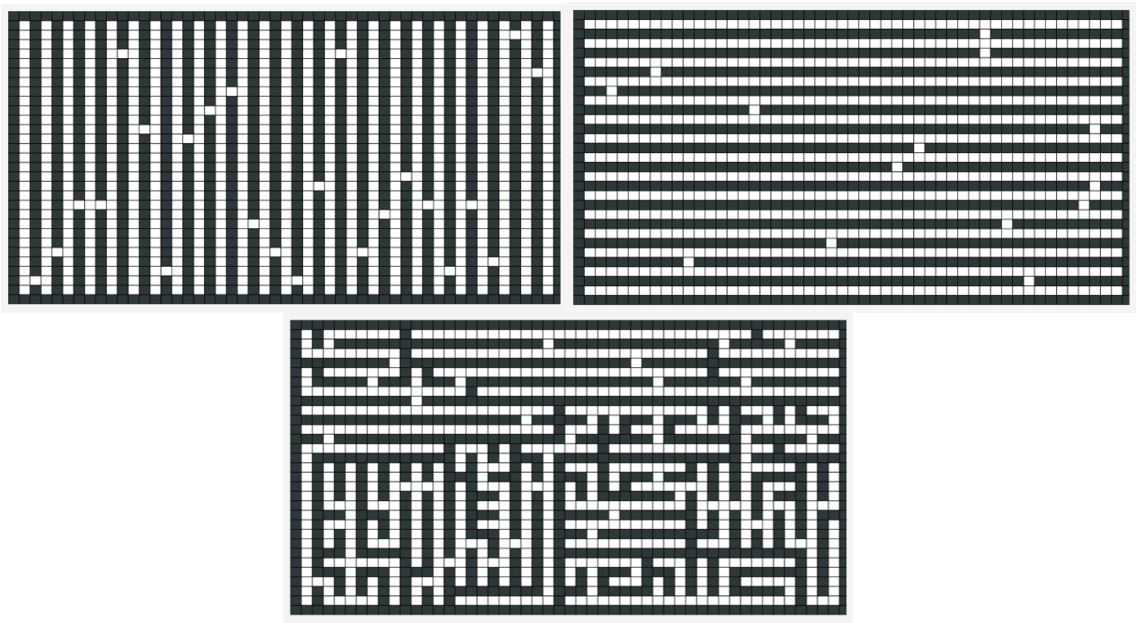


Figure 17: Maze Generation Algorithms (L to R: Randomised Verticals, Randomised Horizontals, Recursive Division)

- User must be able to create and save new maze layouts
- User must be able to load saved maze layouts
- User must be able to edit the maze layout by toggling obstacle
- User must be able to delete saved maze layouts
- User must be able to see the changes reflected immediately upon performing any maze specific actions
- User must be able to clear maze animation and retain maze layout
- User must be able to reset maze animation and maze layout

5.1.3 Maze DB Management

Maze DB is a remote cloud datastore that maintains all maze layouts attributed to an authenticated user.

- User must be able to create new maze record with maze name
    - User must receive a confirmation dialog box for creation
- User must be able to load saved maze by name
- User must be able to update an existing maze by name
    - User must receive a confirmation dialog box for updation
- User must be able to delete a saved maze by name
    - User must receive a confirmation dialog box for deletion

5.1.4 Multi-Agent Path Finding

For the purpose of this application, MAPF is solved using a Conflict Based Search algorithm with lower level solver as A* Search, which uses differing heuristics.

- User must be able to specify the chosen lower level solver heuristics

- User must be able to specify number of agents
- User must be able to specify the start and goal positions of each agent on the maze
- User must be able to obtain a complete solution, including the execution cost and execution time of the same
- User must be able to visualise the traversal of the solution on the maze layout in a perceptible manner

5.1.5 Navigation Statistics

A navigation statistics pipeline is established to collect MAPF statistical parameters such as execution cost and execution time, which are subsequently visualised to gain strong predictive insights and navigation trends.

- User must be able to visualise navigation statistics of all previous MAPF runs on every saved maze layout for all possible lower level heuristics
    - Scatter Plot: Execution Cost vs. Agent Count, for specified heuristics
    - Scatter Plot: Execution Time vs. Agent Count, for specified heuristics
    - Line Plot: Average Execution Cost vs. Agent Count, for all heuristics
    - Line Plot: Average Execution Time vs. Agent Count, for all heuristics
- User must be able to see changes reflected immediately in the navigation statistics upon performing additional MAPF runs

5.1.6 Application Hosting

The application must be hosted and deployed such that it is easily accessible to any user via Internet connection.

5.2 Non-Functional Requirements

5.2.1 Performance Requirements

The system must assume a strict upper bound limit of 30 seconds for the MAPF execution. On the other hand, all other application operations including authentication, maze generation, maze layout management and navigation statistics visualisation must assume a strict upper bound limit of 10 seconds.

5.2.2 Security Requirements

The system must protect user data such as user credentials, saved maze layouts and corresponding navigation statistics through end to end AES-256 Encryption.

5.2.3 Usability Requirements

To facilitate an accessible and intuitive application interface, the system must adhere to Schneiderman's 8 Golden Rules of interface design [17], key ones being as follows:

- **Reduce short term memory load**

  The implementation of a navigation bar to help users navigate seamlessly through the features.

- **Permit easy reversal of actions**

  Confirmation dialog boxes for all significant user actions to allow for straightforward reversal of actions.

- **Internal locus of control**

  Defined buttons and titles for each system feature to ensure navigation and task activation is clear for users.

- **Strive for consistency**

  Achieve consistent layout of application features through the use of synonymous fonts, colours and button themes.

6. System Design

6.1 System Architecture

The high level overview of the MAPF system design architecture, as aligned with the functional and non-functional requirements outlined above, is visualised below:

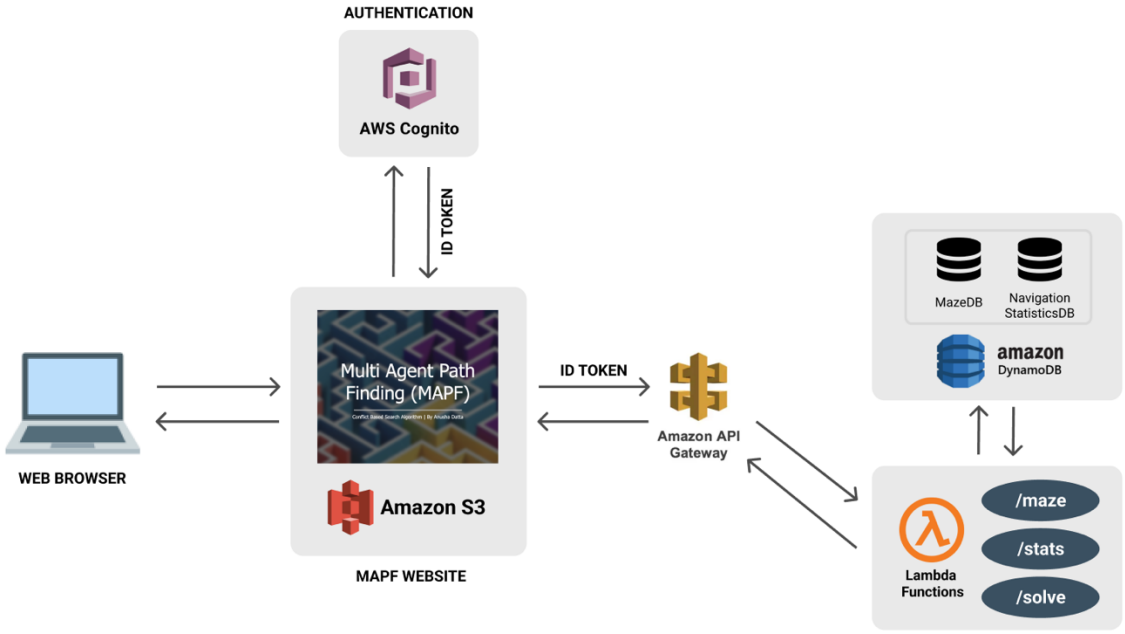


Figure 18: MAPF System Design Architecture

Note that all design choices henceforth enforce textbook principles of system modularity, low inter-module coupling and high intra-module cohesion. As shown, the web browser interacts with the MAPF website, which is hosted using Amazon S3. An Amazon S3 Bucket is a public cloud storage resource which provides object storage built to store and retrieve any amount of data from at any given instance. The advantage of this simple storage service is that it offers industry leading durability, availability, performance, security, and virtually unlimited scalability at very low costs.

As mentioned above, the user is required to log in, or sign up, before being provided access to the any features on the website. This authentication is implemented by the AWS Cognito, which provides a seamless solution to control access to AWS resources from the web application. This is achieved by returning temporary security credentials, specifically an ID Token, upon successful authentication. This ID Token is then used to access the application's backend resources in AWS or any service behind Amazon API Gateway.

All endpoint requests are routed through the Amazon API Gateway, which is an API management service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. This is beneficial as API gateways help to prevent malicious attacks by providing an additional layer of protection from attack vectors such as SQL Injection, or denial-of-service (DoS) attacks. They also enable support for mixing communication protocols.

The core application logic is implemented via AWS Lambda Functions. As visualised above, the architecture includes three primary endpoints - /maze, /stats and /solve. /maze and /stats aim to manage the database operations on MazeDB and NavigationStatsDB respectively, as outlined in the functional requirements above. On the other hand, /solve performs the fundamental MAPF execution to produce the required solution. As mentioned above, API invocations by the application will be processed by API Gateway and further routed to Lambda Functions for processing.

These Lambda Functions interface with AWS DynamoDB, which is a key-value NoSQL database service designed to run high-performance applications at any scale. Key benefits of DynamoDB include built-in security, continuous backups, automated multi-region replication, in-memory caching, and data export tools. As observable, the two databases maintained by the application are MazeDB and NavigationStatsDB. Further design details are described in the following sections.

6.2 Modified A* Search

The conflicts identified by the high level search, as described above, may be classified as one of two types – a Position Conflict or an Edge Conflict. These conflicts are identified and appropriately transformed to constraints by the higher level search, as detailed below.

A Position Conflict occurs when two agents end up at the same grid maze coordinates at the same timestamp. The constraint for this conflict simply takes the form of (Agent X, Position Y, Timestamp Z) to specify that Agent X cannot be at Position Y at Timestamp Z. On the other hand, an Edge Conflict occurs when two agents find

themselves traversing the same edge in the grid maze, or interchanging positions, on the same timestamp. The constraint for this conflict simply takes the form of (Agent X, [Position A, Position B], Timestamp Z) to specify that Agent X cannot traverse the edge between Position A and Position B at Timestamp Z. It may be noted that theoretically another approach is to enforce the edge constraint via two position constraints. However, this would result in an overly strong constraint for this case, as an agent may arrive at a specific position via a different path. Hence two different constraints are designed for these two different cases of conflict [2].

Subsequently, the lower level search enforces these constraints to find an updated optimal solution. This is achieved by modifying the algorithm to first filter the valid neighbours before evaluating the best action based on the evaluation cost function. Invalid neighbours are either obstacles, or unallowable positions due to the constraints list. Hence, at each iteration every neighbour shall be checked against every constraint to determine its validity for consideration. This ensures complete enforcement of constraints to generate an appropriate solution.

6.3 Database Schema

Remote data storage capabilities are central to the cloud driven implementation of the MAPF application. Consequently, there is a strong design requirement for a robust database schema that can efficiently facilitate the management, creation and retrieval of information from the cloud datastores.

A well designed database schema is critical to prevent potential scenarios of data redundancy, data inconsistencies, table join losses or insertion/updation/deletion anomalies. Hence, this section outlines a database schema design consistent with both the application's functional requirements and the mitigation of these concerns.

6.3.1 Entity Relationship (ER) Design

The conceptual design for the database has been visualised using an Entity Relationship (ER) diagram [18]. An ER model describes the relationships of distinct entities within a database schema. Refer to Appendix [A.1] for the ER model design element notation utilised.

In accordance with the ER elements described above, the following Entity Relationship diagram is constructed. As shown, we have the entity sets Users, Mazes and NavigationStatistics. Every navigation statistics record must belong to exactly one user and exactly one maze, both of which it requires to uniquely identify itself. Hence, the *NavigationStatistics* entity set is a weak entity set with *Users* and *Mazes* as the supporting entity sets.

Additionally, the relationships between the entity sets can also be clearly deciphered. As observable, the *Users* create the *Mazes*, following which the *NavigationStatistics* are produced by the *Mazes* and accessed by the *Users*.

As a navigation statistics record belongs to exactly one user, and a user may have multiple such records, there is a many-to-one relationship between *NavigationStatistics* and *Users*. Similarly, as a navigation statistics record belongs to exactly one maze, and a maze may have multiple such records, there is also a many-to-one relationship between *NavigationStatistics* and *Mazes.*

Finally, we may also discern the attributes and key attributes of each entity set from the diagram below. *Users* has attributes *user_id* and *user_credentials*, with *user_id* as the key attribute. Similarly, *Mazes* has attributes *maze_id, maze_name* and *maze_string*, with *maze_id* as the key attribute.

It may be noted that since *NavigationStatistics* is a weak entity set, its key attributes will be the union set of its own key attributes as well as the key attributes of its supporting entities. Hence, NavigationStatistics has attributes *run_timestamp, agents_count, execution_time, execution_cost* and *lower_level_solver*, with key attributes being *run_timestamp, user_id* and *maze_id.*

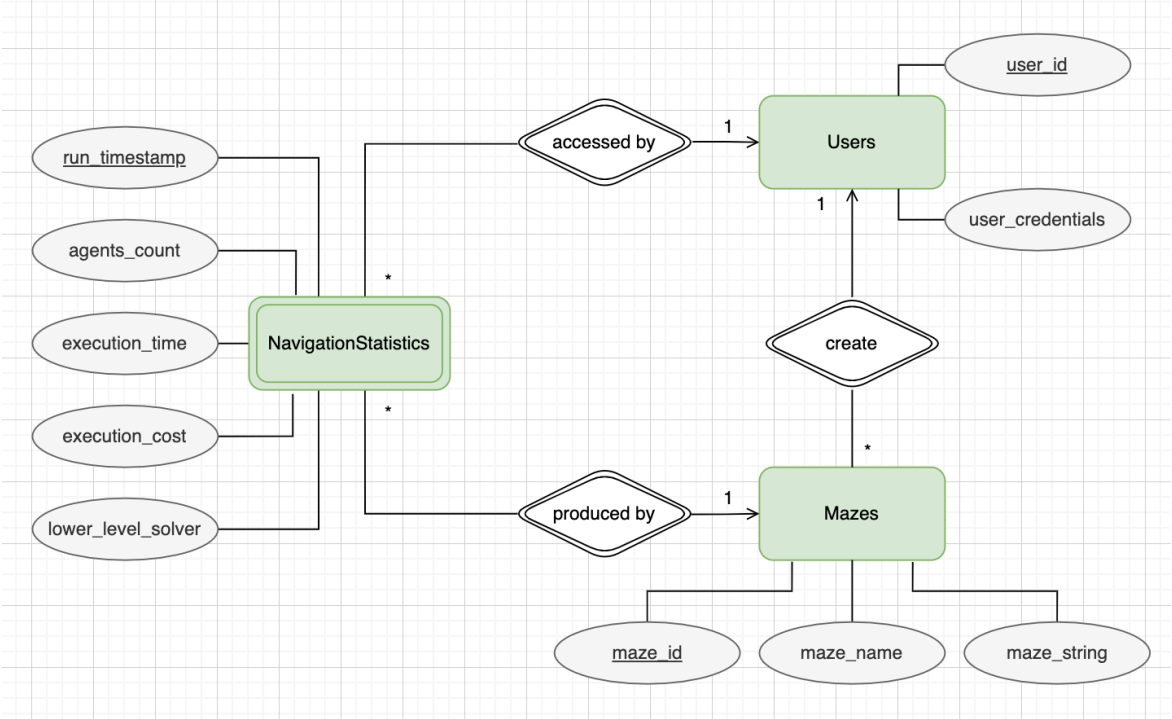


Figure 19: Entity Relationship Diagram

6.3.2 Relational Schemas and BCNF Normalisation

The next step of the design process is to transform the Entity Relationship Model obtained to a relational schema, or in this case multiple schemas for a set of tables. A relational schema is a table definition which comprises of the table name and its attributes.

This set of relational schemas must be further processed by undergoing Normalisation. Database Normalisation is imperative, as it helps mitigate data integrity concerns mentioned above such as data redundancy, data inconsistencies, table join losses or insertion/updation/deletion anomalies. Here, the choice method of normalisation is Boyce Codd Normal Form Decomposition, as outlined in Appendix [A.2]. This normalisation process may be augmented with FD transformation using Armstrong Axioms given in Appendix [A.3]. Following the BCNF steps, we derive the Functional Dependencies as given below. Note that in interest of eliminating a weak entity set and enforcing BCNF normalisation, the attribute of run_id has been introduced to the NavigationStatistics relation.

user_id → user_credentials
maze_id → user_id, maze_name, maze_string
run_id → user_id, maze_id, agents_count, execution_time, execution_cost, lower_level_solver, run_timestamp

Finally, we transform these BCNF decomposed functional dependencies into relational schemas, which shall be consequently utilised to construct the final database schema.

Users(user_id, user_credentials)
Mazes(maze_id, user_id, maze_name, maze_string) where *user_id* is a foreign key referencing *Users*
NavigationStatistics(run_id, user_id, maze_id, agents_count, execution_time, execution_cost, lower_level_solver, run_timestamp) where *user_id* and *maze_id* are foreign keys referencing *Users* and *Mazes* respectively

Figure 20: BCNF Decomposed Relational Schemas

6.3.3 Database Schema

A database schema is a set of relational schemas. As detailed above, we derived a set of BCNF normalised relational schemas guided by the Entity Relationship model created preceding that step. Hence, utilising this set of relational schemas we derive the final database schema as illustrated below.

As observable, there are three relations *Users, Mazes* and *NavigationStatistics* with primary *keys user_id, maze_id* and *run_id* respectively. Moreover, the many (*) to one (1) referential integrity enforced may be noted through the highlighted foreign keys.

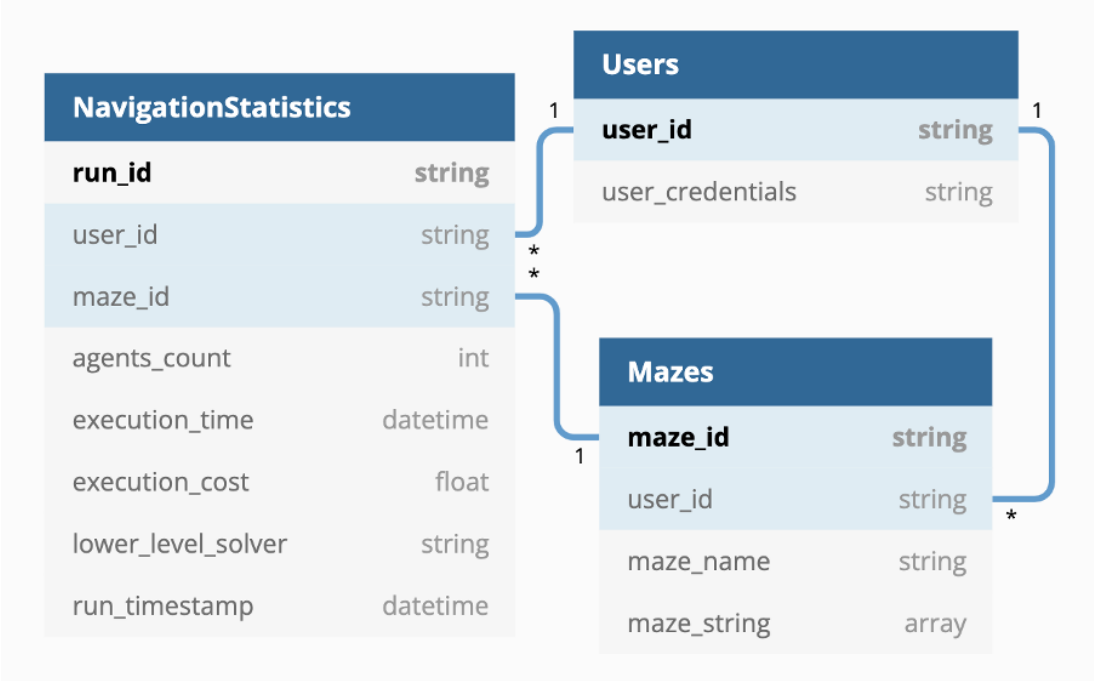


Figure 21: MAPF Database Schema

7. Implementation

Following the requirements elicitation and design stages, the next stage of the Waterfall Model SDLC is application implementation. Hence, this section outlines implementation specific details, in line with the comprehensive MAPF application requirements and subsequent system design summarised above.

7.1 Cloud Hosting: AWS S3

Application hosting may be achieved by either Web Hosting or Cloud Hosting [19]. For Web Hosting, the application is hosted on a personal (or organizational) computer or server. Whereas, in Cloud Hosting, the application is hosted on multiple interconnected web servers. Hence the data is stored and gathered from different servers located in different data centres, which may be in various locations.

As opposed to Web Hosting, Cloud Hosting is more scalable due to the servers interconnectivity of the network the application is hosted on. Furthermore, it is more cost effective as the pricing for cloud hosting is determined by the usage. This is beneficial for applications with fluctuating traffic trends. It is also more reliable as the downtime of any server can be easily compensated by another server on the same network. Overall, Cloud Hosting is well known for dynamic, demand-dependant scalable structure. Hence, for the purpose of this application, we opt in for a Cloud Hosting solution.

In the interest of maintaining a homogenous AWS cloud infrastructure ecosystem [20], AWS Simple Storage Service, or more commonly AWS S3, is chosen. In the context

of this project, it has been used for static file hosting of the MAPF web application, which is achieved by uploading all relevant HTML/CSS/JavaScript files to an AWS S3 bucket.

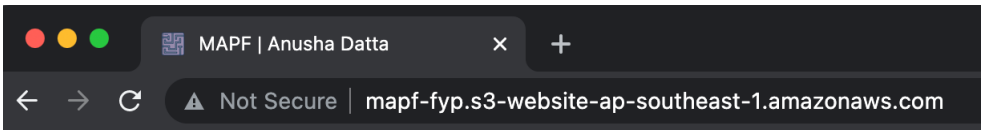


Figure 22: MAPF Cloud Hosting by AWS S3

7.2 Front End Technologies

After much deliberation, a conscious decision was made to abstain from utilizing a programming framework, such as React.js or AngularJS, in the interest of avoiding over-engineering. As a result, the front end implementation of this web application is in vanilla JavaScript, which is suitably compatible with the chosen visualisation library, p5js.

7.2.1 Visualization: p5js

p5js is a free, open source JavaScript library used for creative coding and visualisation. All versions of p5.js are stored in a CDN (Content Delivery Network) [21]. Hence, here we access a hosted version of the p5js library, by adding a CDN link in the source code.

This library is utilised to implement the key elements of the MAPF application, including the maze generation, maze updation, agent placements and path finding visualisations. The grid maze dimensions are defined as 30x50 units, with each unit being a grid maze cell. This implementation can be easily altered by modifying the parameters as shown below. Note that the defined parameters, as shown in Figure 23, are 2 units greater than those stated above. The reasoning for this is that the grid maze has an uninterrupted obstacle periphery which serves as the maze boundary. Hence this consumes the 2 stated units, which are not considered upon calculation of the actual grid maze's dimensions.

```
const HEIGHT = 500; //pixels
const WIDTH = 950; //pixels
const mazeHeight = 32; //units
const mazeWidth = 52; //units
const mazeHeightUnit = HEIGHT / mazeHeight; // pixels/unit
const mazeWidthUnit = WIDTH / mazeWidth; // pixels/unit
```

Figure 23: Grid Maze Dimensions

The grid maze is instantiated by invoking the p5js built in method setup(). This method executes the initial sketch of the grid maze, which is observable upon first accessing the MAPF application.

```
p5.setup = function () {
    p5.createCanvas(WIDTH, HEIGHT);
    p5.background(220);
    initGrid(p5);
};
```

Figure 24: p5js Set Up

Moreover, p5js also offers the built in method touchStarted(), which aids detection of user interactions with the grid maze. This enables the user to edit the maze by adding or removing obstacles, or altering agent start and goal positions.

```
// p5js inbuilt method to detect screen touches
p5.touchStarted = () => {
    place = onTouchFunctions[onClickFlag];
};
```

Figure 25: p5js User Click Detection

On the event of a user click, the maze will be altered by invoking another p5js built in method colourBox(p5, index, colour), which takes parameters of p5js object, the coordinates of the grid maze cell to modify and the colour the cell must be set to. This colour may be specified as either a Hex Colour Code or an RGB value. This built in method may be used for both obstacles and agents, as passing the appropriate colour shall modify the cell appropriately. This MAPF application accounts for 15 agents, each of which has start and goal positions denoted by darker and lighter gradients of the same colour. For instance, Agent 1's start position is deep crimson red (#990000), while its goal position is red (#CC0000).

This maze editing functionality is executed via a flag system, which may be analogised as most similar to the Subscriber – Publisher architectural design pattern. This pattern provides a framework for exchanging messages between publishers and subscribers. This pattern involves the publisher and the subscriber relying on a message broker that relays messages from the publisher to the subscribers. In this context, we may assume user actions as the publisher, maze rendering functions as the subscribers and program flag variables as the message brokers. All flag variables are initialised as False. When a user action occurs, the flag is updated to True. On the subsequent render cycle, the corresponding maze rendering function is invoked, the completion of which concludes with resetting the flag to False. The program flags declared are:

- **Maze Flag:** Allows modification of maze cells for obstacles or agent positions
- **Visualise Solution Flag:** Allows modification of maze to visualise MAPF solution through agent traversals
- **Clear Flag**: Allows resetting maze by clearing it
- **Clear Animation Flag:** Allows resetting maze by clearing the agents and their traversals, but retaining the maze layout itself

Furthermore, it may be noted that p5js achieves its visualisation by maintaining a render cycle invoked nearly 60 times every second. This rapid refresh cycle makes it challenging to visualise the path traversal in a step wise manner. To mitigate this, a

custom delay function is designed to intercept the render cycle and produce a perceptible visualisation.

```
function delay(time) {
    return new Promise(resolve => setTimeout(resolve, time));
}
```

Figure 26: p5js Time Delay Function

7.2.2 Subsystems User Interface

This section showcases various subsystem user interfaces in the application, along with comprehensive descriptions detailing the features and functionalities of the same.

7.2.2.1 Authentication

Authentication is essential to enforce accountability and traceability of actions. Upon initial access to the MAPF application, users are required to authenticate themselves before being allowed to view or access any features of the application. As outlined above, AWS Cognito is utilised to achieve this functionality. Upon initial access to the MAPF application, the user is redirected to AWS Cognito's hosted UI which implements the entire authentication flow of sign in, sign up and password retrieval. Moreover, in the interest of user account security, it also enforces complex password policy to ensure the password meets all strong password guidelines. These entails requirements that the password must contain at least 8 characters including lower case letter, upper case letter, special character and number.

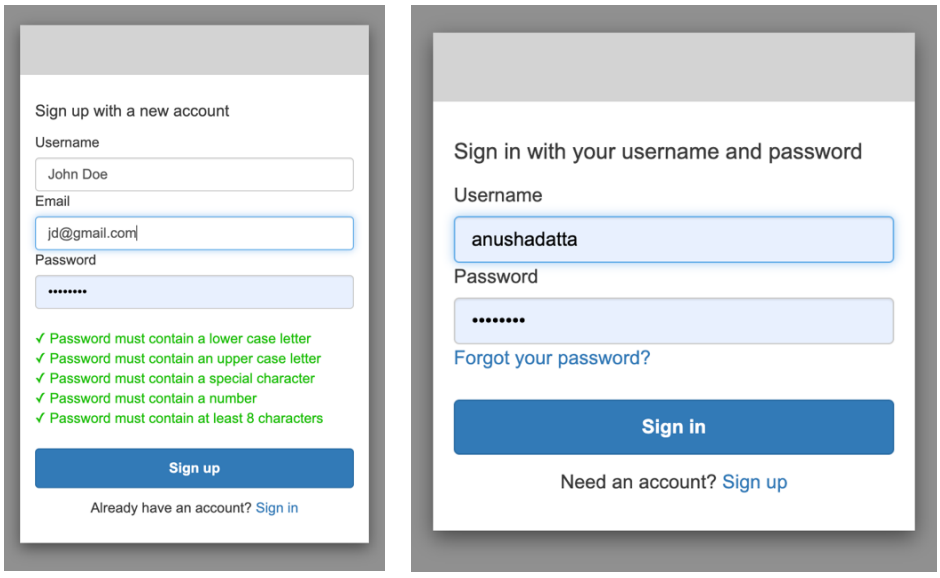


Figure 27: AWS Cognito (L to R: Sign Up, Log In)

Furthermore, there is an additional provision that user credentials may be saved by the web browser. This enhances the user experience by providing a convenient sign in route over multiple uses of application.

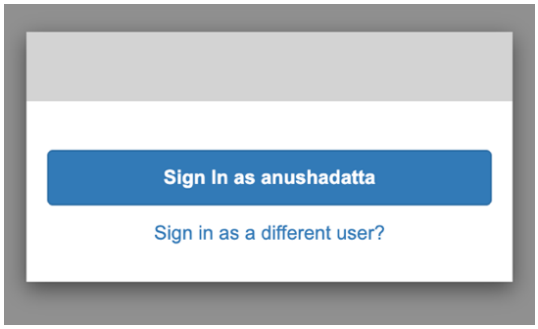


Figure 28: AWS Cognito Caching User Credentials

7.2.2.2 Maze Management

The Maze Management section allows the user to create a new maze, load a saved maze, edit the maze and delete the maze. Creating a new maze entails saving the grid maze layout created to the cloud datastore under a chosen name. This name will then be visible in the load saved maze dropdown, which can be selected to retrieve that maze from the database and display it. Furthermore, the maze generation algorithms randomised verticals, randomised horizontals and recursive division are also provided to the user, aimed to aid the maze layout creation process. These generated mazes, or even any new or loaded mazes, may be edited with the Toggle Obstacle option. When selected, this button option allows the user to modify the maze cells by simply clicking on them, such that an obstacle cell is turned to non-obstacle cell and vice versa. Moreover, Clear Animation allows resetting maze by clearing the agents and their traversals, but retaining the maze layout itself. This allows the user to reselect the agents positions and rerun the MAPF Algorithm on the same grid maze. While Reset Maze allows resetting the maze by clearing the entire maze itself. All these changes to a saved maze may be uploaded to the cloud datastore by selecting the Update Maze option, which updates the maze selected in the Load Saved Maze dropdown menu. Moreover, this selected maze may also be deleted from the cloud datastore by selecting the Delete Maze option.

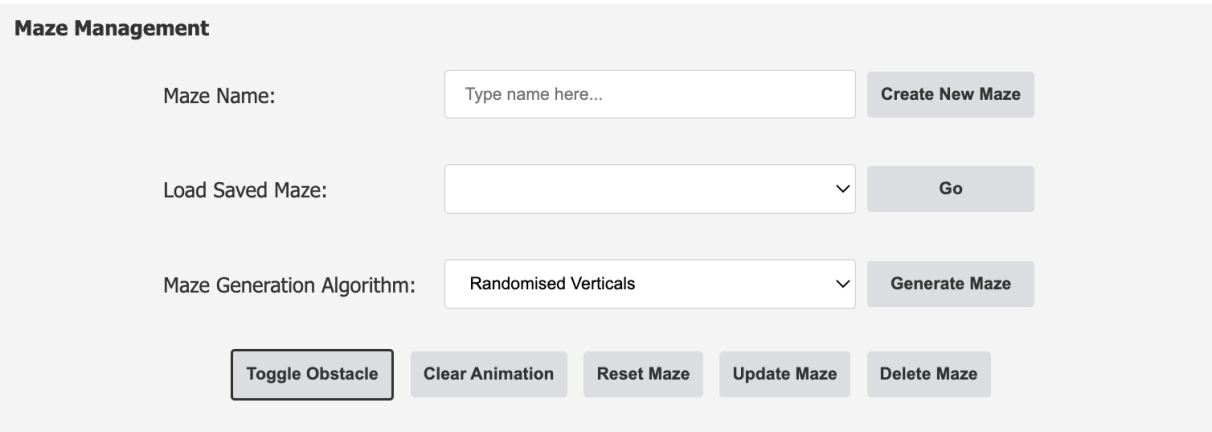


Figure 29: Maze Management User Interface

Additionally, this set of user interactions have been supplemented with informative dialog boxes to yield closure of action or confirmatory dialog boxes to ensure easy reversal of actions. These implementations are in accordance with the design choices guided by Schneiderman's 8 Golden Rules [17] of interface design.

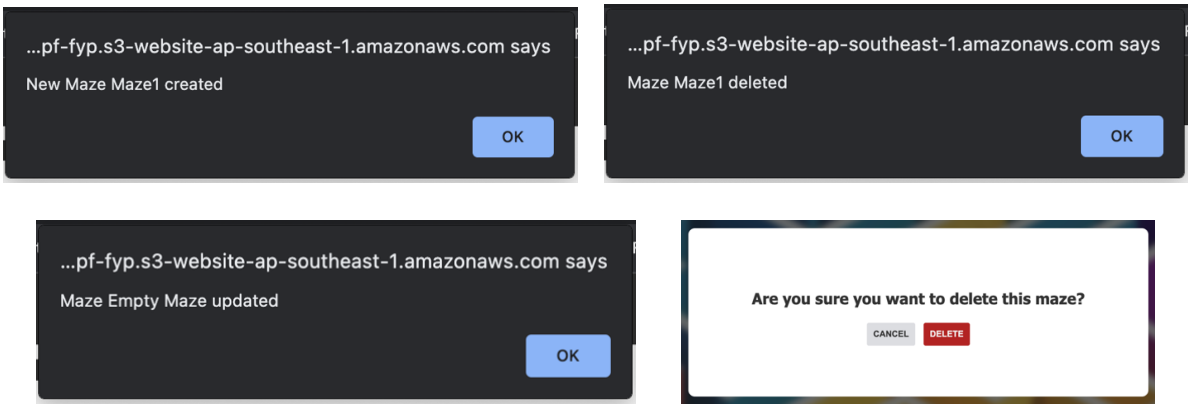


Figure 30: Informative/Confirmatory Dialog Boxes

7.2.2.3 MAPF Implementation

Following the Maze Generation is the set up for the MAPF Variables. This includes specifying the A* Search Heuristic, number of agents and their respective start and goal positions on the grid maze. The choice of heuristics include No Heuristic, Manhattan Distance, Chebyshev Distance and Euclidian Distance. Furthermore, the agents placement dropdown is dynamically updated based on the selection made in the agents count dropdown. Hence, this drop down is initialised as empty until the user selects a valid agent count. Once this dropdown is populated, the agent start and goal positions may be placed on the grid maze denoted by darker and lighter gradients of the same colour, as described above.

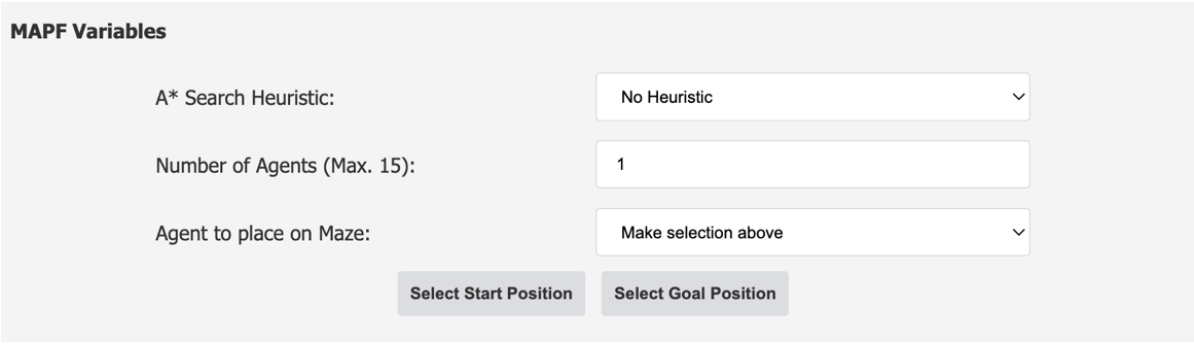


Figure 31: Multi-Agent Path Finding Interface

The complete set of these user interactions have been subjected to input validation and error checking. For instance, the agents count dropdown has inbuilt input validation to ensure the input is an integer between 1 and 15. Any value lower than 1 or greater than 15 is automatically set to 1 or 15 respectively. Moreover, any fractional value between 1 and 15 is appropriately rounded up or down to ensure the input is a

whole number. It may be noted that an additional inbuilt error checking has been implemented for this feature as well, disallowing the user to select multiple coinciding start or goal positions for the same agent. Moreover, it also disallows agent placements to coincide and requires all agent placements to be specified before the MAPF solve operation.

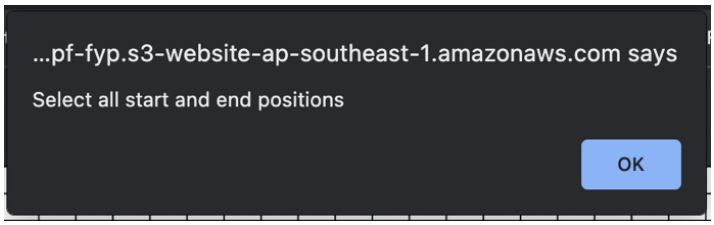


Figure 32: Agent Placement Validation

Following the specification of all MAPF variables, the MAPF solution set of collision free paths for all agents may be generated through the solve button. This will result in program execution of the MAPF Algorithm to return a solution, which is subsequently visualised on the grid maze as illustrated below. It may be noted that if an agent traverses a grid maze cell previously traversed by another agent, the path visualisation of the most recent agent will overwrite the previously traversed path.

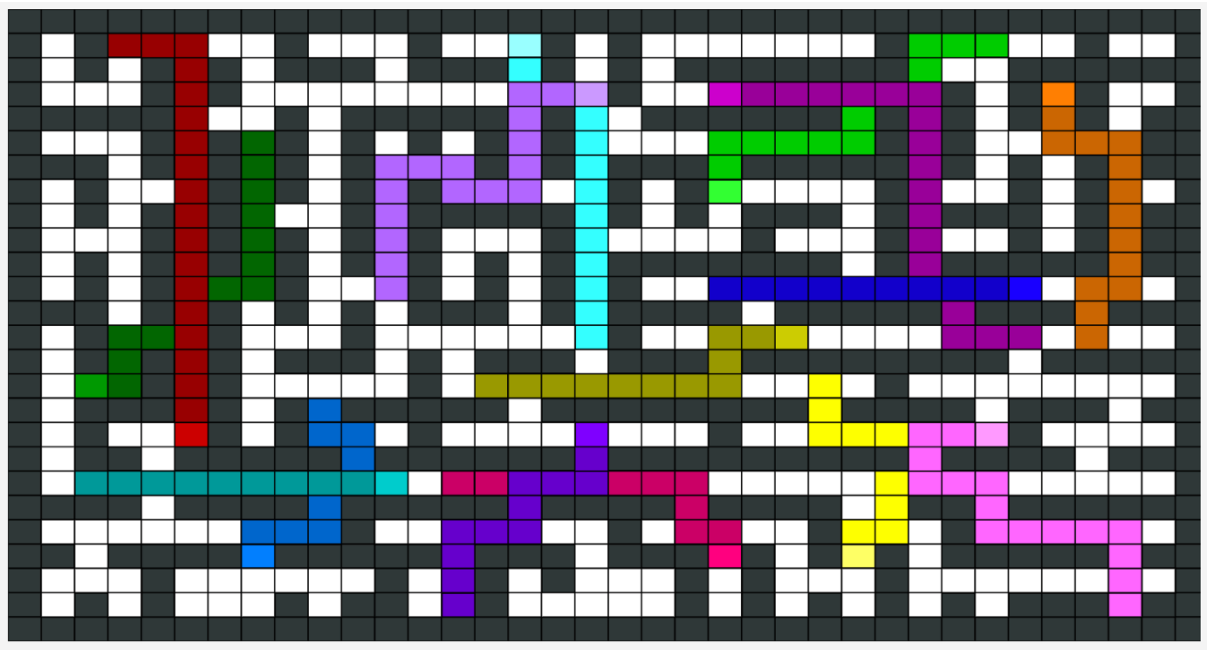


Figure 33: MAPF Maze Visualisation

Furthermore, the Execution Cost and Execution Time for the specific MAPF solution are also computed and displayed on the interface. The Execution Cost refers to the total cost of actions, move and wait, for the collective paths of all agents. While the Execution Time measures the total computation time for the MAPF algorithm to execute and produce the solution generated.

Figure 34: MAPF Solution Cost & Time

7.2.2.4 Navigation Statistics

As detailed above, the MAPF algorithm execution produces a set of collision free solutions for all agents, along with the navigation statistics of Execution Cost and Execution Time. It may be noted that these statistics vary across different maze layouts due to differing obstacle densities, agent counts and agent placement initialisations. The visualisations are essential to curate this data in a manner that highlights trends, distinguishes outliers and provides strong predictive insights to drive intelligent business decisions. Hence, to effectively visualise and interpret these statistical navigation parameters, we utilise features provided by the Chart.js library [22], which is an open-source JavaScript library for data visualization.

The first visualisation provides an overview of all MAPF execution runs for a maze and heuristic combination, which may be specified by the user. This generates a scatter plot that showcases execution cost and execution time against the agent count associated with an individual MAPF execution run. These discrete data points are indicated for all MAPF execution runs for the specified maze and heuristic selected. Through the use of a scatter plot, the user may discern trends based on the data point clusters and identify outliers as well.

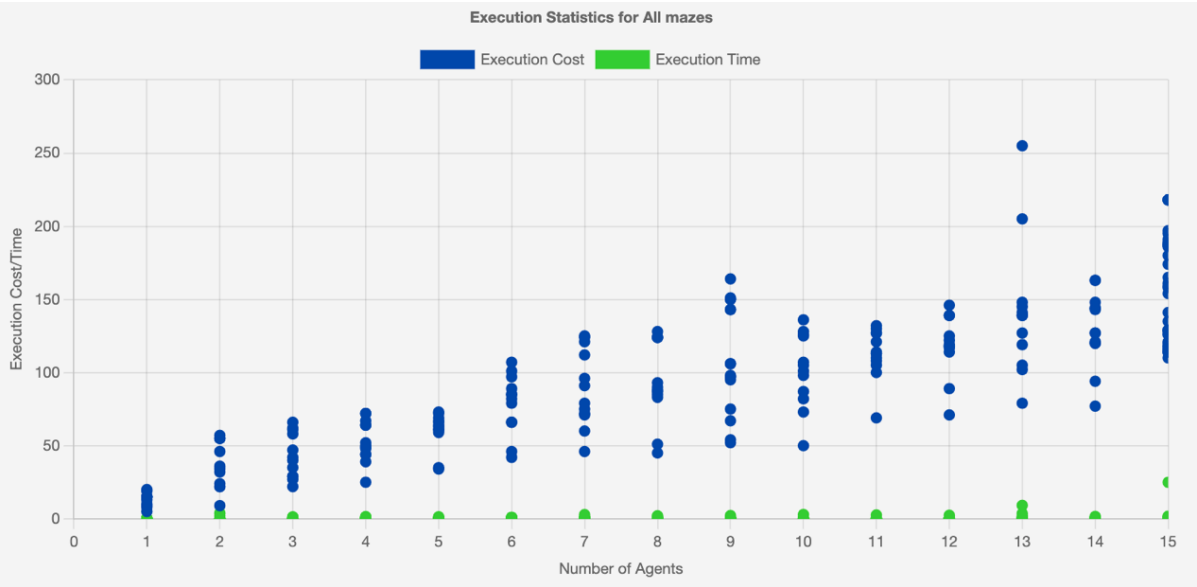


Figure 35: MAPF Execution Statistics Scatter Plot

The next visualization provides a deeper insight into the heuristic performances of the MAPF execution runs for a user specified maze. This is achieved by generating a line chart that showcases the average execution time over all MAPF execution runs against the agent count for every heuristic. In other words, each data point maps to an agent count and the average execution time for all execution runs for the corresponding agent count. Through the use of averages, the user may gauge individual heuristic performances and the related trends for the same.

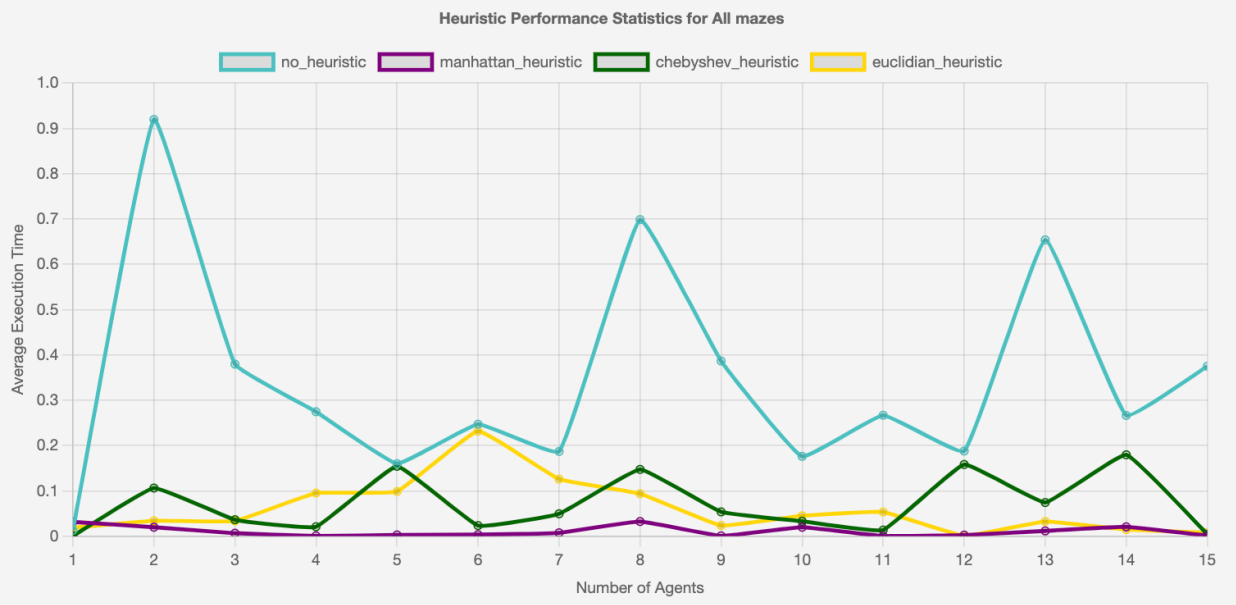


Figure 36: MAPF Execution Time Heuristic Performance Line Chart

Similar as above, a line chart is also generated for average execution cost against agent count, for a specific maze.

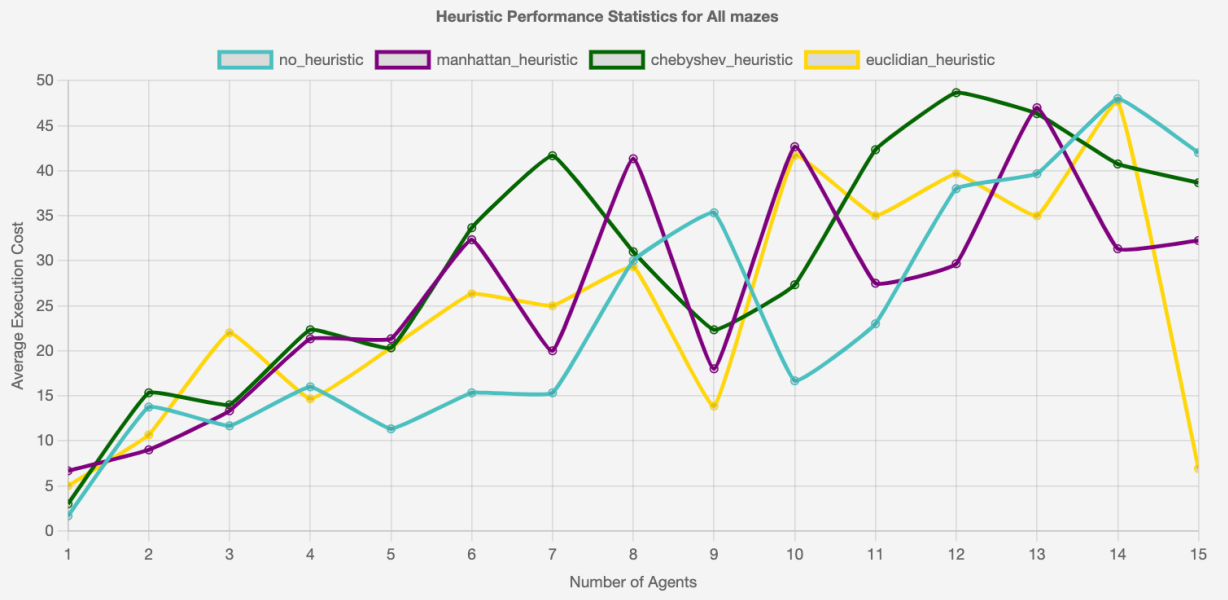


Figure 37: MAPF Execution Cost Heuristic Performance Line Chart

As highlighted previously, Conflict Based Search provides an optimal MAPF solution. A* Search also produces an optimal path given the heuristic function *h(x)* estimates a lower bound on the actual cost from the current node to the goal node. Since every distance heuristic here satisfies this requirement, these differing heuristics shall all produce optimal paths with identical execution costs for the same agent initialisations. As a result, the Execution Cost visualisations do not serve any insights on heuristic performances. Rather, these aim to provide an insight into the distribution of MAPF execution runs in terms of agent initialisations. MAPF execution runs with higher execution costs allude to distant initialisations of agent start and goal positions. Conversely, MAPF execution runs with lower execution costs imply congested initialisations with agent start and goal positions closely positioned. These inferences are valuable to ascertain navigation trends through common agent initialisation and traversal patterns.

Also note that each of these graphs have dynamic scales which adjust with the changing data points. This ensures effectual visualisations of the available data.

7.3 Back End Technologies

The back end implementation of this project is in Python3. As a versatile, efficient, and reliable programming language, Python advances many benefits. Not only is it syntactically simplified, but also offers a vastly diverse set of libraries and frameworks which can greatly aid development. It is also extensively used in fields of Artificial Intelligence, Automation, Big Data and Cloud Computing, which yields a significant corpus of support and resources. Moreover, it seamlessly interfaces with AWS through Boto3, which is the Python AWS SDK that provides a Python API for AWS infrastructure services.

7.3.1 AWS Serverless Application Model (SAM)

The AWS Serverless Application Model (SAM) is an open-source framework for building serverless applications, by providing shorthand syntax to express functions, APIs, databases, and event source mappings. To access SAM, we use AWS Command Line Interface (CLI) to create a serverless application, which is subsequently packaged and deployed in the AWS Cloud. Hence, as a first step we set up AWS CLI using the following shell commands:

```
$ curl "https://awscli.amazonaws.com/AWSCLIV2.pkg" -o "AWSCLIV2.pkg"
$ sudo installer -pkg AWSCLIV2.pkg -target /
```

Figure 38: AWS CLI Set Up

Next, we configure SAM using the following commands. Note that the build and deploy commands are used iteratively throughout the development of the project.

```
#Step 1 - Initialise application
sam init

#Step 2 - Build application
cd sam-app
sam build

#Step 3 - Deploy application
sam deploy --guided
```

Figure 39: SAM CLI Commands

AWS SAM consists of two key components, namely the AWS SAM template specification and AWS SAM command line interface. The SAM specification is used to define the serverless MAPF application, while the SAM CLI is used to build and deploy it. The specific details of SAM configuration are shown as follows.

samconfig.toml

```
version = 0.1
[default]
[default.deploy]
[default.deploy.parameters]
stack_name = "mapf"
s3_bucket = "aws-sam-cli-managed-default-samclisourcebucket-1dlt4sf3rwpnt"
s3_prefix = "mapf"
region = "ap-southeast-1"
confirm_changeset = true
capabilities = "CAPABILITY_IAM"
image_repositories = []
```

Figure 40: SAM Configuration

7.3.2 Authentication: AWS Cognito

AWS Cognito aids the abstraction of building and managing a backend solution to handle identity management, network state, storage, security and data synchronisation. This is beneficial as it allows a greater development focus on creating a seamless application experience for the user.

It is configured and set up through modification of the *template.yaml* file, with the following specifications.

```
ApiGateway:
  Type: AWS::Serverless::Api
  Properties:
    StageName: default
    Cors: "'*'"
    Name: MapfGateway
    Auth:
      DefaultAuthorizer: CognitoAuthorizer
      Authorizers:
        CognitoAuthorizer:
          UserPoolArn: arn:aws:cognito-idp:ap-southeast-1:590170059730:userpool/ap-southeast-1_kCpcekVL4
```

Figure 41: Authorisers Configuration for existing AWS Cognito Pool

As mentioned above, successful authentication results in the generation of an ID Token, which is processed by Authorisers to enable access to other services associated with AWS APIGateway. More specifically this token is a JSON Web Token (JWT), a Base64-encoded JSON string that contains user information claims [23]. A JWT consists of a header, payload and signature. The JWT signature is a hashed combination of the header and the payload, which must be decoded and validated for authentication. In the interest of advanced security, this token expires every 30 minutes, requiring the user to re-authenticate themselves in order to continue using the MAPF application.

7.3.3 Cloud Database: AWS DynamoDB

Amazon DynamoDB is employed to fulfil the database functional requirements as outlined above. It offers built-in security, continuous backups, automated multi-region replication, in-memory caching, data export tools and guaranteed reliability with a service level agreement of up to 99.999% availability. DynamoDB supports PartiQL [24], a SQL-compatible query language, to select, insert, update, and delete data in DynamoDB. It enables unified query access across multiple data stores and data formats by separating the syntax and semantics of a query from the underlying format of the data or the data store that is being accessed. Hence, it enables users to interact with data with or without regular schema.

7.3.3.1 AWS DynamoDB Set Up

The AWS DynamoDB is set up by modifying the *template.yaml* file as shown below, and subsequently deploying using Serverless Application Manager (SAM).

```
MazeTable:
  Type: AWS::Serverless::SimpleTable
  TableName: MazeTable
  Properties:
    PrimaryKey:
      Name: maze_id
      Type: String

RunStatisticsTable:
  Type: AWS::Serverless::SimpleTable
  TableName: StatisticsTable
  Properties:
    PrimaryKey:
      Name: run_id
      Type: String
```

Figure 42: AWS DynamoDB Set Up

This results in the creation of the Maze Table and the Run Statistics Table on AWS DynamoDB. These remote cloud datastores are populated with records in accordance

with the database schema outlined above. It may be noted that further optimisation is achievable by setting up indices to speed up the queries.

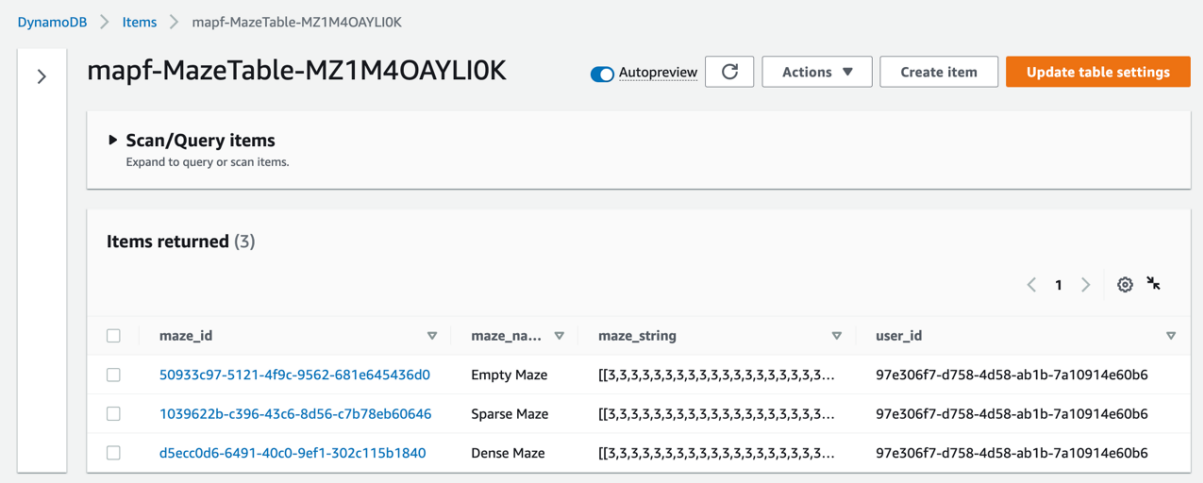


Figure 43: AWS DynamoDB Maze Table

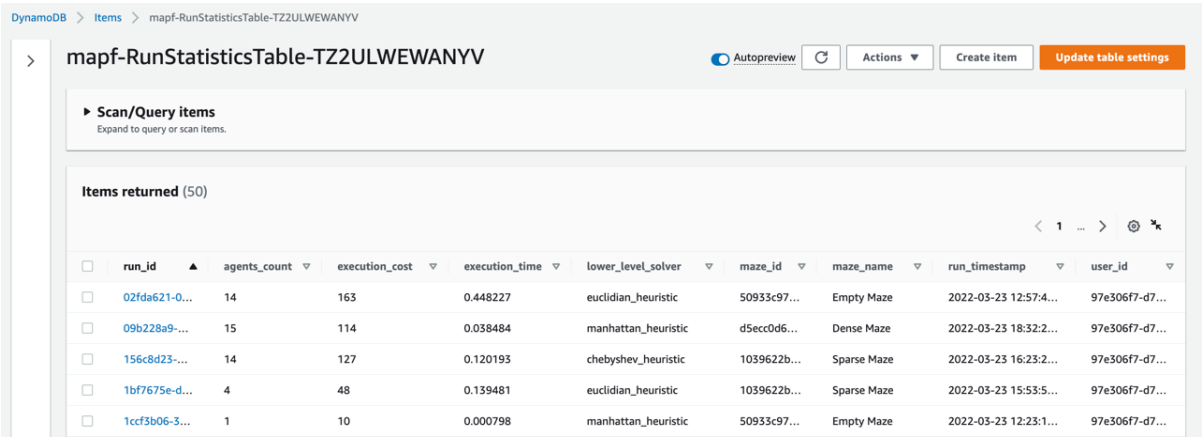


Figure 44: AWS DynamoDB Run Statistics Table

Another point to note is that whenever a maze item is deleted from the Maze Table, all corresponding run statistics for that maze shall also be deleted from the Run Statistics Table. This is to ensure referential integrity, as outlined in the database schema design above.

7.3.3.2 Principle of Least Privilege

Each service under AWS has a Permissions Model that specifies which actions an entity is authorised to perform. On that note, a Function Policy is created for table on AWS DynamoDB to set up appropriate permissions for all application services. This

helps enforce the principle of least privilege [25], which states that an entity should only be given those resources and permissions required by it to complete its task. In the context of this project it implies that every endpoint has access to its corresponding database table only.

```
Policies:
  - DynamoDBCrudPolicy:
      TableName: !Ref RunStatisticsTable
  - DynamoDBCrudPolicy:
      TableName: !Ref MazeTable
```

Figure 45: Enforcing Principle of Least Privilege

7.3.4 Application Programming Interface

The interactions between the AWS Cognito, API Gateway, Lambda Functions and Endpoints has been described in the system architecture section above. An overview of the endpoints created for this MAPF application is shows as follows.

Endpoint	HTTP Method	Action
/maze	GET	Retrieve all mazes created by authenticated user from Maze Table
/maze	POST	Create new maze under authenticated user from Maze Table
/maze	PUT	Update a maze record in Maze Table by modifying the maze layout
/maze	DELETE	Delete a maze in Maze Table
/stats	GET	Retrieve all navigation statistics from Run Statistics Table associated with all execution runs performed on user's mazes
/solve	POST	Perform a MAPF execution run, return computed solution and update Run Statistics Table with the corresponding navigation statistics for that execution run

Figure 46: MAPF Endpoints

When an HTTP method request is sent to an API Gateway endpoint, the corresponding Lambda function is invoked. We may gain a greater detailed view of these interactions between the Tables, Lambda Functions, API Gateway Stages, Roles and Permissions as showcased in the visualisation below. This is achieved by utilising AWS CloudFormation, an infrastructure as code (IaC) service that allows easy modelling, provisioning, and managing of AWS and third-party resources.

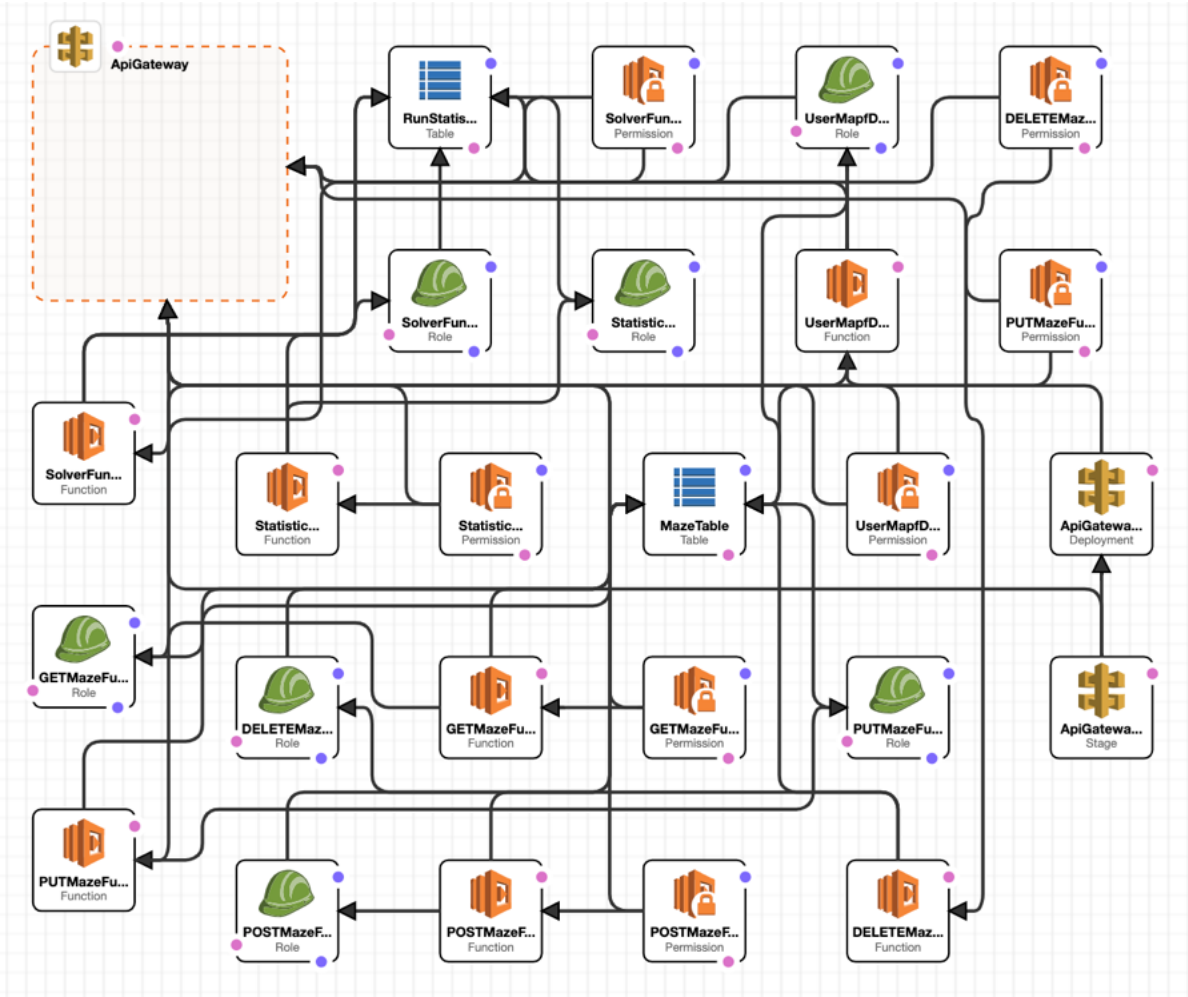


Figure 47: AWS Backend Interactions

7.3.5 Development Tools

This section outlines select development tools and applications utilised for the development of this MAPF application.

7.3.5.1 Postman

Postman is an application that simulates an API client, to aid effortless creation, sharing, testing and documentation of APIs [26]. In the context of this application,

Postman is especially beneficial in unit testing, troubleshooting and error isolation. It also allows the creation of API collections, which is useful to organise API end points for a simplified development experience.

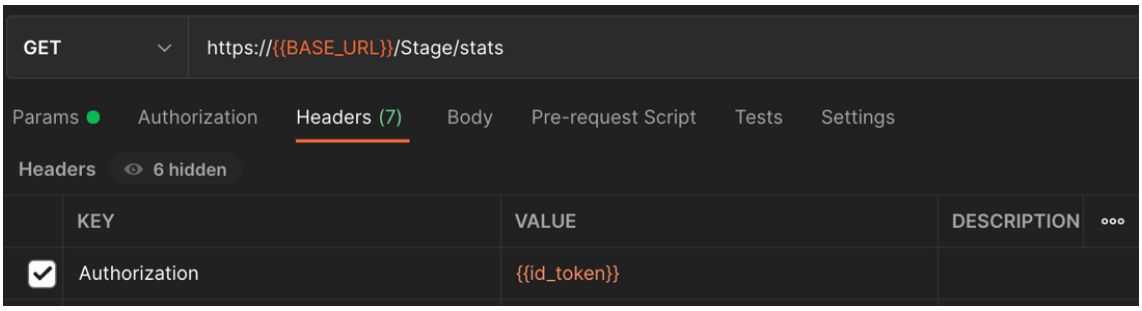


Figure 48: Postman API Invocation

Furthermore, Postman also provides configuration of environment variables, which is highly efficient for cases of repetitive use. As shown below, the MAPF base URL and the user ID Token are configured as environment variables. This allows us to change the variable values in a single centralised location, instead of modifying it across multiple instances of reuse.

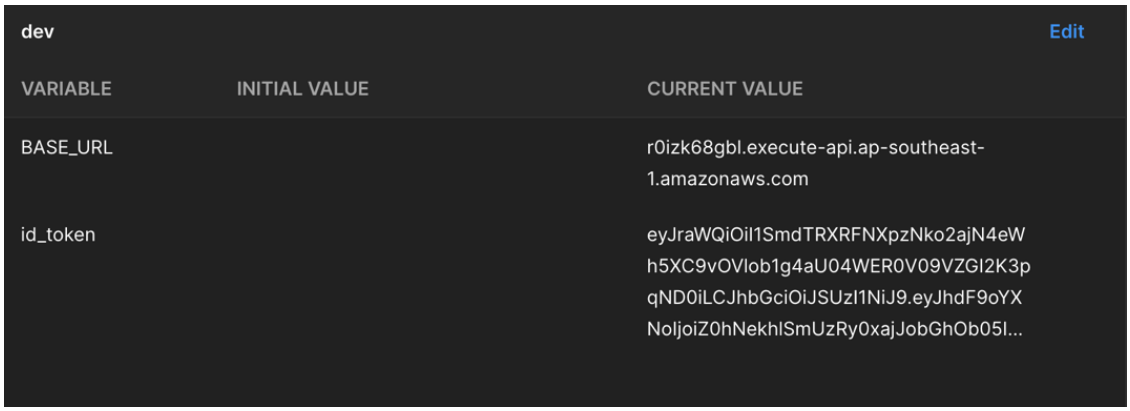


Figure 49: Postman Environment Variables

7.3.5.2 AWS CloudWatch

Amazon CloudWatch is a monitoring and management service that provides data and actionable insights for AWS, hybrid, and on-premises applications and infrastructure resources [27]. A key feature of this platform is that it allows the comprehensive monitoring of all API invocations.

This information is organised as log groups for each API end point. Further, each log group contains a log steam for every instance of invocation, which subsequently contains log events that describe the details of that invocation. Hence, this information is highly valuable for error tracing and debugging during integration.

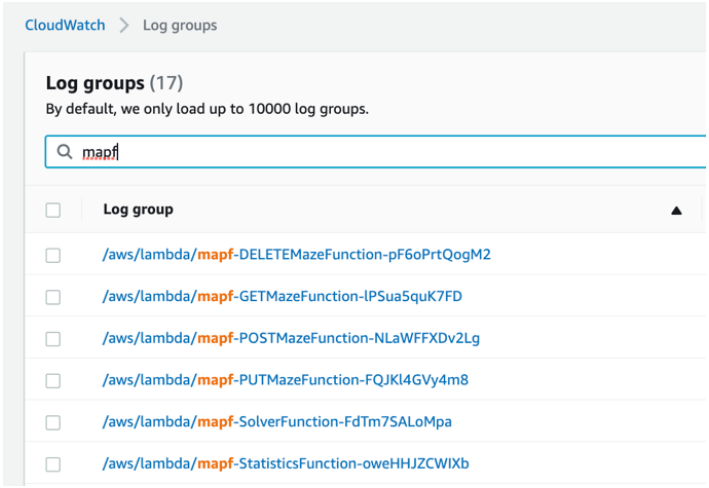


Figure 50: AWS CloudWatch Log Groups

7.4 Continuous Integration/Deployment (CI/CD) Workflow

Continuous Integration/Development, or more commonly CI/CD, is an approach to introduce automation in the stages of application development, from integration and testing phases to delivery and deployment [28]. Continuous Integration (CI) refers to new code changes being regularly built, tested, and merged to the hosting repository. Whereas Continuous Deployment (CD) handles the automation in consequent stages of the pipeline to facilitate deployment.

For efficiency and seamless development, a CI/CD pipeline has been designed and set up for the development of this project. The Continuous Integration workflow is triggered for every commit and pull request made, with the corresponding tests outlined in the “Testing” section below. On the other hand, Continuous Deployment workflow has been enforced via GitHub Actions [29], which is a CI/CD platform that allows automation of the build, test, and deployment pipeline. Some key features are showcased as follows:

- Creation of multiple workflows, that build and test every pull request to your repository, or deploy merged pull requests to production.

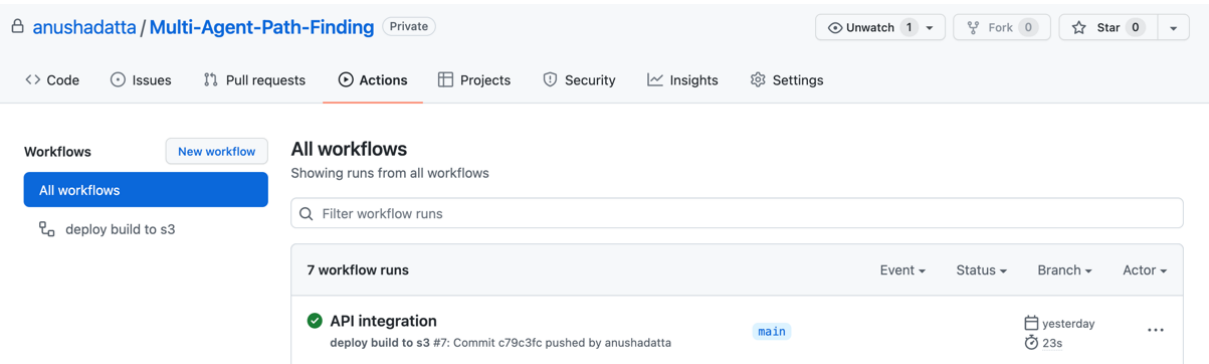


Figure 51: GitHub Actions Workflows

- View complete history of each workflow run execution, useful for traceability of development

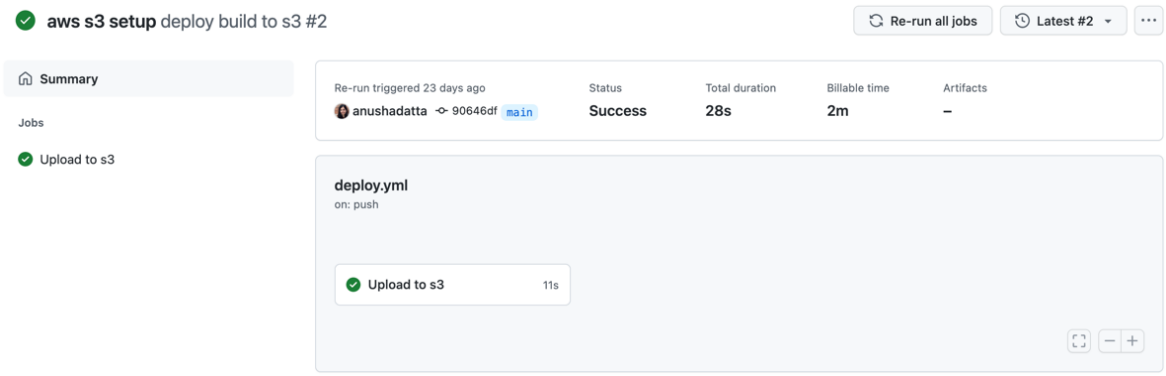
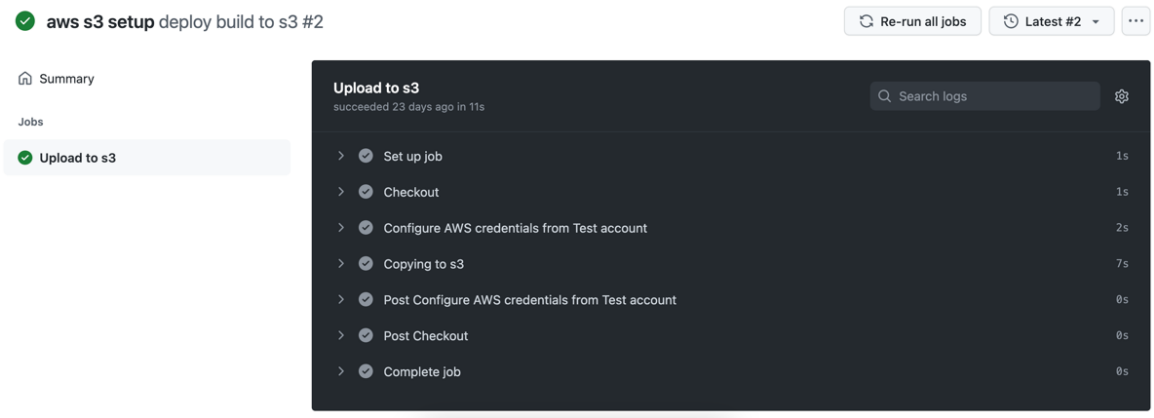


Figure 52: GitHub Actions Workflow History

- View workflow specific logs, indicative of successful or unsuccessful S3 deployment and useful for debugging the same. As observable below, we may additionally access details of each log, which provides valuable informative feedback to help identify the root of a deployment issue.



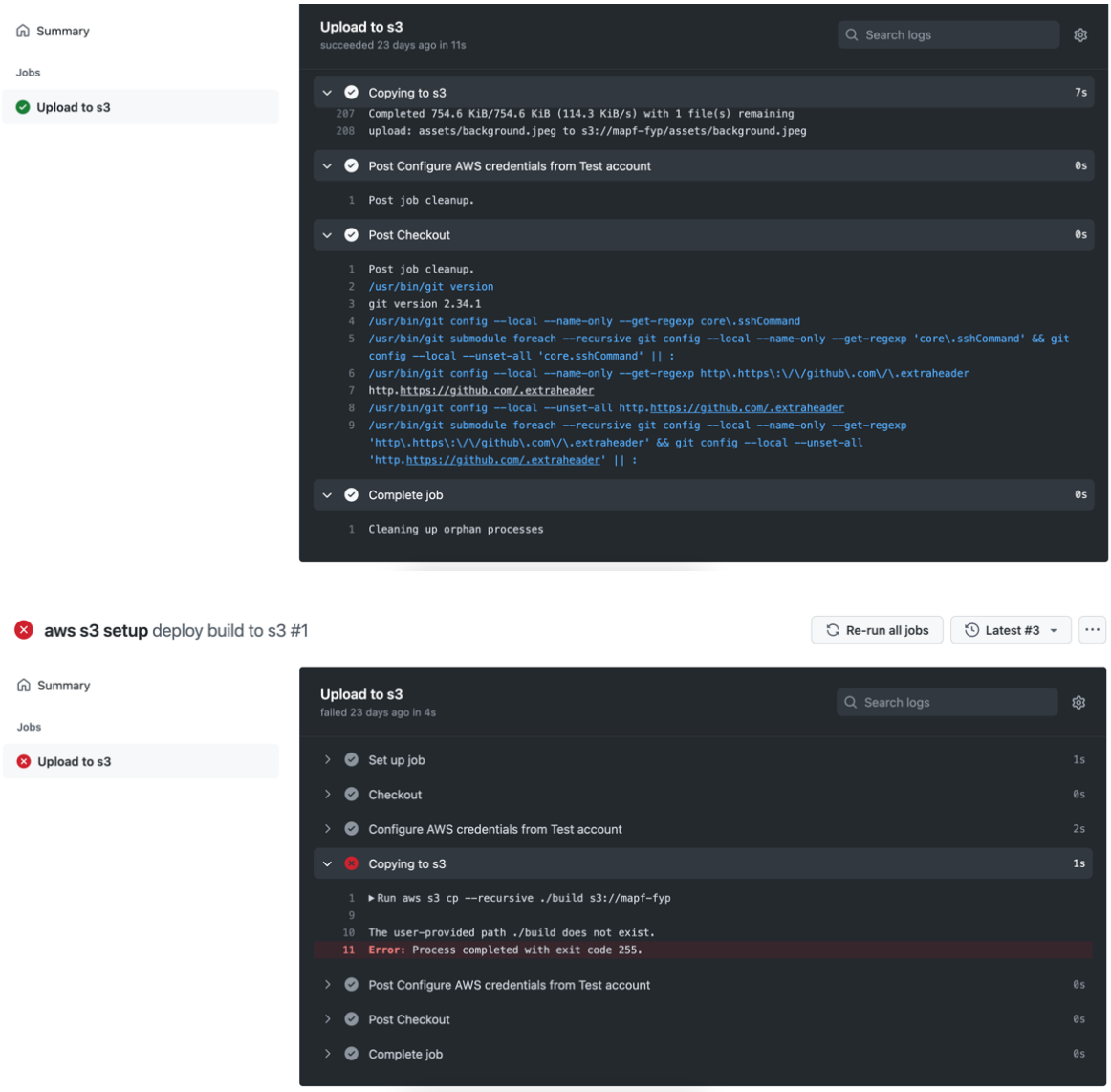


Figure 53: GitHub Actions Workflow Logs

8. Verification & Validation

Following the Implementation stage of the SDLC, we now move onto the next stage which is Verification and Validation. Verification process includes checking of documents, design, code and program whereas Validation process includes testing and validation of the actual product [30]. As per convention, Verification precedes Validation in this phase of the SDLC.

In context of this project, we perform Verification to check if the final software developed confirms to the specifications outlined in the design phase above. This is achieved by conducting investigative periodic reviews, application walkthroughs and inspections of the MAPF application throughout its development.

Upon the successful completion of implementation, we proceed with the Validation process. As opposed to Verification that aims to find bugs early in the development cycle, Validation aims to discover bugs that the Verification process did not identify. Validation checks whether the software meets the software requirements, through means of unit testing, white box testing, black box testing and integration testing. To ensure a comprehensive Validation process, these test cases are written with the goal to achieve full test coverage for the entire MAPF application.

9. Maintenance

The maintenance phase of the SDLC occurs after the software product is in full operation. Maintenance of software to modify and update software applications after deployments to fix bugs and improve system performance [31]. Moreover, software applications often need to be upgraded or integrated with new systems to keep pace with changing customer requirements. To ensure an easier software maintenance curve over the application's operational lifetime, this project takes a comprehensive approach to maintenance by aiming to address the following four types of software maintenance.

9.1 Corrective Maintenance

Corrective maintenance is a reactive modification of a software product to correct a known problem. This type of maintenance fixes defects or bugs in software, which often takes the form of quick updates performed on a recurring basis.

9.2 Adaptive Maintenance

Adaptive maintenance is the modification of software to keep it usable after a change to its operating environment. Many factors can change an application's environment, including new technical knowledge, hardware and security threats. These changes occur with greater frequency in most environments, so software that doesn't receive regular adaptive maintenance quickly becomes outdated.

9.3 Perfective Maintenance

Perfective maintenance improves the software's functionality and usability. It includes refining and deleting existing features as well as adding new features, easily making it the largest category of software maintenance. In addition to changing an application's functionality, perfective maintenance can also affect the way it looks. Changes to the software's interface and user journey are thus part of perfective maintenance.

9.4 Preventive Maintenance

Preventive maintenance is the modification of software to detect and correct software errors before they take effect. This type of maintenance is also commonly known as

future proofing. It includes making the software easier to scale more easily in response to increased demand and fixing latent faults before they become operational faults.

Hence, focusing on these discrete types of maintenance shall ensure that the MAPF application continues to evolve and adapt effortlessly with the changing software dependencies and system requirements.

10. Experiments

Following the completion of the software development lifecycle, which resulted in the full stack implementation of a cloud driven MAPF application, we now focus on exploration of MAPF execution performances and the factors affecting it.

This experimentation constitutes observing MAPF execution time and cost statistics for varying maze density layouts, agent counts, position initialisations and lower level search heuristics. For this purpose, we consider three mazes of differing densities - an empty maze with no obstacles, a sparse maze with scarce obstacles and a dense maze concentrated with obstacles. The purpose of considering varying maze layout densities is to facilitate collection of a representative set of MAPF data, as it is anticipated that different maze layout densities shall yield different results.

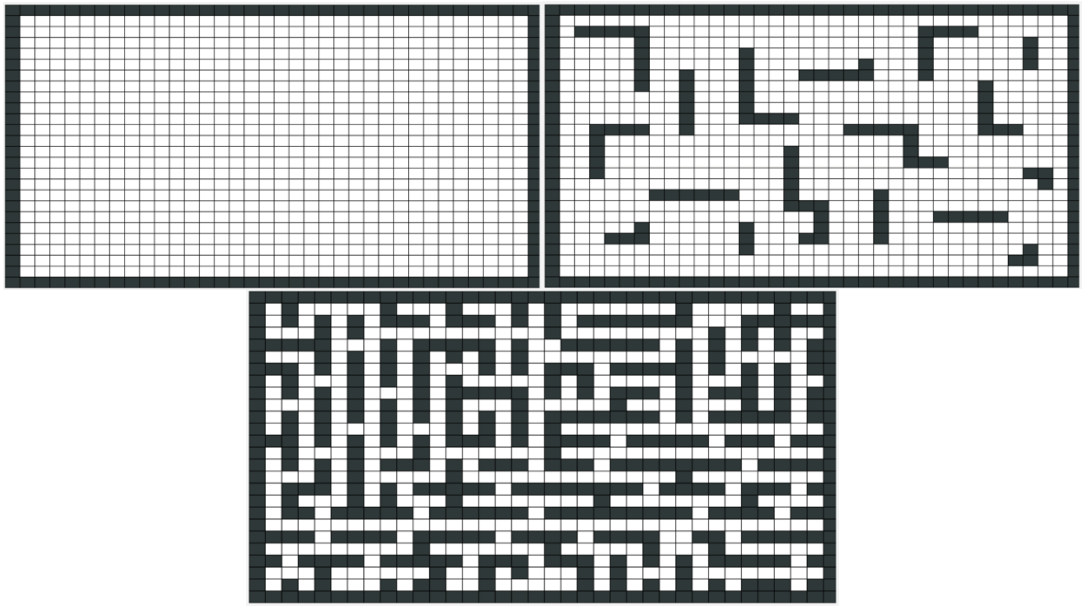


Figure 54: Empty, Sparse and Dense Maze Layouts

As for the agent counts, we consider the previously defined counts of 1 to 15 agents. Similarly, for the search heuristics we consider the choices of Manhattan Distance, Chebyshev Distance, Euclidian Distance. Additionally, we also consider a supplementary choice of No Heuristic, which simulates Uniform Cost Search that is optimal and complete but inefficient, as discussed above.

The experimentation entails that we perform 180 MAPF execution runs, with 1 instance of each maze layout, agent count and lower level search heuristic

combination. This would entail 60 execution runs for each maze layout (Empty, Sparse, Dense), with 4 sets of 15 runs for each heuristic (No Heuristic, Manhattan Distance, Chebyshev Distance, Euclidian Distance) and agent count (1 to 15) combination. These execution runs all comprise of identical agent initialisations in terms of start and goal positions. This is to ensure suitable comparisons between the execution time performances for different maze layouts and heuristics. As highlighted above, no comparison may be drawn for execution costs as all heuristics shall produce optimal paths with identical costs for runs with same agent initialisations. Hence, in this context, the execution cost visualisations simply serve to showcase the distribution of agent initialisations across all MAPF runs conducted. For this purpose, we perform an additional 135 MAPF execution runs with 45 execution runs for each maze layout (Empty, Sparse, Dense), with 3 sets of 15 runs for each heuristic (Manhattan Distance, Chebyshev Distance, Euclidian Distance) and agent count (1 to 15) combination, with varying agent initialisations.

In summary, a total of 315 execution runs are conducted with 180 execution runs having identical agent initialisations and the remaining 135 execution runs having differing agent initialisations. The experimental results derived for MAPF algorithmic performance, in terms of execution time, are based on execution runs with identical initialisations to ensure proper comparisons are drawn. Following the extraction of these insights, we proceed to perform the remaining execution runs with varying initialisations. These data points are subsequently analysed to obtain inferences regarding MAPF execution run distributions with respect to execution costs.

Note that these executions runs are conducted fully via the application graphical user interface developed. The statistical parameters generated from these experiments are pipelined into the navigation statistics visualisations, which are subsequently analysed to derive strong insights and predictive navigation trends.

11. Results

As described above, the experimentation process above resulted in a minimum lower bound of 315 MAPF execution run data points, which have been evaluated, visualised and studied extensively for navigation forecasting trends and insights.

As a first step, we closely examine the effect of lower level search heuristics on the MAPF performances. These performances are measured in terms of execution time for execution runs having identical agent initialisations. Upon studying the graphs for all maze layout densities, a general observation we may draw is that Manhattan Distance Heuristic performs superior in comparison to the other heuristics. This may be justified by the fact that Manhattan Distance is the most appropriate heuristic for agents with four allowable movement actions, as in this case.

Note that due to the limitation of computing resources, which restricts MAPF algorithm execution times to an upper bound limit of 30 seconds, we receive execution time graphs which are greatly constricted for higher agent counts. Hence, the positive corelation between agent counts and execution time is not well perceptible. However, on a general scale the data points are sufficiently informative to fuel our experiment findings for heuristic comparisons.

Upon studying execution time statistics for an empty maze, we may observe that No Heuristic performs the worst, with average execution time peaking at a higher agent count. This result is accountable as No Heuristic simulates Uniform Cost Search which is significantly inefficient. On the other hand, Manhattan Distance Heuristic performs the best with the lowest execution times of all heuristics. The substantiation for this potentially lies in the reasoning that Manhattan Distance is the most appropriate heuristic for our application, as detailed above.

However, on an overall scale it may be concluded that all heuristics perform largely the same, with notable variations only at higher agent counts. This may be potentially attributed to the lack of obstacles in an empty maze. As the execution times across runs for all heuristics are all nominally low, we may presume that an empty maze fails to provide a sufficiently challenging maze environment to adequately distinguish various heuristic performances, especially for lower agent counts.

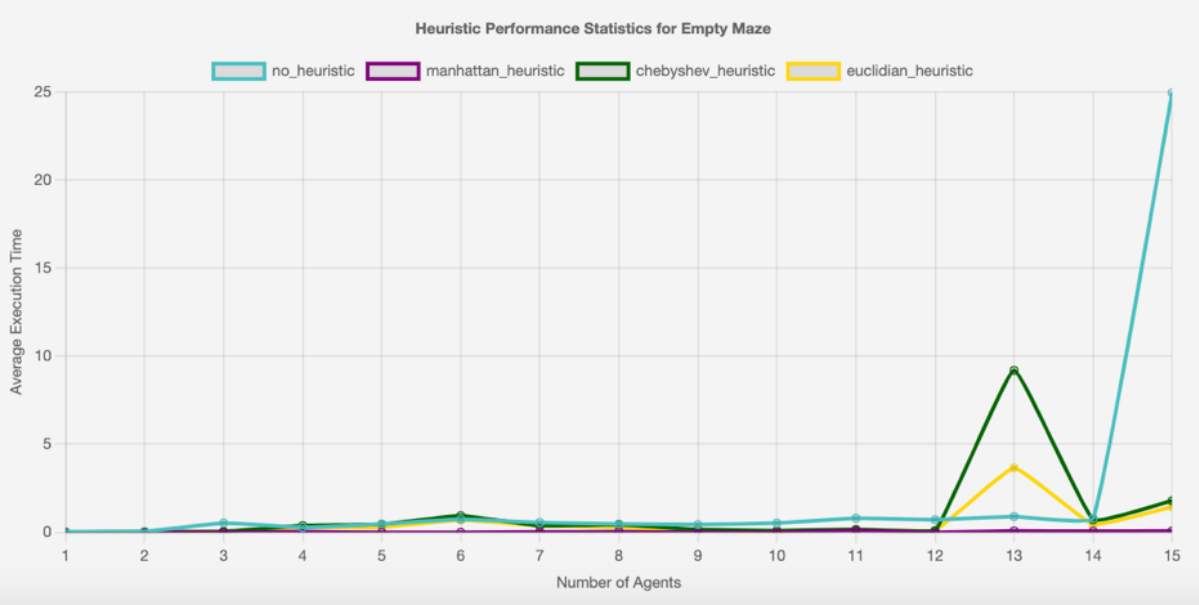


Figure 55: Heuristic Performances for Empty Maze

Next, we proceed to examine execution time statistics for the sparse maze layout. Upon inspection, we observe No Heuristic performs significantly worse in comparison to all heuristics. Similar as above, this is a foreseeable outcome considering its recognized inefficiency. However, in this case the difference in performance is starkly prominent in comparison to the previous case of the empty maze. In continuation to the line of argumentation above, this may be attributed to the maze obstacles that are absent in the empty maze. Hence, in contrast to the previous case, the presence of maze obstacles in the sparse maze may provide a sufficiently challenging maze environment to adequately distinguish various heuristic performances.

On the other hand, a similarity found between the sparse and empty maze is that Manhattan Distance performs the best with the lowest execution times of all heuristics, which is consistent with the justification detailed above.

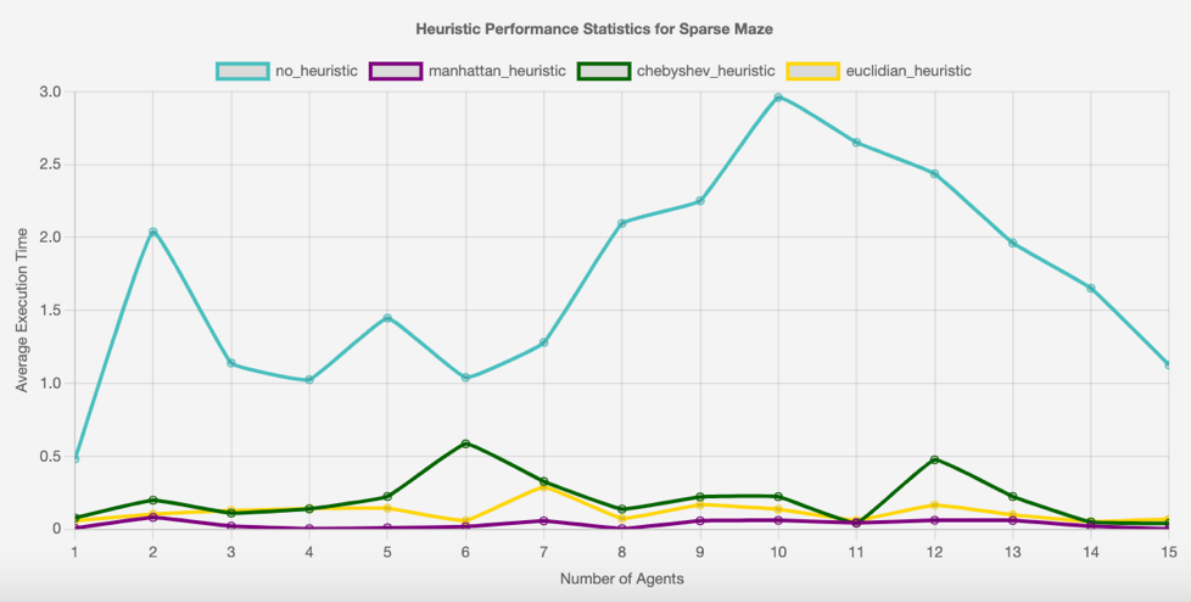


Figure 56: Heuristic Performances for Sparse Maze

Finally, we study execution time statistics for the dense maze layout. Upon observation, we conclude that largely all heuristics, except No Heuristic, appear to perform comparably in terms of execution time. However, as in both previous cases of sparse and empty maze layouts, Manhattan Distance still performs notably better than its counterparts. This is consistent with the argumentation outlined above.

Furthermore, similar to the case of sparse maze, No Heuristic performs significantly worse with unmistakably high difference in execution time for the dense maze layout too. This may be rationalised for the dense maze layout by the same obstacle presence logic discussed for the sparse maze layout.

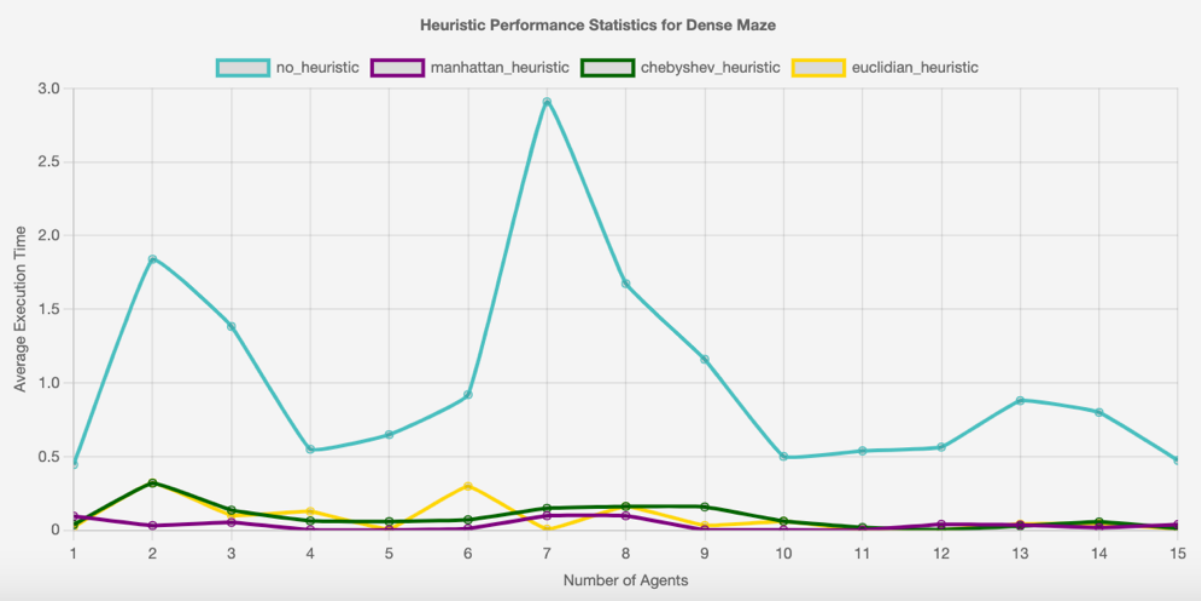


Figure 57: Heuristic Performances for Dense Maze

Following the experimentation on heuristic performances with identical agent initialisations, we now proceed to examine the MAPF execution runs distributions for varying agent initialisations. At a high level glance, we observe a positive correlation between the number of agents and the execution cost and time. Note that this correlation is applicable regardless of the lower level search heuristics, maze layout density or agent initialisation pattern. This is an expected trend outcome, as it may be concluded that a greater number of agents would decidedly result in a higher number of collisions. These increased collisions are assumed to contribute to a more expensive MAPF solution in terms of both cost and time.

Upon closer inspection, it may be observed that the execution costs and times, on average, are low for the empty maze layout, higher for the sparse maze layout and highest for the dense maze layout across varying agent initialisations. Intuitively, this too is an explainable result as an agent would be required to traverse longer paths around obstacles as opposed to direct paths for obstacle free maze layouts.

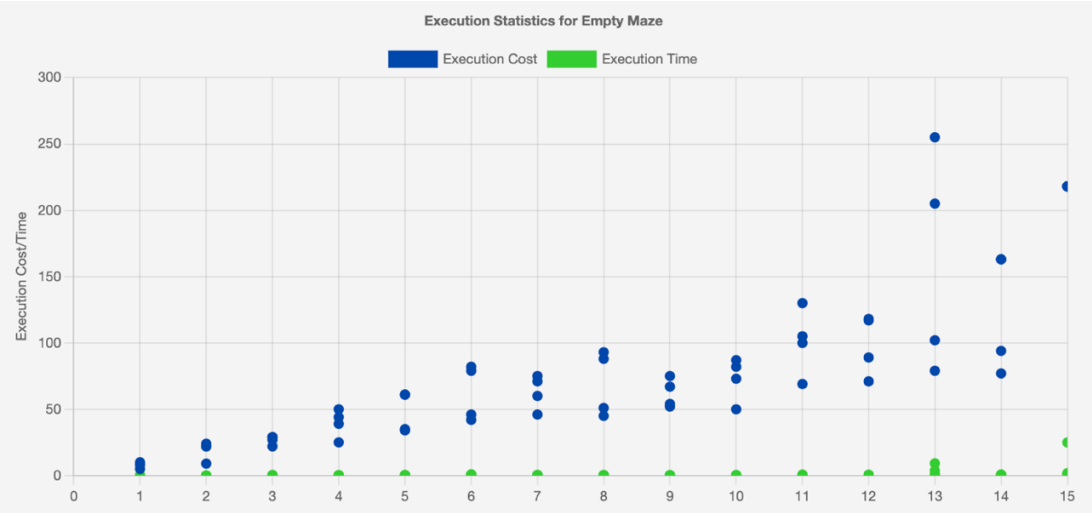


Figure 58: Execution Statistics for Empty Maze

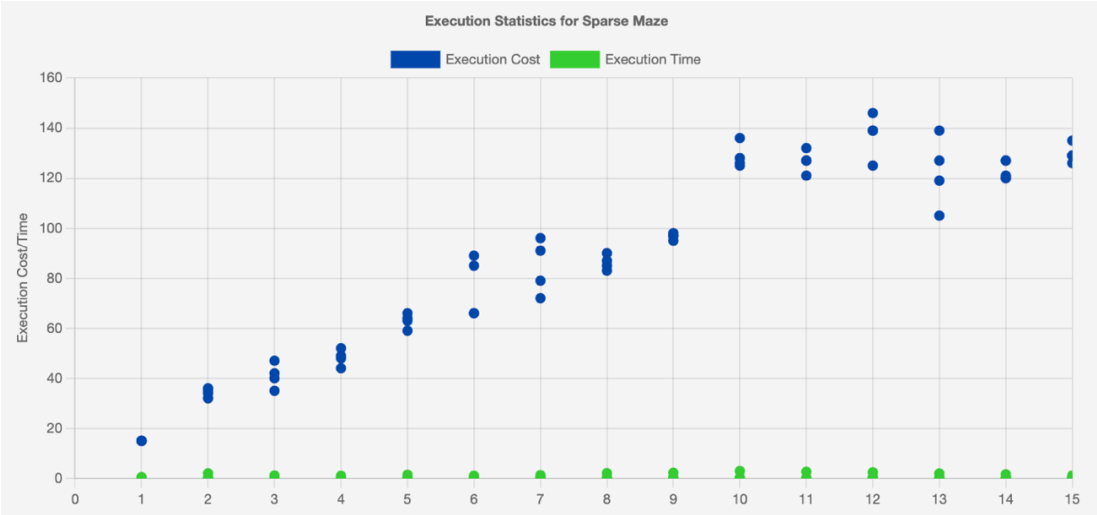


Figure 59: Execution Statistics for Sparse Maze

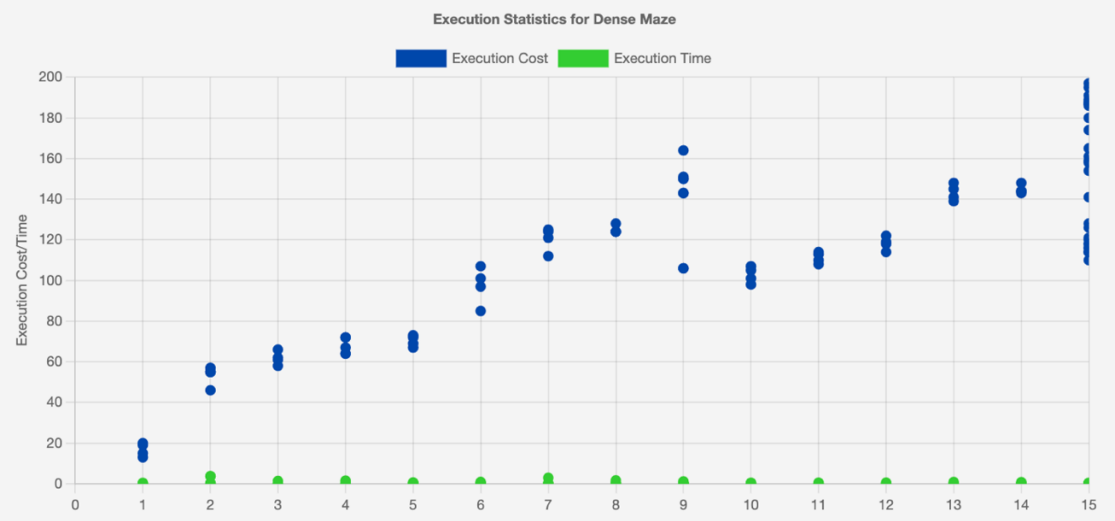


Figure 60: Execution Statistics for Dense Maze

Additionally, we may also observe heuristic specific average execution cost graphs, which aim to provide more granular insights into execution run distributions for varying agent initialisations. For this set of graphs, we observe that the Manhattan Distance Heuristic broadly yields low execution cost curves in comparison to the other heuristics. This insinuates that MAPF execution runs for this specific heuristic are distributed in a manner that may entail agent initialisations to be closely positioned in most cases.

For the empty maze layout, we may conclude that Chebyshev Distance Heuristic yields the highest execution cost, while No Heuristic yields the least. However, note that Manhattan Distance Heuristic provides a very comparable performance to No Heuristic in that context. As detailed above, the outcomes of this distribution suggest distantly spaced and closely spaced agent initialisations respectively.

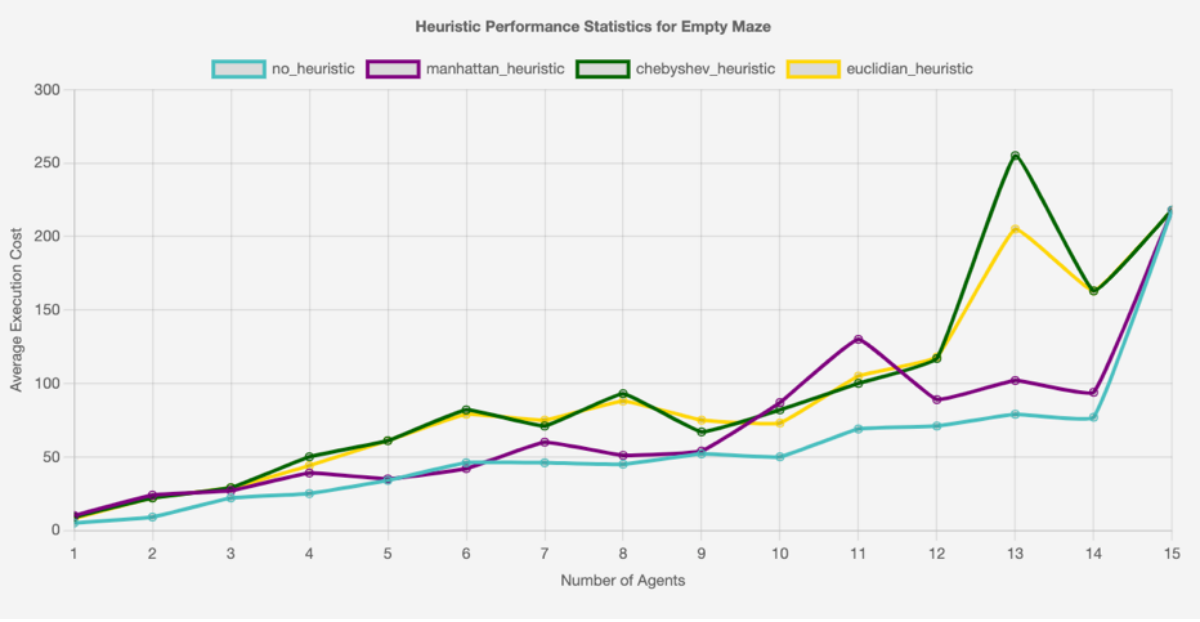


Figure 61: Execution Cost Distribution for Empty Maze

Next, for the sparse maze layout, we observe that execution costs are comparable for all heuristics across all agent counts. This trend suggests agent initialisations, and the corresponding distances between start and goal positions, being largely consistent across the execution runs for all heuristics.

However, note at agent counts 14 and 15 there is a sharp drop in execution cost for Chebyshev Distance Heuristic and Manhattan Distance Heuristic respectively. Conversely, this may indicate outlier execution runs which hint at more concentrated agent position placements.

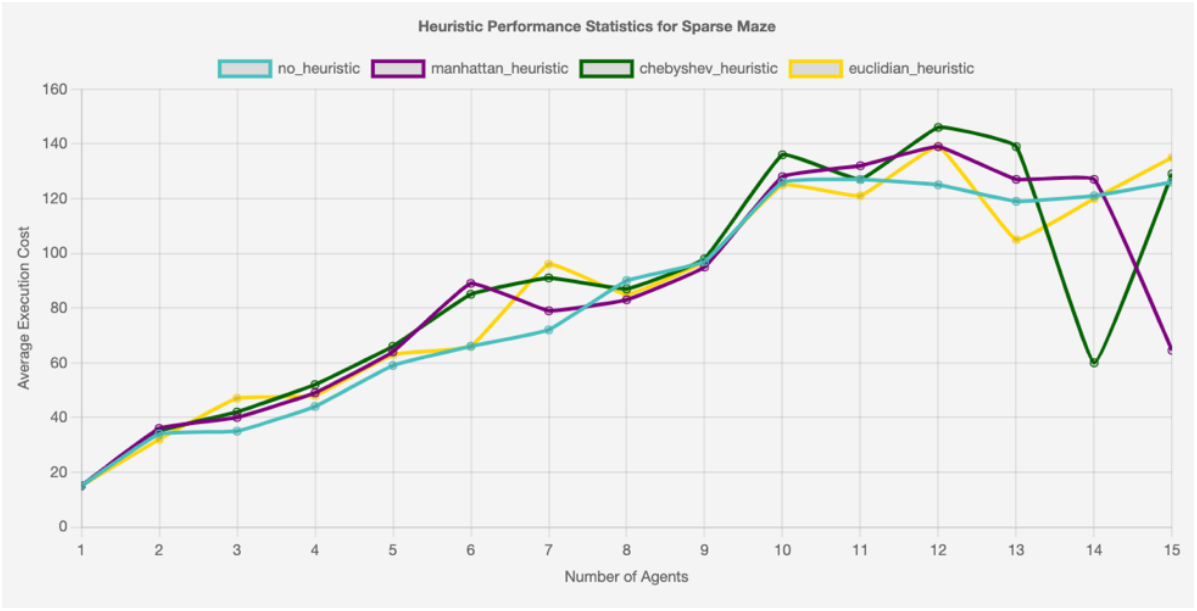


Figure 62: Execution Cost Distribution for Sparse Maze

Finally, for the dense maze layout, we observe that Chebyshev Distance Heuristic yields the highest execution cost, while Euclidian Distance Heuristic yields the least execution cost. However, note that Manhattan Distance Heuristic provides a comparable outcome to Euclidian Distance Heuristic in that context. Similar as above, these observations suggest distant and close agent placements across MAPF execution runs respectively.

Notably, these statistics are more wavering as opposed to those in the previous graphs. This may be attributed to the higher density of maze layout obstacles, which make the MAPF execution performances increasingly sensitive to the agent start & goal positions.

This rationalisation may be supported by the idea that differences in the initialisation parameters would determine shorter or longer paths the agents need to traverse around the obstacles, which could potentially lead to the varying results we observe below. Hence, it may be assumed that this specific graph provides a stronger indication of agent initialisation distribution.

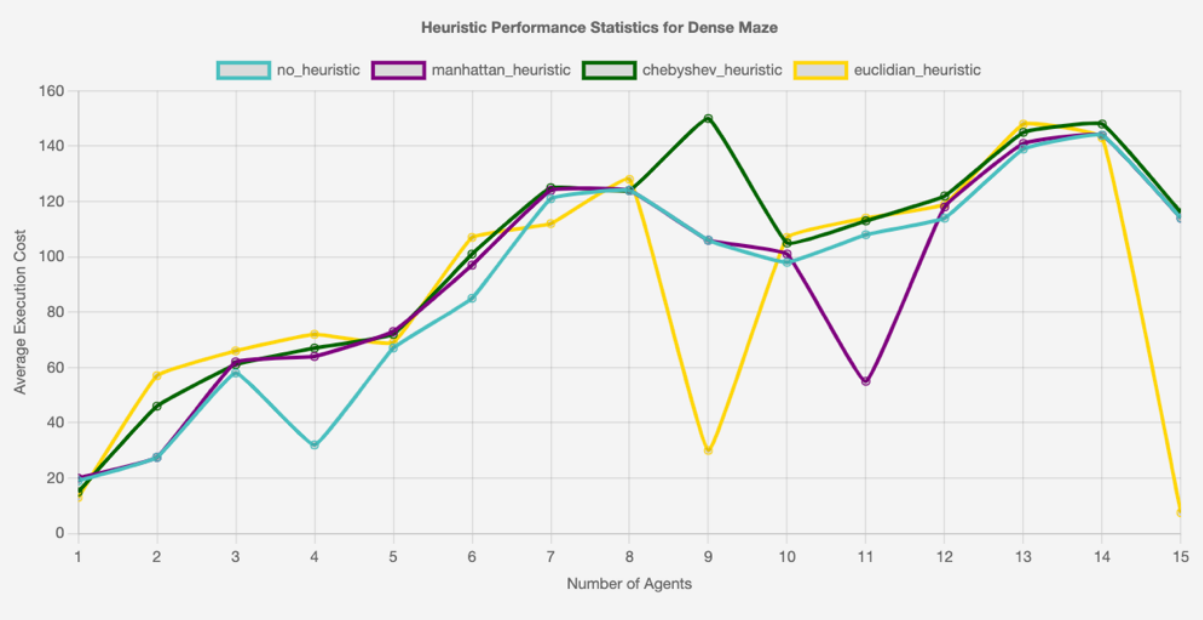


Figure 63: Execution Cost Distribution for Dense Maze

Note that these results are highly sensitive to agent initialisations and specific to the data points obtained for the experiments conducted in this project. Resultantly, the current outliers identified may be participative of a trend that is not discernible due to insufficient data. On the same note, more outliers may emerge which are not currently in existence. Hence it may be concluded that these trends and insights are subject to change with additional experimentation including, but not limited to, new maze layouts, differing agent start and goal positions or other unexplored heuristics.

12. Challenges Faced

Notable challenges encountered during the conception and execution of this project are listed as follows:

- **Cross-origin resource sharing (CORS)**
  CORS is a browser security feature that restricts cross-origin HTTP requests that are initiated from scripts running in the browser. In the context of this project, this error can be easily mitigated by configuring the permissions to “Enable CORS” on the AWS API Gateway console. However, the significant concern is presented due to the AWS API request handling being carried out in a manner that abstracts any error as a CORS error to the client. This abstraction is undesirable as it makes it extremely difficult to troubleshoot and mitigate the error.

- **Amazon Web Service Quotas**
  All AWS Services utilised for the purpose of this project are contingent to the AWS Service Quota Limit under the AWS Free Tier (refer to Appendix [A.4]), which restrict the allowable resources available for consumption. This limitation affects the MAPF executions, as they cannot exceed an upper bound of 30

seconds. As a result, there are instances wherein a solution cannot be generated for agent start and goal positions too far apart, due to a computation timeout. This also makes it challenging to gather sufficiently diverse data for a higher count of agents. Consequently, this affects the data collected for the navigation statistics and may bias the trends, especially for execution time. This concern may be mitigated by upgrading to a higher service quota. However, due to monetary limitations it was not feasible for this project.

13. Conclusion

To summarise, the successful completion of this project has resulted in the extensive study on the effect of lower heuristics on MAPF execution performances, and the implementation of Multi-Agent Path Finding as a full stack application powered by cloud services.

This MAPF application offers a seamlessly intuitive and personalised user experience with access to a multitude of features. This individualisation is achieved through authentication and allows the user to view, add, edit and delete their mazes stored persistently in the remote cloud datastore. Moreover, it allows the user to interact with the MAPF algorithm, visualise the traversal of the path finding solution and record statistical navigation parameters such as execution cost and execution time of the same. Subsequently, this data is funnelled into a navigation statistics pipeline to provide user customised predictive insights and navigation trends to aid intelligent data driven decisions.

This software system is designed in a manner that makes it both universally adoptable for diverse MAPF applications and easily adaptable to organisation specific requirements. Conclusively, this project hopes to inspire widespread adoption of industrial efficiency & productivity in terms of automation, by aiding organisations collectively monitor, control and measure the performance metrics of their fleet of agents effectively.

14. Future Works

While this project has been successful in comparing search performances of various lower level search heuristics and implementing a cloud driven MAPF application, below are few suggestions for future works in this field:

- Exploration and implementation of other lower level search algorithms (for instance Breadth First Search or Dijkstra's)
- Include option for user to upload a configuration file (suggested format JSON), of agent start and goal positions, to parse the required information directly
- Design and develop a live MAPF application walkthrough tutorial for users, for a more intuitive and engaging user experience
- Conduct User Acceptance Testing (UAT) to further refine user experience
- Extending web application to Android and iOS platforms as well

References

[1] LAM, C.-P., CHOU, C.-T., CHIANG, K.-H., & FU, L.-C. (2011). Human-Centered Robot Navigation-Towards a Harmoniously Human-Robot Coexisting Environment. IEEE Transactions on Robotics, 27(1), 99–112. https://doi.org/10.1109/TRO.2010.2076851

[2] Sharon, G., Stern, R., Felner, A., & Sturtevant, N. R. (2015). Conflict-based search for optimal multi-agent pathfinding. Artificial Intelligence, 219, 40–66. https://doi.org/10.1016/j.artint.2014.11.006

[3] Semiz, F., & Polat, F. (2021). Incremental multi-agent path finding. Future Generation Computer Systems, 116, 220–233. https://doi.org/10.1016/j.future.2020.09.032

[4] Z. Ren, S. Rathinam and H. Choset. (2021). Multi-objective Conflict-based Search for Multi-agent Path Finding, 2021 IEEE International Conference on Robotics and Automation (ICRA), pp. 8786-8791, doi: 10.1109/ICRA48506.2021.9560985

[5] Boyarski, E., Felner, A., Stern, R., Sharon, G., Betzalel, O., Tolpin, D., & Shimony, S.E. (2015). ICBS: The Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding. *SOCS*.

[6] *A* Search Algorithm* [online]. Available: https://en.wikipedia.org/wiki/A*_search_algorithm

[7] Candra, A., Budiman, M. A., & Hartanto, K. (2020, July). Dijkstra's and A-Star in Finding the Shortest Path: a Tutorial. In 2020 International Conference on Data Science, Artificial Intelligence, and Business Analytics (DATABIA) (pp. 28-32). IEEE.

[8] Siahaan, A. P. U. (2018). Implementation of A-Star Algorithm in Determining the Shortest Path on Graph.

[9] F. Duchon, A. Babinec, M. Kajan, P. Beno, M. Florek, T. Fico, & L. Jurisica, “Path Planning with Modified A Star Algorithm for a Mobile Robot”, Elsevier Modelling of Mechanical and Mechatronic Systems, pp. 59-69, 2014.

[10] Cui X and Hao S (2011) A*-based pathfinding in modern computer games International Journal of Computer Science and Network Security. 11.1. pp.125-130

[11] Coghetto, R. (2016). Chebyshev distance. Formalized Mathematics, vol.24, no.2, 2016, pp.121-141. https://doi.org/10.1515/forma-2016-0010

[12] *Heuristics* [online]. Available: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#S7

[13] J. Pearl, Search and heuristics / edited by Judea Pearl. Amsterdam ;: North-Holland Pub. Co., 1983.

[14] A. Felner, “Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding”, ICAPS, vol. 28, no. 1, pp. 83-87, Jun. 2018.

[15] Rizvi. (2012). Appendix 5—Waterfall Software Development Lifecycle Model. In Microcontroller Programming (pp. 513–514). CRC Press. https://doi.org/10.1201/b11682-22

[16] *Right to Erasure* [online]. Available: https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/individual-rights/right-to-erasure/#:~:text=Under%20Article%2017%20of%20the,be%20created%20in%20the%20future

[17] Shneiderman, B. (2004). Designing for fun: how can we design user interfaces to be more fun?. interactions, 11(5), 48-50.

[18] F. H. Lochovsky, “Entity-relationship approach to database design and querying : proceedings of the Eighth International Conference on Entity-Relationship Approach, Toronto, Canada, 18-20 October 1989 / edited by Frederick H. Lochovsky.,” 1990.

[19] Molnar, D., & Schechter, S. E. (2010, June). Self Hosting vs. Cloud Hosting: Accounting for the Security Impact of Hosting in the Cloud. In WEIS.

[20] *AWS Documentation* [online]. Available: https://docs.aws.amazon.com/

[21] *p5js Documentation* [online]. Available: https://p5js.org/

[22] *Chart.js Documentation* [online]. Available: https://www.chartjs.org/

[23] *Java Web Token* [online]. Available: https://jwt.io/

[24] *PartiQL Documentation* [online]. Available: https://partiql.org/

[25] X. Ma, R. Li, Z. Lu, J. Lu, and M. Dong, “Specifying and enforcing the principle of least privilege in role-based access control,” Concurrency and Computation: Practice and Experience, vol. 23, no. 15, pp. 1313–1331, 2011, doi: 10.1002/cpe.1731.

[26] *Postman Documentation* [online]. Available: https://www.postman.com/

[27] *AWS CloudWatch* [online]. Available: https://aws.amazon.com/cloudwatch/

[28] Shahin, M., Babar, M. A., & Zhu, L. (2017). Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE Access, 5, 3909-3943.

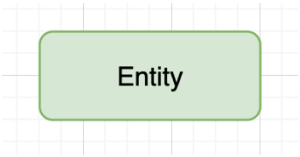
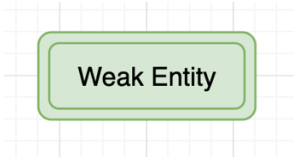
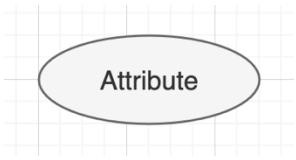
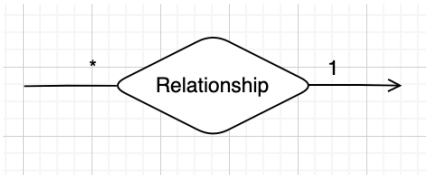
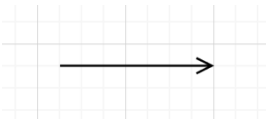
[29] *Workflow syntax for GitHub Actions* [online]. Available: https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions

[30] A. Dasso and A. Funes, Verification, validation and testing in software engineering [electronic resource] / Aristides Dasso, Ana Funes [editors]. Hershey, Pa: IGI Global 701 E. Chocolate Avenue, Hershey, Pennsylvania, 17033, USA, 2007.

[31] Schneidewind, N. F. (1987). The state of software maintenance. IEEE Transactions on Software Engineering, (3), 303-310.

Appendix [A]

[A.1] Entity Relationship Notation

Symbol	Name	Description
	Entity Set	A table/relation.
	Weak Entity Set	An entity that cannot be uniquely identified by its attributes alone. Hence, it requires a foreign key in conjunction with its attributes to create a primary key. The foreign key is typically a primary key of an entity it is related to.
	Attributes	A table characteristic (or column). Key attributes are denoted with an underline.
	Many-to-1 Relationship	Cardinality that specifies a relationship between two entities where there are multiple records from one entity associated with a single record from another entity.
	Referential Integrity	Referential Integrity, or forced participation, is the property ensuring all data references are valid. It requires that if a value of one attribute of a relation references a value of another attribute (Foreign Key), then the referenced value must exist. This ensures data accuracy and consistency across tables.

[A.2] Boyce Codd Normal Form Decomposition

The beneficial properties of BCNF include minimal data redundancy and a guaranteed lossless join across relations. To perform this normalisation, we first perform a validity check to see if the current schema is in BCNF. If not, then we proceed to the decomposition step.

- **Checking for BCNF**
  - Derive an exhaustive list of all Functional Dependencies
  - A Functional Dependency (FD) is a constraint between two sets of attributes in a relation from a database. This is represented in the form of $X \rightarrow Y$, where X and Y are attributes and $\rightarrow$ implies that Y may be derived from X.
  - Derive all non-trivial FDs, which is any FD where some attribute in the right hand side (RHS) of FD does not appear on the left hand side (LHS)
  - Derive all keys for the relation
  - Check if the LHS of each non-trivial FD contains a key or super key of relation
  - If yes, FD is in BCNF else decompose it following steps below

- **Decomposition into BCNF**
  - Given a relation R, find every FD $X \rightarrow Y$ on R that violates BCNF
  - If no such FD found, stop
  - Compute the closure set $\{X\}^+$, defined as the set of attributes that may be determined by an attribute X
  - Decompose R into two tables R_1 and R_2, such that
  - R_1 contains all attributes in $\{X\}^+$
  - R_2 contains X and all attributes not in $\{X\}^+$
  - Repeat steps recursively on R_1 and R_2

[A.3] Armstrong's Axioms

1. Reflexivity: Given $XZ \rightarrow YZ$, we have $XZ \rightarrow Y$
2. Augmentation: Given $X \rightarrow Y$, we have $XZ \rightarrow YZ$
3. Transitivity: Given $X \rightarrow Y$ and $Y \rightarrow Z$, we have $X \rightarrow Z$

[A.4] AWS Budgeting

This section contains a service-wise breakdown of the costs required to use each AWS offered service utilised in this project.

[A.4.1] AWS Cognito

AWS Cognito is charged based on the number of Monthly Active Users (MAU). The Cognito Your User Pool feature has a free tier of 50,000 MAUs for users

who sign in directly to Cognito User Pools. After expiration of the free tier, the following pricing model is utilized.

Pricing Tier (MAUs)	Price (per MAU)
50,001-100,000 (after the 50,000 free tier)	$0.0055
Next 900,000	$0.0046
Next 9,000,000	$0.0033
Greater than 10,000,000	$0.0025

[A.4.2] AWS Lambda Functions

AWS Lambda Functions are charged based on the number of times a function is executed. The free tier of Lambda allows users 1 million free executions. After expiration of the free tier, the following pricing model is utilized.

Architecture	Duration
x86 Price	$0.0000166667 for every GB-second
Arm Price	$0.0000133334 for every GB-second

[A.4.3] AWS API Gateway

AWS API Gateway is charged based on the number of API calls made, with the first 1 million API calls made for free. After the expiration of the free tier, the following pricing model is utilized.

Number of Requests (per Month)	Price (per million)
First 300 million	$1.25
300+ million	$1.13

[A.4.4] AWS S3

AWS S3 is charged based on the number of files uploaded to a bucket. The free tier of S3 enables users to store up to 5GB of data in any bucket. After the expiration of the free tier, the following pricing model is utilized.

First 50 TB (per Month)	$0.025 per GB
Next 450 TB / Month	$0.024 per GB
Over 500 TB / Month	$0.023 per GB

[A.4.5] AWS DynamoDB

The free tier of AWS DynamoDB allows 25 GB of storage and up to 200 million read/write requests per month. After expiration of the free tier, pricing may be set up either on an on-demand basis or on in provisioned capacity mode.