Multi-Agent Asynchronous Real-time Maze-solving

by

Hamid Alian

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Control Systems

Department of Electrical and Computer Engineering

University of Alberta

Abstract

In recent years, mazes have been used to study robot behavior by assessing their ability to navigate a maze using various methods. The act of finding a path through a maze from beginning to end is known as maze solving. Some maze-solving methods are employed for use by an agent with no previous knowledge of the maze, while others are designed for use by someone or a computer program capable of seeing the whole maze at once. We consider a situation in which multiple agents are randomly distributed inside an arbitrary rectangular maze and have no previous knowledge of the maze. We provide a solution in the form of an algorithm for the agents to cooperate collaboratively to discover and achieve the hidden goal. We divide our algorithm into two major phases and present rules for each, with each agent is programmed to follow these rules individually. We explained the algorithm's implementation by addressing the challenges we have such that the agents can follow the algorithm in such a way that all agents may move simultaneously. We evaluate our approach using a computer simulation of a square-shaped maze with varying sizes and a variable number of agents. The algorithm performs well in the simulation, is efficient, and reflects the trade-off between utilizing a single agent and multiple agents. We provided the solution's results for both phases. We then validate our algorithm on a physical system consisting of a real maze and many robots. Our solution is primarily based on cooperating and working parallel with all agents.

To my beloved parents, and my sister Delaram
for their endless love and support

Acknowledgments

I would like to convey my heartfelt gratitude to Professor Horacio J. Marquez for his invaluable assistance and supervision during this study endeavour. Without his constant supervision, this thesis would not have been feasible.

I would like to express my gratitude to all the friendly folks I encountered throughout my master's program who assisted me in calling Edmonton home. I'd want to express my gratitude to Mohammad, Afshin, Reza, Mahyar, and Amir for their warm encouragement and help. I would like to thank Athena for all her love and support.

I want to thank my friends, Mohammad, Mohsen, Marzieh, Behdad, and Yusuf, who helped me with the problems I encountered in these two years.

Last but not least, I want to express my heartfelt thanks to my parents and sister for their unconditional love.

Hamid Hajimirza Alian
Edmonton, Alberta
Canada

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis considers the problem of solving a maze by a group of multiple agents distributed randomly in an arbitrary rectangular maze, assuming that the agents have no prior knowledge of the maze. We propose a solution in which the agents, working cooperatively, find the hidden destination. The solution is first tested using computer simulation to verify the expected performance of the proposed algorithm. Finally, the results are validated by implementing our algorithm in real-time using a team of robots. In this chapter, we provide an overview of the subject along with some preliminary background.

1.1 Background

A maze is a path or a series of paths that lead from one point to another, usually from an entrance to a destination. Both branching tour puzzles, in which the solver must identify a way to a goal, and simpler non-branching ("unicursal") patterns, that go through a complicated layout to a destination, are referred to as a "maze". The term "labyrinth" is sometimes used interchangeably with "maze" although it can also refer to a "unicursal" design. A maze's routes and walls are usually fixed. Labyrinths and mazes are found all over the world and have long been a source of fascination. Explorations in various cultures throughout the world, including Egypt, India, and North America, have added to the labyrinth's dominance in western civilization [6]. The act of finding a path through a maze from beginning to end is known as maze solving. Some maze-solving methods are intended for use by an agent who has no prior knowledge of the maze, while others are intended for use by someone or a computer program that can view the entire maze at once [8].

In recent years, mazes have been used to study the behaviour of robots by examining

Traquair House Maze Scotland[7]

Maze at the Missouri Botanical Garden in St. Louis[8]

A hedge maze at Longleat stately home in England[9]

Figure 1.1: Mazes created by humans from across the globe

their ability to explore a maze using different algorithms [10] [11].

Many studies have been reported in the literature on the solution of a maze. We begin by describing what a "maze" refers to in the research studies, which allows us to discuss the topics around the maze in a common language.

Reference [12] defines a maze as a two-dimensional grid of any size, generally rectangular, that is grid-like. A maze is made up of cells. A cell is the basic maze element. The maze may include any number of distinct obstacles. The agent is put in the maze on an empty cell at random. The agent may travel in any direction but must do so only through unoccupied space. The goal is to figure out a strategy for getting to the destination, or "goal," as quickly as possible. When the goal is reached, the maze is considered solved.

To solve a maze, a variety of methods derived from graph theory and non-graph theory have been proposed in the literature. [13] clarifies how graph theory can be utilized to solve a maze and, after a thorough examination, demonstrates how graph theory outperforms non-graph theoretic algorithms and compares the algorithms' efficiency. Graph theory algorithms used for maze searching in the research include *flood fill* algorithms (FF), *modified flood fill* algorithms (MFF), *depth first search* algorithms (DFS), and *breadth-first search* algorithms (BFS)[13].

In terms of graph theory solvers, a maze may be mathematically modelled as a graph, $G = (V, E)$, with each cell in the maze representing a vertex in the set V and the obstacle-free corridor between two cells being an edge in the set E [14].
DFS, also referred to a *Tremaux* algorithm by some authors, starts at the root of the graph as the entrance point and explores the deeper sections of the graph until reaching a dead-end and then backtracks. The algorithm begins from a vertex, then branches out adjacent vertices until it reaches the end or goal point. The whole maze is represented as a graph, with nodes or vertices acting as maze cells. The agent visits each cell once in each direction before returning to the source or original cell. The agent will continue to search the cells until finding the desired cell, keeping track of the cell walls/obstacles[13][15].

Flood Fill (FF) algorithms are inspired by the concept that water always flows from a higher to a lower height [16][17]. FF algorithms implement this concept by assigning a number to each cell in the maze that represents the distance between a cell and the destination cell. The cells with greater values represent higher heights, while those with lower values represent lower elevations [18]. The destination cell is given a value of zero, which corresponds to the lowest height. The agent is one cell distant from the goal if it is standing in a cell with a value of 1. The agent is three cells distant from the destination if it is standing in a cell with a value of 3, assuming that the robot is unable to move diagonally. After the maze has been flooded and the cell values renewed, it is traversed, and the maze map is updated after each traversal. When a new cell is traversed, the array described above is created, and the adjacent cell with the lowest value is determined. The agent path always consists of cells from higher values to cells with lower values [17].

Unlike the flood fill algorithm, the modified flood fill algorithm does not flood the maze when a new cell is reached. Instead, it uses recursive steps to update adjacent cells:

- Push the current cell to the top of the stack.
- Keep repeating this step until the stack is empty: From the stack, pop the current cell location. If the minimum distance of the adjacent open cells is not equal to the current cell's distance - 1, replace the current cell's distance with the

minimum distance + 1 and push all adjacent cells locations onto the stack[18].

Reference [18] provides a detailed explanation of the algorithm summarized above. More research in the literature discusses novel algorithms for maze solving, which are improvements to the algorithms described above.

[19] improves the flood fill algorithm in maze-solving by omitting the calculations needed when entering a dead-end channel. There is only one way to proceed in these channels, and there is no need to renew the maze array values. [20] proposes the "Partition-central Algorithm," a maze-exploring algorithm that finds the shortest path in a micromouse competition maze. This algorithm breaks a maze into 12 divisions and applies various rules to different sections, making the exploring process more flexible and increasing a micromouse intelligence. [21] lookes at how light beams diffract from a source to a target to solve a challenging maze with open regions. With a ray-based approach to maze-solving, it may be feasible to pick a small number of vertices (reradiation points) and link them with a limited number of pathways (those that the light rays would follow) to characterize the maze adequately. A maze with open regions may be reduced to an abstract form appropriate for typical maze-solving algorithms using this method. [22] demonstrates how discretely assigned potential levels may be utilized to determine autonomous route selections for a mobile robot. It also shows how to assign and manipulate these potentials to give locally optimum path choices while keeping the potentials' integrity. [23] presentes a maze-solving robot system based on image processing and a graph theory algorithm. While traveling through a real maze, the system selects the optimal route for a car-like robot from its starting location to its destination position. A camera captures the whole maze, which is then processed and evaluated by a program based on the Breadth-First Search algorithm.

The Agent, an autonomous mobile robot that explores the maze, is a key concept in the solutions above. The agent is a part of the environment who is able to make decisions regarding the states of the environment and is able to cooperate, communicate and adapt to the environment and other agents [14]. Autonomous mobile robots play a significant part in our lives and may be the best alternative for various jobs. They may be used in industries to carry components and products accurately and quickly from one station to another. They have also been employed to save lives and reach dangerous locations where humans are unable to go. Mobile robots may also be used for home automation, such as autonomous vacuum cleaners that must navigate themselves across the house while cleaning it simultaneously [23][24].

As mentioned earlier, much research has focused on maze solving. Multi-agent methods have been introduced as improvements to these solutions, which significantly reduce the time-consuming aspect of the solution. Hereafter, we focus on the definition of multi-agent concepts and summarize what has been done so far.

According to the book "An Introduction to MultiAgent Systems" [25], an agent is an entity with domain knowledge, objectives, and specific behaviours. Multi-agent systems are a group of agents that communicate in a shared space. Multi-agent systems are concerned with the design and coordination of complex systems incorporating several agents. A multi-agent system is a distributed computer system containing autonomous interacting intelligent agents that collaborate or compete to accomplish their objectives. Reference [26] highlights many benefits of multi-agent systems including increased efficiency, a broader work domain, and the ability to move about in a dispersed manner. Furthermore, a multi-robot coordination mechanism may help with the problem-solving phase in terms of flexibility and adaptability. A major task is broken into tiny subtasks and distributed among numerous agents in multi-agent robotics. Each agent must do their own specific task and communicate with one another, relaying information about their location, activity, direction. Cooperation and information exchange would aid them in reaching the destination more quickly. Each agent in a multi-robot system has limited capabilities; nevertheless, by cooperating, they may benefit from the abilities of others. This results in decreased energy usage and a faster job completion time[27].

Multi-agent systems is a vast area of study, and many research studies have been conducted toward deep understanding and connections: For example, the problems of traditional "agent-centered" multi-agent systems are highlighted in [28]. This reference argues that an organization-centered multi-agent system, or OCMAS for short, may be utilized to overcome these problems and proposes a set of fundamental principles for designing real OCMAS. This organization-centered multi-agent system is the main formation that we focused on in this thesis. Also, our interest is in path planning for multiple robots or agents. More explicitly; we consider a group of agents who simultaneously search for the maze exit in a cooperative manner. In this scenario, agents share their knowledge of previously explored cells in the maze, helping other agents to reach the exit. Clearly, the cooperative nature of the problem can highly reduce the time required by the group of agents to reach the solution.

The task of "multi-agent pathfinding" refers to planning a sequence of moves by a group of agents to reach a certain goal/location. The agents move in a certain field to find the goal while avoiding obstacles and collisions among them [29] [30]. MAPF can be divided into two types: "distributed setting," in which each agent has its own processing power and decision-making system [31], and "centralized setting," in which a single decision-maker manages all agents [14]. In previous studies, multi-agent pathfinding was studied using two main approaches, namely: (i) search-based solvers, [14], and (ii) artificially intelligent solvers, [32]. Search-based solvers aim to minimize the time required by all agents to reach their destination. Agent movements are planned one at a time according to predefined orders [14]. Artificial intelligence solvers, on the other hand, learn to generate a maze description and find an exit without having to relearn new rules every time they encounter a new maze. These solvers mainly discuss the so-called *complete information case*, consistent of finding an optimal path over a field, assuming that the graph model is known to the user. In contrast, we consider the *incomplete information* case; *i.e.* we assume that the graph model of the maze is unknown and agents do not know the geometry and position of the obstacles and the way to exit. We assume that agents use their sensors to detect obstacles in the maze. In this case, the solution requires a *local online algorithm*, defined as one that operates with limited information at any given moment, [33].

Reference [34], for example, demonstrates an architecture for the design and deployment of cooperative maze discovery robots (CLDRs), which work together to find a path out of an undiscovered maze. CLDRs make use of semantic technologies to describe and retrieve maze data like pathways and obstacles. For future study, this paper also recommends multithreaded programming in a real-time operating system for the microcontroller rather than sequential processing with hardware interrupts, which was the emphasis of our research.
The problem of multi-agent maze solving is presented in reference [35], in which a group of coordinated agents must go from an entering location to a target position without previous knowledge of the maze. This reference proposes an algorithm that improves the depth-fist search maze solving method. This paper's algorithm spreads the agents in the maze and provides a valid solution. It also evaluates it in terms of the average number of steps needed. While the depth-first search algorithm is for a single agent, the whole group's actions in this algorithm is a breadth-first search strategy since the other agents repel each other. As a result, this algorithm combines these two graph search methods. Moreover, this article proposed an extended version of its algorithm to be used when the agents do not start in the same location.
The goal of reference [36] is to provide Open-World Assumption-based solutions to maze challenges. The work demonstrates that the Open-World Assumption might be used to replace traditional logic programming methodologies. It also presents a technique for deciding whether to use the Closed-World Assumption or the Open-World Assumption to drive decision-making and reasoning and used this approach to

manage collaborative maze exploration robots' decision-making.

1.2 Summary of Contributions

In this thesis, our interest is in path planning for multiple robots or agents. More explicitly; we consider a group of agents who simultaneously search for the maze exit in a cooperative manner. In this scenario, agents share their knowledge of previously explored cells in the maze, helping other agents reach the exit. Clearly, the cooperative nature of the problem can greatly reduce the time required by the group of agents to reach the solution.

In our case, the field is a maze, defined as a two-dimensional grid of interconnected cells. The maze can be mathematically modelled as a graph, $G = (V, E)$, such that each cell in the maze represents a vertex in the set V and the obstacle-free corridor between two cells represents an edge in the set E. Each agent $a_1, a_2, \cdots, a_k$ starts from a position $s_i \in V$, and all agents seek to reach the goal position $g \in V$, [14]. we assume that the graph model of the maze is unknown and agents do not know the geometry and position of the obstacles and the way to exit. We assume that agents use their sensors to detect obstacles in the maze.

We propose a solution in which agents move simultaneously and independently through the maze following a set of pre-defined rules. After independently exploring a cell in the maze, each robot shares information with the rest of the agents in the team to help simultaneously map the maze and search for a solution. Previous solutions involving multiple agents with incomplete information, [35], assume that agents are only allowed to move one-at-the-time, and are limited to a one-way maze. We propose a novel searched-based algorithm using the concept of depth-first search algorithms [15] [37], and inspired in the algorithm proposed in reference [35]. Our solution is based on a modification of the algorithm in reference [35], that permits the independent and simultaneous maze exploration by the group of agents. This change is non-trivial, and can have a noticeable impact on the execution time of the maze exploration. Moreover, unlike previous studies, our algorithm does not require that agents start exploration at any specific point in the maze. Indeed, our solution is capable of handling the situation in which agents start exploring from different locations in the maze. Furthermore, our algorithm can be implemented either using computer simulation or a physical maze. Our approach is not confined to theoretical solvers as it is implemented in continuous time and not sequential or timestep-based solvers. However, the performance is measured over a predefined timeframe. In this work, the agent does not wait for other agents to do their tasks and continue their

exploratory duties. While previous solutions [27][35][36][34] assume the agent's movement is discrete, we, on the other hand, consider the agents' thinking and movement procedures to be part of the process. By pipelining each agent to a thread, we combined the programming idea of "threading" with the concept of multi-agent. As a result, the agents' orchestration is such that each thread's agents will carry out their responsibilities independently of other threads while contributing to a shared map. Finally, we emphasize a significant challenge in our approach's multi-agent maze solution problem, which has been overlooked in prior studies: agent conflicts! More precisely, how the agents should act and make decisions when interacting with one another and how they should communicate throughout these encounters.
A brief summary of the algorithm can be stated as follows:

> All agents start to search the maze using a depth-first search approach. Agents share the visited field and obstacles encountered to a global map, which is shared among all agents. After discovering the hidden destination, they work together to link the subfields they have investigated, and then they all proceed to the destination.

1.3 Thesis Outline

The rest of this thesis is organized as follows:

Chapter 2: This chapter covers the main concepts and terminology underlying the graph theory and algorithms used in this thesis.

Chapter 3: In this chapter, we define and introduce some baisc definitions regarding our maze and define the problem to be solved.

Chapter 4: In this chapter, we describe our algorithm.

Chapter 5: We discuss the implementation and explain how possible conflicts are resolved.

Chapter 6: In this chapter, we explain the testing phase, the hardware used to simulate our problem, and the solution in a real physical system.

Chapter 7: A summary and conclusion is provided along with research plan for future work.

Chapter 2

Graph Theory Preliminaries

In this chapter we present some basic definitions and results from graph theory that are needed throughout the rest of the thesis, including the fundamental definitions and algorithms of graph traversal, minimum spanning tree computation, and shortest path calculation.

2.1 Preliminary Definitions

A *graph* is a structure consisting of a set of vertices and edges. To define a graph, we must first specify the members of two sets: vertices and edges [38]. A graph can be represented as an ordered pair $G = (V, E)$ comprising:

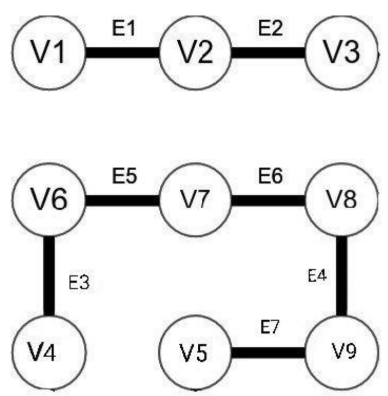


Figure 2.1: representation of a graph

- V: set of vertices (nodes, points).
- E: set of edges, unordered pairs of vertices two distinct vertices [39]. In Figure 2.1, the set V is $\{V1, V2, V3, V4, V5, V6\}$.

The set E is $\{E1 = (V1, V2), E2 = (V2, V3), E3 = (V4, V6), E4 = (V9, V8), E5 = (V6, V7), E6 = (V7, V8), E7 = (V5, V9)\}$.

- **Adjacent Node:** If and only if there is an edge between u and v, a node v is considered to be an adjacent node of node u.
- **Path:** A stream of edges that connects a set of vertices that are all distinct (and since the vertices are distinct, so are the edges) [40].
- **Cycle:** In a graph, a cycle is a non-empty path with only the start and ending vertices being equivalent[41].
- **Connected graph:** When every pair of vertices in a graph has a path between them, the graph is said to be connected. In a connected graph, no node is inaccessible [42].
- **Degree:** The number of vertices that are adjacent to a vertex determines its degree. A graph's degree is equal to the maximum of its vertices' degrees [39].
- **Empty Graph:** A set of vertices that do not have any edges between them [38].
- **Subgraph:** A subgraph G' of a graph G is a graph G' with vertex and edge sets that are subsets of the vertex and edge sets of G. G is said to be a supergraph of G' if G' is a subgraph of G [43].
- **Tree:** A graph that has exactly one path connecting any two vertices [41].

2.2 Graph Traversal

The term "graph traversal" or "graph search" refers to the process of visiting (investigating or updating) each vertex in a graph. These traversals are categorised according on the sequence in which they visit the vertices [39].

2.2.1 Depth First Search

The Depth First Search (DFS) algorithm explores a graph in a depth-first manner and utilizes a stack to record the next vertex to search for when an iteration encounters a dead end.

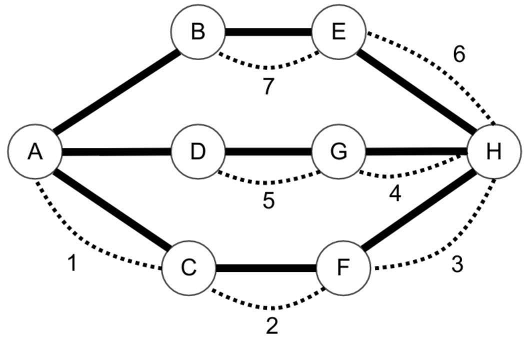


Figure 2.2: Depth-first traversal of a graph

As in Figure 2.2, the DFS algorithm goes from A to C to F to H to G to D, then to E, and finally to B. The following rules govern this process:

1. Visit a previously unvisited vertex adjacent to the current vertex. Indicate that it has been visited. Print it. Push it in a stack.

2. If an adjacent vertex is not found, a vertex from the stack is popped up. (It will pop up all the vertices in the stack that are not adjacent.)

3. Repeat Rules 1 and 2 until the stack is empty [44].

2.2.2 Breadth First Search

The Breadth-First Search (BFS) algorithm traverses a graph in a breadth-first manner and employs a queue to keep track of the next vertex to search for when an iteration encounters a dead end.

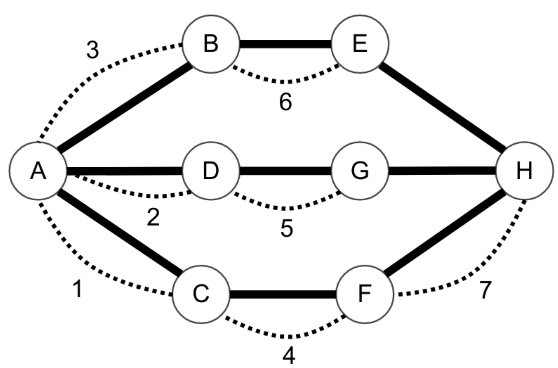


Figure 2.3: Breadth-first traversal of a graph

As in Figure 2.3, the BFS algorithm proceeds from A to C to D to B, then to F to G to E, and finally to H. The following rules govern this process:

1. Visit an unvisited vertex adjacent to the current vertex. Indicate that it has been visited. Print it. Add it to a queue.

2. Remove the initial vertex from the queue if no neighbouring vertex is found.

3. Repeat the first and second rules until the queue is empty.

At this point, there are no unmarked (unvisited) nodes remaining. However, the algorithm requires that we continue dequeuing to reach all unvisited nodes. The process finishes when the queue is empty [45].

2.3 Spanning Tree Problem

A spanning tree is a subgraph of an undirected graph that is a tree that contains all of the graph's vertices. Prim's algorithm [46] and Kruskal's algorithm are two main algorithms to identify the spanning tree in a graph. Now we will describe the Kruskal algorithm as it is utilized to generate mazes in our simulation.

2.3.1 Union-Find Algorithm and Disjoint Set Data Structure

A disjoint-set data structure manages a set of items partitioned into a number of distinct (non-overlapping) subsets. *Kruskal's algorithm* requires disjoint-set data structures for determining the graph's spanning tree. A *union-find algorithm* is an algorithm utilizing a disjoint-set data structure that conducts two operations on it:
Find: Identity which subset an element belongs to. This function determines if two items are members of the same subset. The *Find* operation searches the parent pointer chain from a given query node x to a root element. This root element denotes the set x belongs to and may also be x itself. The root element reached by *Find* is returned.
Union: Joins two subsets together to form a single subset. First, we must determine if the two subsets are members of the same set. Otherwise, we will be unable to execute union. $Union(x, y)$ replaces the set containing node x with the set containing node y. Union begins by determining the roots of the trees containing node x and node y using Find function. If the roots are the same, nothing further has to be done. Else, the two trees must be combined. This is accomplished by either setting node x's root's parent pointer to node y's or setting node y's root's parent pointer to node x's [47].

Figure 2.4: Creating 8 distinct sets in the beginning

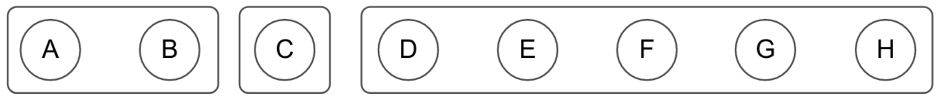


Figure 2.5: Following various *Union* operations, some sets are grouped together.

2.3.2 Kruskal Algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes an input graph and determines the subset of its edges that forms a tree that contains every vertex. The instructions below explain how to compute the minimum spanning tree using Kruskal's algorithm:

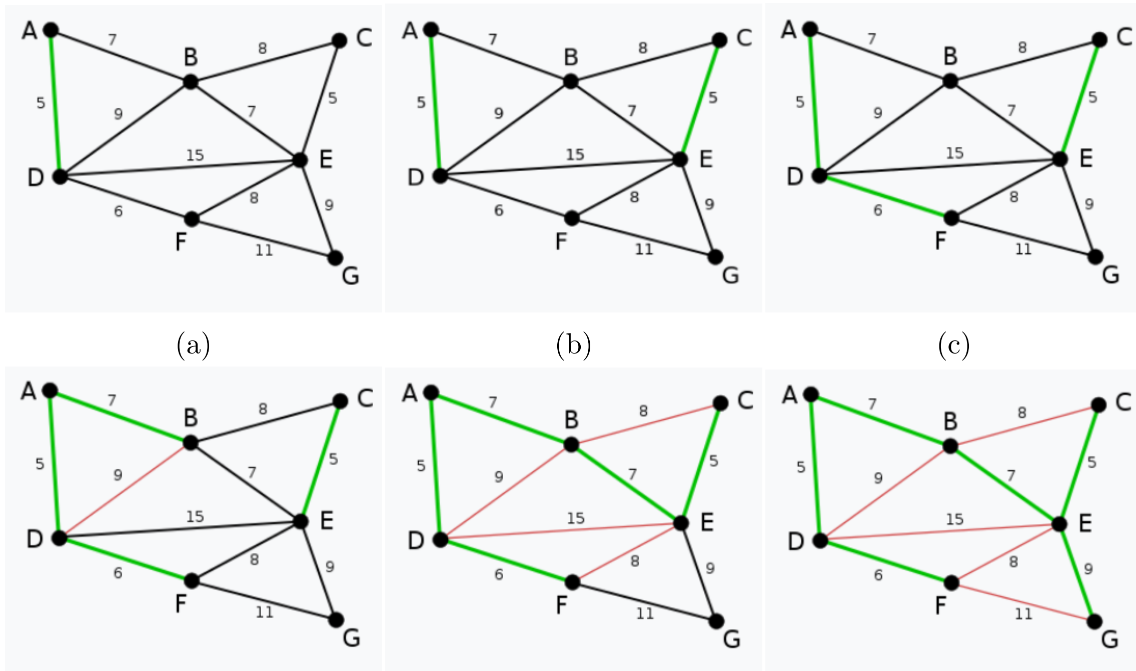
1. Build G (a *disjoint set* of trees), with each vertex in the graph representing a distinct tree.

2. Create a set S that contains all of the graph's edges. When S is not empty and G is not spanning yet:

   (a) Remove an edge with the smallest weight from S;

   (b) If the removed edge links two distinct trees, add it to the set G, so merging two distinct trees into a single tree using the *union* function.

At the algorithm's completion, the set G becomes the graph's minimum spanning tree. If the graph is connected, the set G has just one member and so forms a minimum spanning tree [48].



(a) (b) (c)
(d) (e) (f)

Figure 2.6: An example of the execution of the Kruskal algorithm [1]

2.4 Shortest Path Problem

The shortest path problem is a graph theory problem that involves finding a path between two vertices (or nodes) in a graph that minimises the sum of the weights of its respective edges. The *Dijkstra algorithm*, which calculates the shortest path from a particular vertex to all other vertices of the graph, and the *A* algorithm*, which calculates the shortest path between two specified vertices, are the two primary algorithms employed in this thesis to address the shortest path problem.

2.4.1 Dijkstra Algorithm

Dijkstra algorithm calculates the shortest path and distance between a source to all destinations in a graph given a starting node.

Assume that the node that we begin with is referred to as the starting node. Assume that the distance of node (A) is equal to the distance between the starting node and node (A). Dijkstra's algorithm will begin with infinite distances for each node and gradually improve them.

1. Declare all nodes to be unvisited. Create a set, named the unvisited set, that contains all the unvisited nodes.

2. Assign a *Distance* value to each node: *zero* for the starting node, and infinite for all others. The Distant value between two nodes v and u is the length of the shortest path identified so far between the nodes v and u. Since no path to any other vertex other than the starting node is known at the beginning (which is a path of length zero), all other *Distance* values are set to *infinity*. *Current* node is set to the starting node.

3. Consider all of the current node's unvisited neighbors and determine their *Distance* values via the *current* node. Comparing the newly computed *Distance* value to the currently assigned value, choose the smaller one. For instance, if the present node A is marked with a *Distance* of 7 and the edge connecting it to a neighbor B is marked with a length of 3, then the *Distance* to B through A will be $7 + 3 \quad = \quad 10$ If B was previously marked with a *Distance* greater than 10, it should now be marked with a *Distance* of 10. Alternatively, the current value will be remain.

4. When all of the *current* node's unvisited neighbors are considered, the *current* node is marked as visited and removed from the unvisited set. A node that has been visited will never be checked again.

5. When all nodes have been marked as visited or in case if the minimum distance between the starting node and a destination is needed, if the *destination* node has been marked as visited, the process will stop and the algorithm is complete.

6. Return to step 3 otherwise, selecting the unvisited node with the shortest *Distance* as the new *current* node [2], [49].

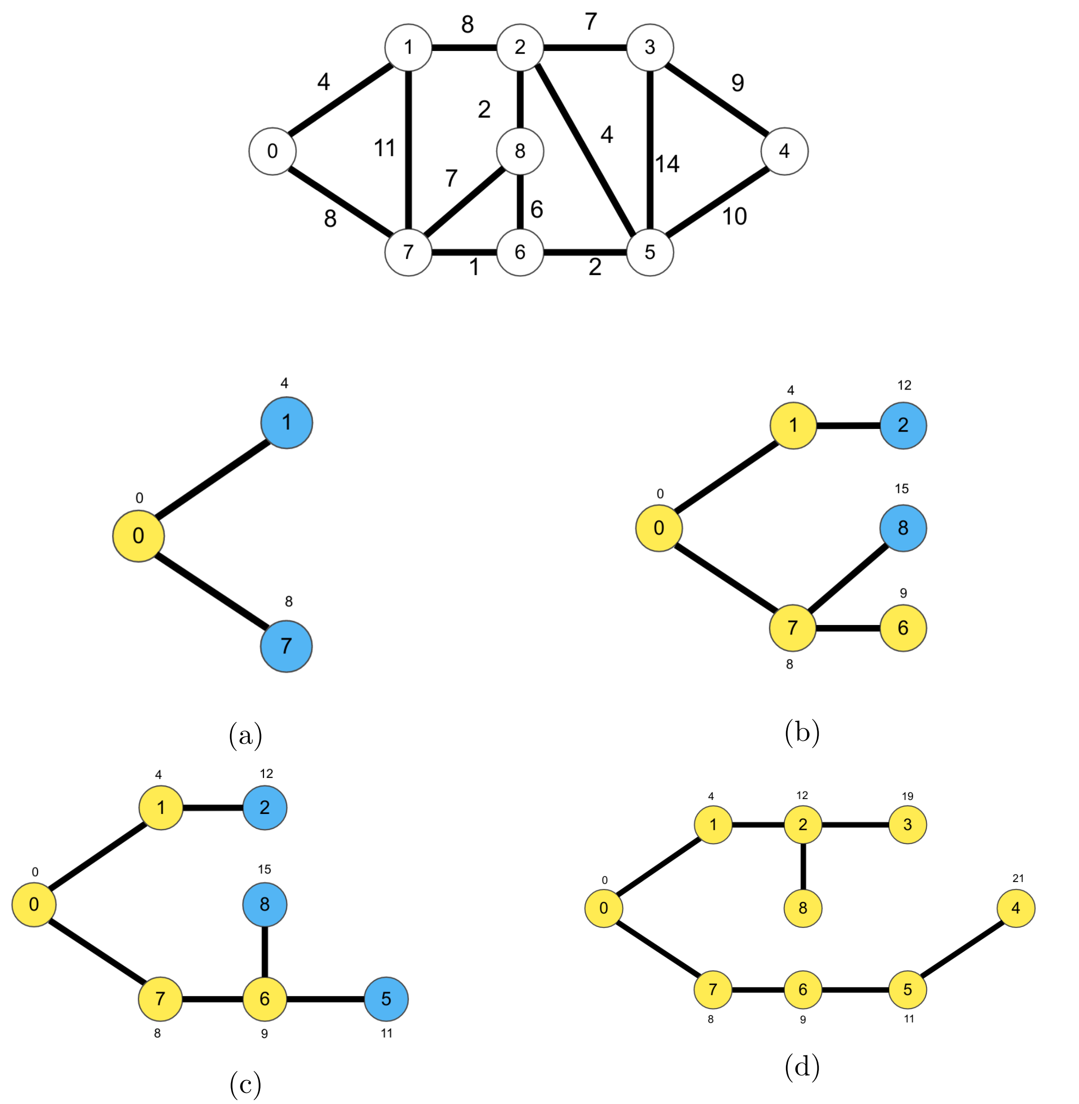


Figure 2.7: The procedure for running the *Dijkstra* algorithm on a given graph [2]

2.4.2 A* Algorithm

*A** is a search algorithm (pronounced "A-star") that attempts to identify the fastest path to a specified target node beginning from a defined starting node in a graph. The *A** algorithm obtains the optimal solution by computing the positions of all nodes between the beginning and ending nodes. Additionally, thanks to the heuristic function, it is quicker than *Dijkstra*'s algorithm [50].

$$f(n) = g(n) + h(n) \tag{2.1}$$

- $f(n)$: The cost of moving from the starting node to a particular node on the grid, following the produced path.

- $g(n)$: Distance between the current node and the start node.

- $h(n)$: The anticipated cost of travel from that specific node to the end destination. This is often referred to as the *Heuristic*, which is just a clever guess. We really do not know the distance until we find the exact path to the destination, since a variety of obstacles might stop our progress.

We copied the procedures for $A*$ algorithm from **Rachit Belwariar**'s Geekforgeeks article [3]. This is a comprehensive and clear explanation that is preferable rather than writing it again:

1. Create *open* List and *closed* List .
2. Initialize the *open* list.
3. Initialize the *closed* list, put the starting node on the *open* list (you can leave its f at *zero*).
4. While the *open* list is not empty:
   (a) Find the node with the least f on the *open* list, call it q.
   (b) Pop q of the *open* list.
   (c) Generate q's eight successors and set their parents to q.
   (d) For each *successor*:
      i. If *successor* is the *destination*: stop search!
      ii. $successor.g = q.g +$ distance between *successor* and q
      iii. $successor.h =$ anticipated distance from destination to *successor*
      iv. $successor.f = successor.g + successor.h$
      v. If a node with the same position as *successor* is in the *open* list which has a lower f than *successor*, skip this *successor*
      vi. If a node with the same position as *successor* is in the *closed* list which has a lower f than *successor*: skip this *successor*, otherwise: add the node to the *open* list.
   (e) push q on the *closed* list

We may use functions that estimate the distance between the current node and the destination as the heuristic function. The Manhattan distance and Euclidian distance are two heuristic functions that have been used in grid-like graphs [51], [3], [52].

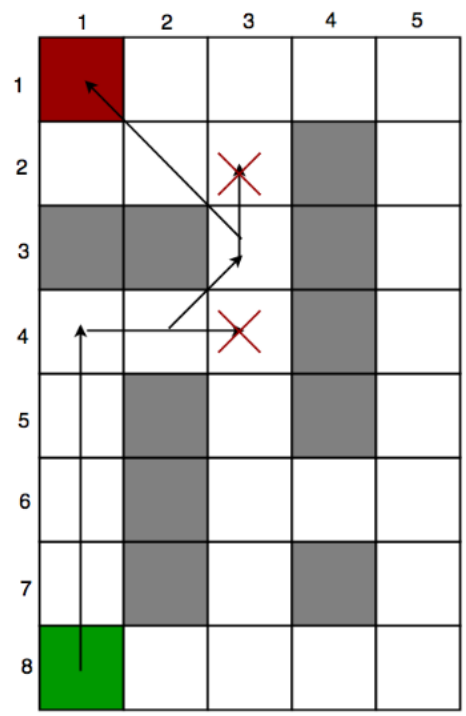


Figure 2.8: Overview of A* algorithm execution on a grid [3]

Chapter 3

Problem Definition

In this section we first introduce the concepts of *agent*, *maze*, and *goal* and then define the main problem to be solved.

3.1 Definitions

- **A**n agent is a part of the environment who is able to make decisions regarding the states of the environment and is able to cooperate, communicate and adapt to the environment and other agents [14].

  We will assume throughout that each agent is equipped with distance sensors that allow the detection of adjacent obstacles. Moreover, we also assume that the agent's location is globally known. A roof-mounted camera is used to detect the location of each agent.

- **Cell:** A cell is a bounded elementary compartment in the two-dimensional space. Two adjacent cells may be connected, thus allowing traffic flow of agents between adjacent cells, or may be separated by an *obstacle*. An obstacle is a wall between two adjacent cells. Walls are also used to define the maze's boundary. Each cell is exclusive, *i.e.* can only be occupied by a single agent at each time.

- **Maze:** A maze can be described as a two-dimensional grid of interconnected *cells*. A maze can be mathematically modelled as a graph, $G = (V, E)$ (Figure 3.1) such that each cell in the maze represents a vertex in the set V of the graph. If two adjacent cells are connected and not separated by an obstacle, we say that the path between two adjacent cells represents the edge in set E in the graph. Clearly, if there is a path between two adjacent cells, then there is an edge between the corresponding vertices of these two cells.

A graph is said to be *connected* if there is a path connecting each vertex. A path on a graph such that the only repeated vertices are the first and the last vertices is called a *cycle*. Finally, a *tree* is a connected graph without cycles. When a graph consists of a single tree then there is a single path between any two vertices.

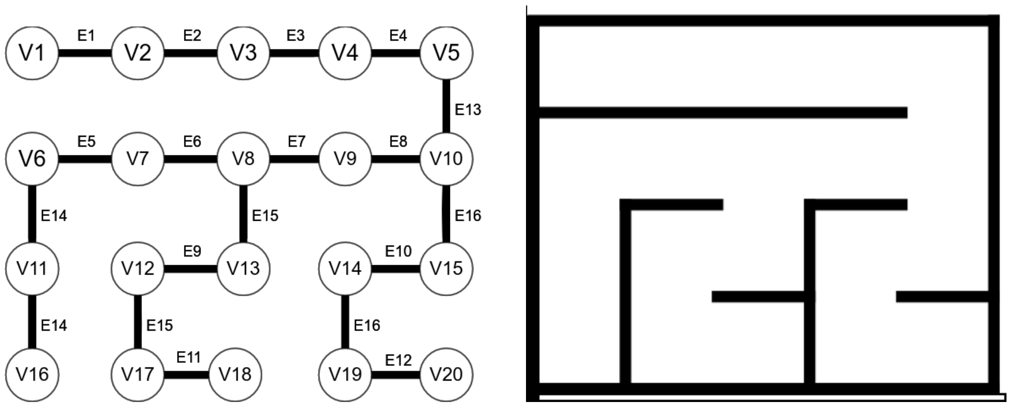


Figure 3.1: Graph representation of a maze

- **Goal:** The goal (or destination) is a specific cell in the maze (respectively, a vertex in the graph G representing the maze) that all agents have to reach.

  We will assume throughout that our maze contains only one goal. At the beginning of the maze exploration, the location of the goal is unknown to all agents. Theoretically, we assume that the goal contains information representing its nature. This information can only be identified when an agent visits the goal for the first time during the exploration process. Furthermore, we assume that after the first visit, the visitor agent raises a flag, and the row, column, and location of the goal in the shared map with all other agents. In

3.2 Problem Statement

We can now define the main problem to be solved:
We consider an $m \times n$ grid maze and k agents, distributed around the maze and occupying different cells. The graph $G = (V, E)$ represents the maze cells and is shared between the agents. Before the exploration begins, V is unknown, and $E = \emptyset$, *i.e.* E is the empty set. Our main objective is to implement a hierarchical decision-making process for each agent, such that the group of agents can collaboratively construct the graph G by exploring the maze and sharing the information with other agents until all agents reach the goal cell.

We assume that the graph G may not be limited to a single *tree* and may contain cycles. We also assume that all agents are moving simultaneously and independently of others and can communicate with others by contributing to the formation of the graph representation of the maze, information that is shared online between agents.

Chapter 4

Proposed Solution

We begin by describing the *shared map*, which contains the information shared by the robots as they move through the maze. We describe the nature of the shared map as a graph $G = (V, E)$.

Recall that the graph $G = (V, E)$, is formed by two sets, namely, the set V that represents the set of vertices and set E that represents the edges of the graph G. Before beginning the exploration both sets are unknown to the agents and the set E is initialized as the empty set $E = \emptyset$. The set E is constructed using information received from the agents using the algorithm described in the next section. The information gathered by the agents for each cell consists of the following elements: *row*, *column*, *color*, and *OC flag*. We will refer to this information as the *attributes* of the elements in the set V in the *shared map*. Each agent maintains a real-time copy of the shared map and the number of times it visited each vertex. We now describe the attributes of the elements in the shared map. Our use of cell colours in this section is inspired in the Depth-first search in graph theory terminology [15].

- **White Cell:** A white cell is a cell that has not yet been visited by any agent. Before the exploration begins, all vertices representing the cells in the shared map are marked as *white* cells.

- **Gray Cell:** A gray cell is a cell that has been visited by an agent at least once. The color of the corresponding cell in the shared map is changed from *white* to *gray* as an agent enters this cell. Each gray colored cell visited by an agent is logged in the agent's memory

- **Black Cell:** A *black cell*, or *dead-end* cell, refers to a cell that is surrounded by either obstacles or other black cells in three of its boundaries. Thus, a black cell is a cell with only one way to enter or exit from this cell. The color of a vertex in the shared map can only be changed to black when an occupying agent determines that three of its boundaries are blocked by other black cells or obstacles. Other agents are not permitted to enter the black cell for the remainder of the exploration.

- **OC flag:** An *OC flag*, or *occupation flag*, is a flag associated with each vertex used to represents when vertex in the shared map is occupied by an agent. The *OC Flag* is a binary flag with two possible values, either *True*, which indicates that the respective cell is occupied, or *False*, which indicates that the cell is empty. In the beginning of the exploration, the *OC flag* for all vertices is set to *False*. When an agent move into a cell, the OC flag is changed to *True*. When the agent leaves the cell and enters an adjacent cell, the OC flag corresponding to the vertex of the initial cell is reverted back to False. Agents are not permitted to enter a cell whose OC flag is labeled True. Figure 4.1 describes the effect of a robot moving between two cells and the OC Flag.

We will use the term *exploration* when referring to an agent that is sensing the adjacent obstacles of a cell with the intention of contributing information to the shared map. Notice finally that an agent can enter a cell only if the following conditions are satisfied: (i) the OC flag of the corresponding vertex is set to *False*, (ii) the corresponding vertex in the shared map is either *white* or *gray*, and (iii) a connecting edge to the corresponding vertex is an element of the shared map, *i.e.* such element has been previously entered into the shared map.

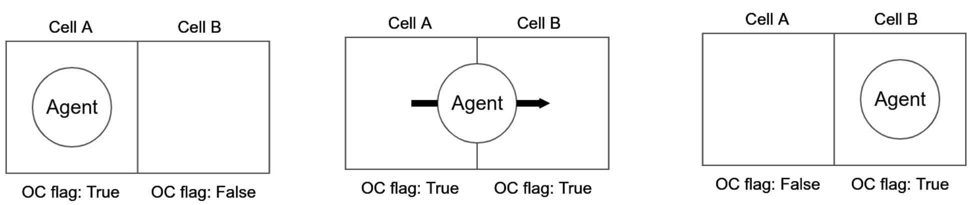


Figure 4.1: Transition from cell A to cell B by an agent and how it affects the value of $OCflag$. Transition can take time depending on the moving speed of the agent

4.1 Algorithm Statement

Our algorithm follows the same approach as reference [35]. The algorithm in [35], however, solves the problem *sequentially*, assuming that only one agent is allowed to move at any given time, in order to avoid prossible conflicts between agents. A second limitation of the algorithm in reference [35] is that it is limited to a one-way maze, *i.e.* mazes that can be represented as *tree*. Our algorithm removes these limitations.

Our approach is such that all agents can move simultaneously and can also be applied to rectangular mazes. We divide our solution into two phases:

1. **Phase One:** The goal cell has to be found by an agent.
2. **Phase Two:** Moving all of the agents to the goal cell.

4.1.1 Phase One: Finding the goal cell

Each agent goes through the following rules in its exploration process:

1. At any arbitrary state, an agent visiting a cell identifies the obstacles in all four directions using its own internal distance sensors and contributes their corresponding values to the shared map. This means that the edges are identified and added to the set E in the graph representation of the shared maze map. Recall that the graph representation of the maze is $G = (V, E)$ where the set V is the set of vertices and set E is the set of edges. The agent makes a connection between all adjacent cells in its subgraph.

2. If the agent identifies an adjacent cell as a *white cell*, then it moves into the adjacent cell provided that the *OC Flag* of the adjacent cell is *False*. If there is more than one adjacent cell. The agent stores white cells that are not selected in its memory for future use.

3. If the agent identifies an adjacent cell as a *gray cell*, then it moves into the adjacent cell provided that the *OC Flag* of the adjacent cell is *False*. If there is more than one adjacent cell, then the agent moves into the cell that has been visited the fewest times by the same agent. If there is more than one cell that has been visited the same number of times, the agent chooses one arbitrarily.

4. In the next level of priority, if an agent identifies an adjacent cell as a a *non-black* cell with and *OC Flag* set to *True*, then the agent shall wait at its current location until the agent occupying the *non-black* cell, reverts the *OC Flag* to *False*. Once the *OC Flag* reverts to *False* the waiting agent will move into the adjacent cell.

5. If there exists only one way to exit from the current cell and all other directions are obstacles or black cells, *i.e.* there is only one edge connected to the vertex representing the current cell in the shared map, then the agent shall change the color of the current cell to *black*.

6. The agent moves to the chosen cell following the instructions stated in rules 2 -5 and (i) changes the color of the current and next cells and (ii) changes the *OC Flag* of both cells according to the the instructions stated in Section 3.1 (see Figure 4.1).

7. Repeat instructions starting in rule 1 until an agent finds the *goal cell*. Once an agent finds the *goal cell* the location of the *goal cell* is uploaded to the *shared map*.

4.1.2 Phase two: Leading all agents through the goal cell

Assume now that an agent has reached the *goal cell*. Once an agent reaches the *goal cell*, the information is shared with other agents using the *shared map*. The main problem at this point is that other agents may or may not be able to reach the *goal cell* with the information available on the shared may, depending on whether or not they find a direct path to the *goal cell*.

This situation can be explained more precisely using graph theory: Recall that visited cells by an agent form an undirected graph in which each cell represents a vertex in the *shared map* (Figure 4.2). Recall also that a *Connected Graph* is a graph in which there exists a path through existing edges between each two cells [37]. If the graph is not connected, then it can be represented as a set of disconnected components or subgraphs. As an example, Figure 4.2 represents a graph of the visited cells by 3 agents and the goal cell has been detected by agent 1. At this state, other agents cannot stop the exploration and proceed to the *goal cell* through the existing (*i.e.* visited) edges, unless they find an edge through the connected component that contains the *goal cell*.

As soon as an agent finds a connecting edge, the two connected components merge and form a single connected component (Figure 4.3). At this state, there exists a path from the current cell of the agent and the *goal cell* since the agent and the *goal cell* are in the same connected component.

With the explanation of Phase Two provided above, we can now proceed to describe the algorithm for this phase:
For each agent not in the *goal cell*'s connected subgraph:

1. Sort all *white cells* in the agent's connected subgraph according to the sum of the Euclidian distance to the *goal cell* and the length of the path from the

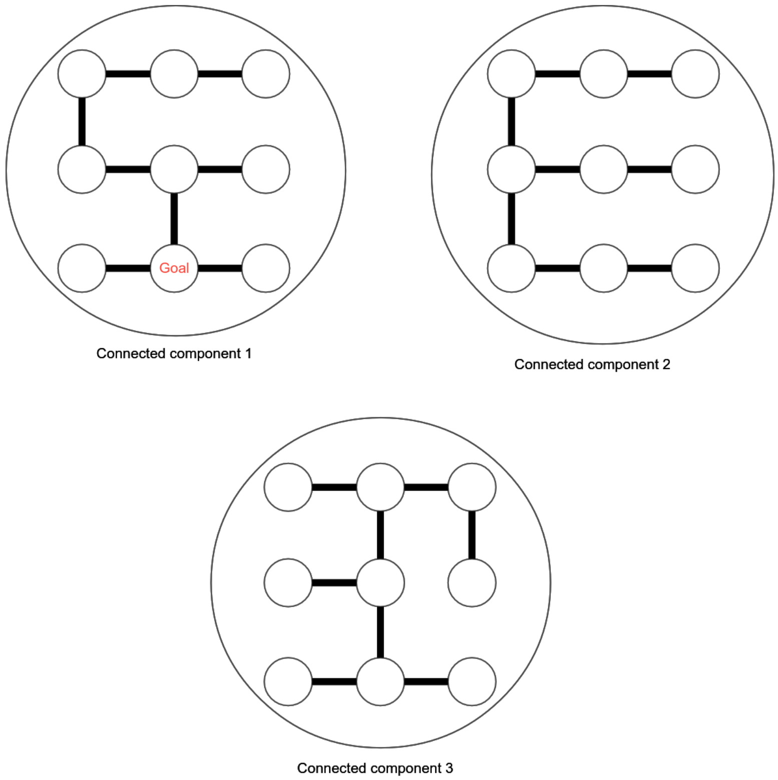


Figure 4.2: Graph representation of the traversed cells by each agent

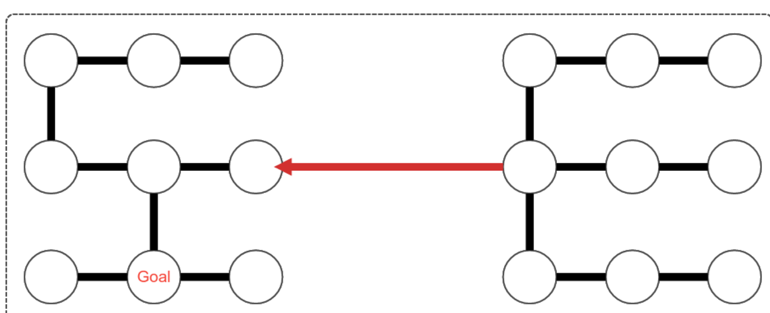


Figure 4.3: Merged connected components

current cell, and choose the one with minimum distance. If more than one cell with the same minimum distance exists, arbitrarily select one.

Notice that the sorted white cells are the ones whose connecting edge to the agent's connected subgraph has been entered to the shared map, but they have not yet been visited. Recall that in phase one of the solution, the first rule states that the agent senses the adjacent obstacles and contributes the adjacent edges to the shared map. Moreover, the second rule states that if there is more than one white cell, then the agent chooses one arbitrarily. Hence, some white cells may remain unvisited, eventhough their connecting edge is available in the shared map. These cells are primarily border cells of the connected subgraph.

2. Find the shortest path toward the chosen cell using the *A* algorihm* [51] and follow the path until reaching the chosen cell. The *A* algorihm* is an algorithm that optimally finds the shortest path between two arbitrary vertices in a connected graph [51].

3. After exploring the closest white cell, if the agent's subgraph has not become connected to the *goal cell*'s subgraph, repeat the instructions from the beginning.

4. Find the shortest path towards the *goal cell* using the *A* algorithm* and follow the path until reaching the *goal cell*. Disappear when reaching the *goal cell*.

After execution of the algorithm, all visited cells form a one-component connected graph. By selecting a heuristic function subject for sorting the *white cells* in phase two, we were inspired by the A* algorithm, which we discussed in Chapter 2. The function is equal to the sum of the *Euclidian distance* between the selected white cell and the *goal cell* and the length of the path between the agent's current cell and the chosen *white cell*. This function may be modified by solving an optimization problem that minimizes the time spent in phase two over several simulations, which is not the focus of our study. But our work provides an starting point to this problem. Algorithm 1 and Algorithm 2 provide the pseudo-code for programming the thread for both phase one and two.

Algorithm 1: Thread Function for Phase One

```
Result: Finding the goal cell by an agent
initialization;
while Goal cell not finded do
    white_cells, gray_not_visited_cells, gray_visited_cells ← [ ];
    obstacle_count ← 0;
    for each [left_cell, up_cell, right_cell, down_cell] do
        if obstacle exist then
            increment obstacle_count by 1;
            disregard cell and continue the while loop;
        end
        if OCflag is True then
            disregard cell and continue the while loop;
        end
        if Cell.color is white then
            append the cell to white_cells;
            add the white cell to the agent's subgraph
        else if Cell.color is gray then
            if cell is in agent history then
                append the cell to gray_visited_cells;
            else
                append the cell to gray_not_visited_cells;
            end
        end
    end
    if obstacle_count is 3 then
        change the cell color to black and make all directions dead end
    end
    if white_cells is non-empty then
        choose a random cell from white_cells as the next_cell;
    else if gray_not_visited_cells is non-empty then
        choose a random cell from gray_not_visited_cells as the next_cell;
    else if gray_visited_cells is non-empty then
        choose a cell from gray_visited_cells which is visited less times as the
          next_cell;
    else
        stay in your current location;
    end
    Move to next_cell and assign OCflag
end
```

Algorithm 2: Thread Function for Phase wo

Result: Finding the goal cell by an agent

initialization;

while *the agent's subgraph is not connected to the* goal cell*'s subgraph* **do**

    choose the *white cell* with the lowest sum of the *euclidian distance* to the *goal cell* and path length from the agent's *current cell*;

    follow the path toward to chosen *white cell*;

    for *each* $[left\_cell, up\_cell, right\_cell, down\_cell]$ **do**

        if *obstacle does not exist* **then**

            connect the agent's subgraph to the selected *white cell.*

        end

    end

    remove the explored *white cell* from the agent's memory;

end

Chapter 5

Implementation

In this chapter, we discuss the implementation of our algorithm on a simulation or a physical system containing multi-robots.

In general, most path planning algorithms encountered in the literature [10] [11] [35] [53] use a *sequential* algorithm, *i.e.* one in which commands are executed one by one (Figure 5.1). In this scenario agents cannot move simultaneously, *i.e.* whenever an agent is moving all other agents must wait. Although effective, this type of algorithm results in slow solutions of the maze. Our main goal is to complete the algorithm

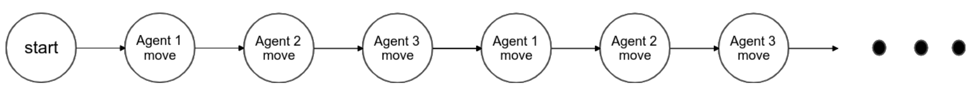


Figure 5.1: Linear execution of the algorithm

in a non-sequential manner, *i.e.* executing the algorithm in such a way that all agents are allowed to move simultaneously. To accomplish this objective we define a *thread* function associated with each agent and execute the algorithm in all threads in parallel rather than sequentially (Figure 5.2). Each agent executes a script in a thread, resulting in simultaneous execution of all threads.
Meanwhile, these threads are collecting (i) sensor data published by the agent's distance sensors for obstacle detection and (ii) location data published by the global camera, both in realtime. All threads start to execute at approximately the same time. A short delay of a few milisecond exists between each threads in order to avoid conflict at the beginning of the execution. These conflict is discussed in more detail in Section 5.1.

Figure 5.3 shows the overall simulation concept. For each agent, two processes are run concurrently, namely, an *agent* process, and a *thread* process. Each process consists of

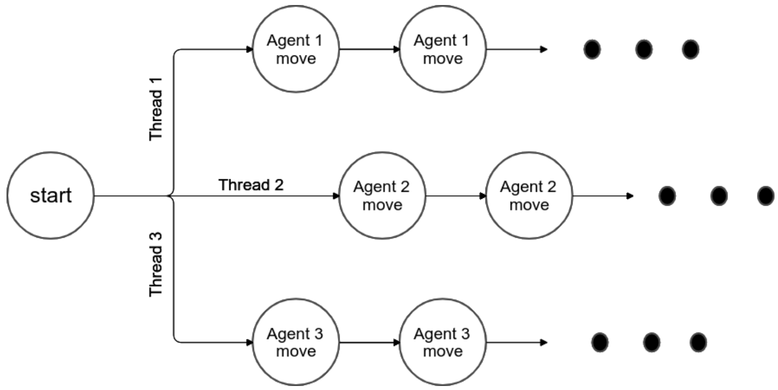


Figure 5.2: Parallel execution of the algorithm in seperate threads for each agent

a set of instructions executed independently from other processes. For each agent, we have the following combination: the *Agent* process publishes the sensor data obtained using the distance sensors. The *Thread* process collects sensor data and also the location of the robots obtained from *camera*. Then, following the instructions in the algorithm, the *thread* process contributes the adjacent edges to the *shared map* and chooses the next cell to be accupied. The results are published as a *move* command to the *agent* process. Simultaneously, the *shared map* collects all contributions from the *thread* processes of all agents, combines them into a single map, and feedbacks the information to all *thread* processes.

5.1 Agent Conflicts

An agent conflict can occur when two agents decide to move to the same cell concurrently. According to the algorithm, as soon as an agent chooses to move into a cell, it changes the *OC flag* to *True*, thus preventing other agents from choosing this cell as long as this flag remains *True*. A conflict can, however, occur at the start when all agents start their threads simultaneously. The situation can avoided by inserting a small ϵ delay time between each thread start.

Also, our algorithm is free of edge conflict and swapping conflict in the sense of reference [54].

Notice also that our algorithm is free of Vertex conflict in the sense of the article [54] as time is continuous in our problem and not sequential, the chance of occurrence of this conflict is nearly zero. As soon as an agent chooses to move into a cell, it changes the *OC flag* to *True*, hence, another agent cannot choose this cell as long as this flag is *True*. There is a chance for this conflict to occur when the algorithm starts as all agents start their threads simultaneously. We can avoid this situation by inserting

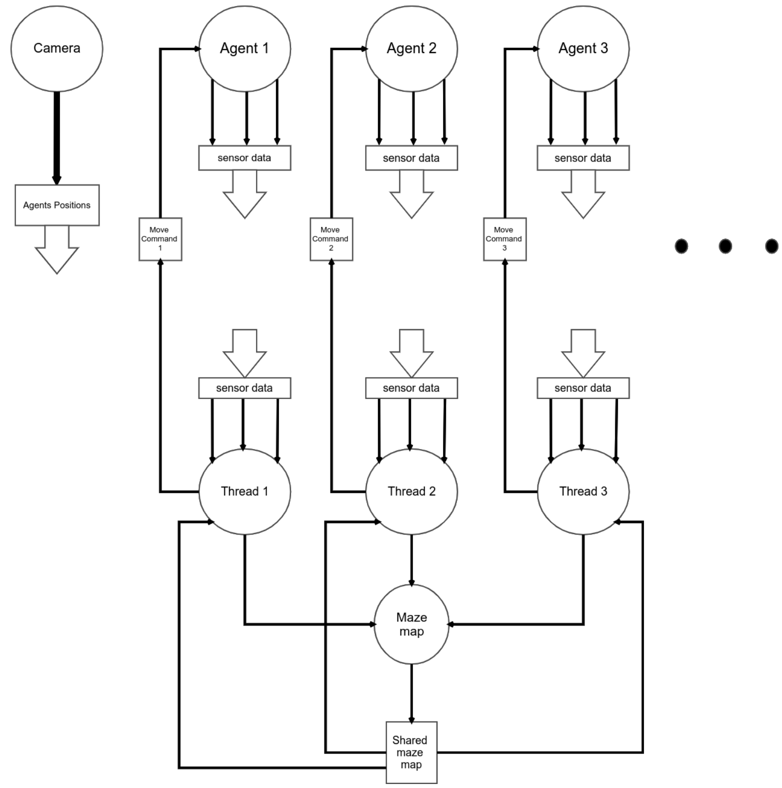
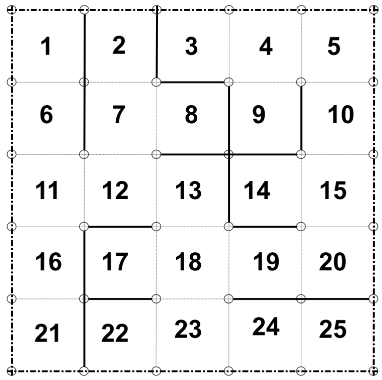


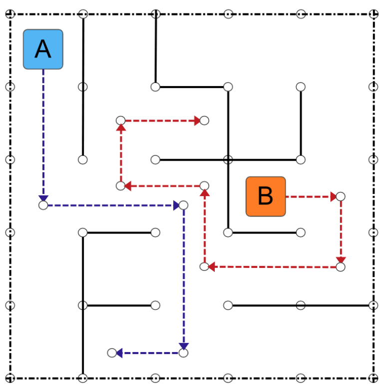
Figure 5.3: Diagram of control system implementation

a small ϵ delay time between each thread start. Also, our algorithm is free of edge conflict and swapping conflict in the sense of reference [54].
During phase two of our algorithm, agents' conflicts can occur. For more enlightenment, in rule 2 of phase two, the shortest path toward the chosen *white cell* in rule 1 is calculated. In a particular case, any two agents can have an intersection in their paths toward the *(*white cell) and may reach the intersection simultaneously. In this case, agents switch their paths as each agent proceeds toward the mutual *white cell*; hence, both white cells are explored, and conflicts in the intersection is avoided.

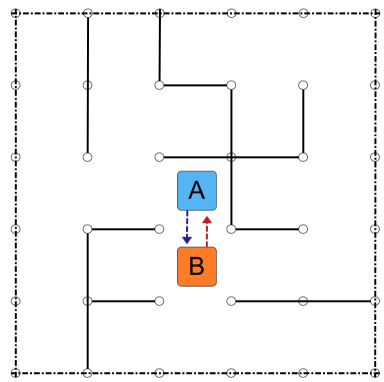
For instance, Figure 5.4 illustrates a sample of a hypothetical conflict. Assume that Figure 5.4a represents an entire maze or a section of a larger maze. Each cell has been allocated a number to simplify navigation. Assume that *agents* A and B are in phase two of the algorithm and are exploring the maze. *Agent* A has picked cell 22 as the *white cell* to explore, whereas *agent* B has chosen cell 8 as the white cell to explore. *Agent* A must follow the blue-colored trajectory shown in Figure 5.4b, whereas *agent* B must follow the red-colored trajectory presented in Figure 5.4b. Their trajectories cross in cells 12, 13, and 18, implying that the agents may collide along these lines. The nature of our algorithm is not dependent on each agent, and the major interest is the collaboration and behavior of the agents operating in teams. Thus, cells 22 and 8 must be explored independently of the agent that investigates them throughout this procedure. Assume that when *agent A* reaches cell 13, *agent B* reaches cell 18, they cannot continue and collide (Figure 5.4c). In this case, they swap destinations and decide to continue on their mutual paths. As a result, *agent A* follows the blue-colored track, whereas *agent B* follows the red-colored trajectory 5.4d. In conclusion, *agent A* and *agent B* choose the blue and red colored paths, respectively (Figure 5.4e).



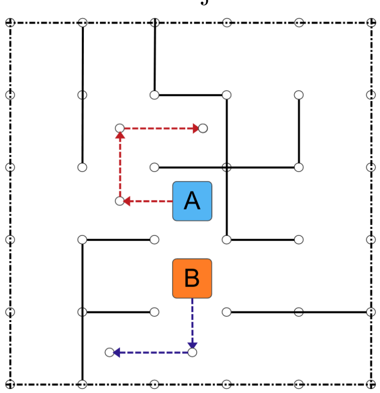
(a) An entire maze or a section of a larger maze. Each cell has been allocated a number



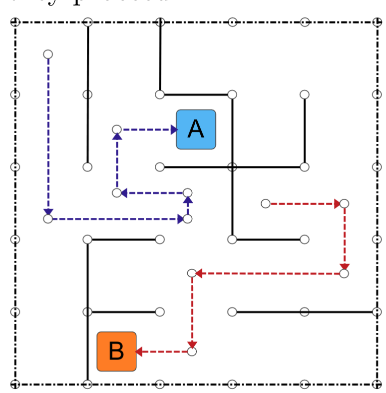
(b) *Agent A* and *Agent B* choose their trajectories



(c) Two agents may collide if they proceed



(d) Two agents swap their trajectories



(e) Overall trajectories of the agents

Figure 5.4: An overview two agents that have overlapping trajectories and how to avoid collision

Chapter 6

Testing

To test the proposed solution and monitor its performance, the solution is ran on a simulation with mazes of different sizes and varying the number of agents. To verify the algorithm's performance in the real world, the results are then tested on a physical system with three mobile robots as the agents, a maze-like field, and a floor-mounted camera. We begin by discussing our method for creating random mazes and then test our algorithm on this generated maze. We begin by describing our maze generating method, which we will use to evaluate our proposed algorithm.

6.1 Random Maze Generator Algorithm

This section describes the random maze generating algorithm that was utilized to conduct a simulation of our proposed algorithm. This algorithm is inspired by the Kruskal algorithm explained in Section 2.3.2. So the process is nearly the same unless it has been executed on a maze. We covered the disjoint sets data structure and the union-find algorithm in Section 2.3.1. This data structure stores the generated maze's nodes, and the union function is employed throughout the maze generation procedure. This method takes the maze's dimensions (i.e., rows and columns) as input. It produces a maze in which all cells are dead-ends, and each cell's four directions lead to an obstacle (Figure 6.1a).
The instructions needed to generate the maze are:

1. Create the maze's representation graph, assuming that each vertex in the graph corresponds to a cell in the maze and that each connecting edge between two vertices corresponds to a path linking the two cells represented by the two vertices.

2. Define a state variable, *Connected Components*, which specifies the number of

connected subgraphs included inside the corresponding graph. This variable's initial value equals the maze's row multiplied by the maze's column.

3. While the *Connected Components* is greater than 1:

   (a) Choose a cell in the maze and one of its neighbors in four directions at random. Assuming they are named cell A and cell B.

   (b) If there is no obstacle between cells A and B: disregard them and repeat the *while* loop.

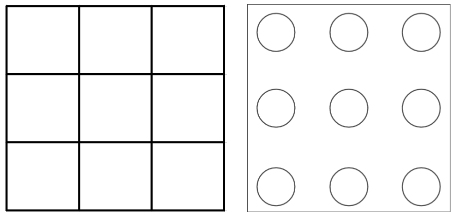
   (c) If cells A and B are connected subgraphs, which means that the results of the *Find* function on both of their corresponding graph vertices are the same: disregard the two cells and repeat the *while* loop.

   (d) Eliminate the obstacle between A and B. Add an edge connecting the corresponding vertices of cells A and B in the graph.

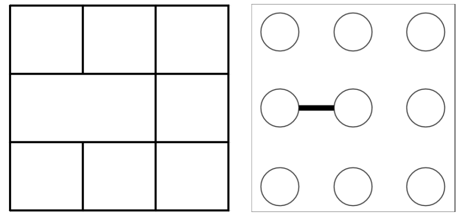
   (e) Execute a *Union* function using the corresponding vertices of A and B as input.

   (f) Decrease the number of *Connected Components* by *one*.

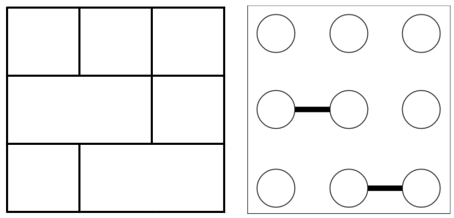
Figure 6.1 showsthe process of generating a 3×3 maze.
The instructions above enable us to generate a one-way maze with a path connecting every two cells. Thus, regardless of the agent's starting point, a path exists between it and any arbitrary destination. However, this approach produces a maze whose corresponding graph is a tree. As a result, there is only one path between each node in this graph, and it does not include any cycles. We may be considering upgrading the maze to a multi-way maze. This approach might be accomplished by assigning a probability to any obstacles remaining in the maze after running the *maze generator* algorithm, generating a random variable between 0 and 1, and eliminating the obstacle if the random variable was less than 0.5 (Figure 6.2).



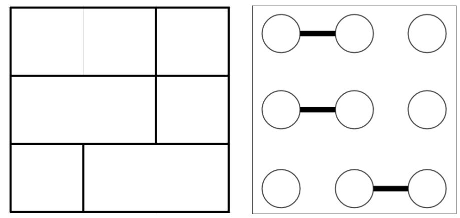
(a) Plain maze with all dead-end cells. Connected components: 9



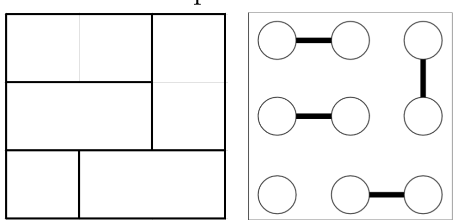
(b) First obstacle removed. Connected components: 8



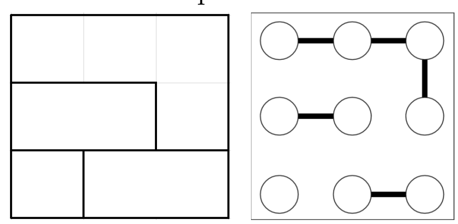
(c) Second obstacle is removed. Connected components: 7



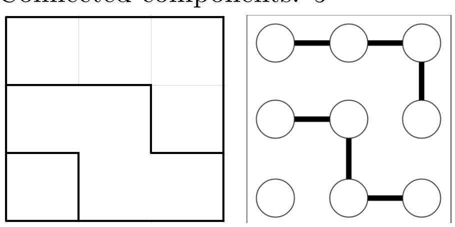
(d) Third obstacle is removed. Connected components: 6



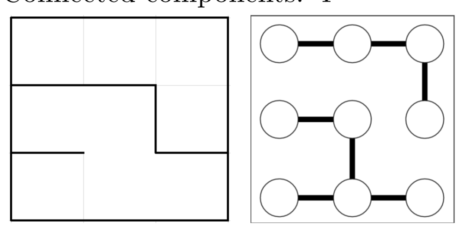
(e) Fourth obstacle is removed. Connected components: 5



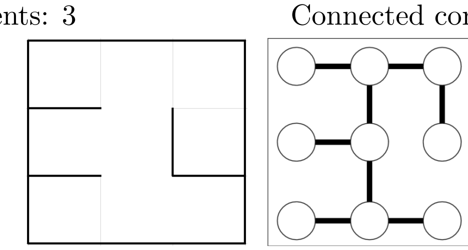
(f) Fifth obstacle is removed. Connected components: 4



(g) Sixth obstacle is removed. Connected components: 3



(h) Seventh obstacle is removed. Connected components: 2



(i) Eighth obstacle is removed. Connected components: 1

Figure 6.1: Execution of Section 6.1's maze generating algorithm on a 3×3 maze.

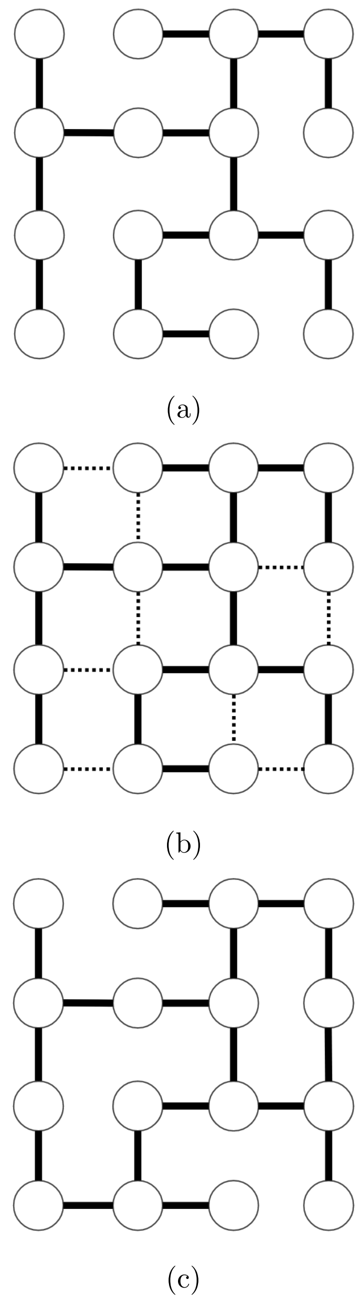


Figure 6.2: A sample of a multi-way maze generation from a one-way generated maze by assigning the probability of 75% to the remaining obstacles

6.2 Simulation and Results

In this subsection we present our computer simulated results. To verify our algorithm we consider a computer generated square-shaped maze with the following dimensions: $\{5\times5, 10\times10, 15\times15, 17\times17, 20\times20, 25\times25\}$ cells. In each case, we run our algorithm assuming $\{1, 3, 5, 7\}$ agents.

To evaluate the performance of each solution and compare the results, the term *timestep* is defined as the time it takes for an agent to move from its current cell to an adjacent cell. To compare the results for phase one of the solution, our benchmark is the number of timesteps taken for the first agent to reach the destination cell from the starting point. The benchmark for the second phase is the number of timesteps needed by the last robot to reach the destination. To remove the effects associated with random bias, the simulation is run several times for each case and the average results are used. Tables 6.1, 6.2, 6.3, 6.4, 6.5 and Figures 6.3, and 6.4 illustrate the results.

As seen in the test results, the case that involved only one agent took a significantly longer time than other cases with multiple agents and improved communication explained in both phase one and phase two of the algorithm. Our results are significantly better than those of reference 35. For mazes with dimensions of 10×10, 15×15, and 20×20, reference [35] values for seven agents are about 70, 125, and 200 timesteps, respectively, whereas our results for seven agents are 49.6, 99.15, and 166.1 timesteps. Generally, timing is improved as the number of agents increases. However, for the smaller mazes, like a 10×10 maze, increasing the number of agents may have an opposite effect on timing. For better enlightenment, it can affect the timing in phase two, as more agents need to be guided to the destination. In smaller mazes, overlapping between agents' subgraphs is more probable. Hence, two agents, who have their subgraphs overlapped, can spend timesteps through their mutual subgraphs while not exploring newer white cells, which is critical for exploring the maze and finding the goal cell as soon as possible. We restate that in rule number 2 of phase one of our algorithm, choosing a *gray cell* to move, the *gray cell* that has been visited fewest times by the same agent is prioritized. This rule can minimize the time spent for any two agents with overlapping subgraphs in their mutual subgraphs. The influence of increasing the number of agents with respect to the increases in the dimension of the maze is inferred from the results. Nevertheless, increasing the number of agents is only possible with using more resources. Thus, as illustrated by the results, optimal decision-making considering the resources and timing is suggested here. Figures 6.6 illustrate the test's timeline of events for a sample simulation ran for the randomly generated maze in Figure 6.5. It can provide us with a detailed report of what occurred throughout the test.

Table 6.1: Average number of timesteps needed to reach the goal cell for the first time (Phase One)

	Number of Agents			
Maze Dimension	1	3	5	7
10 x 10 cells	88	26.2	27.8	17.27
15 x 15 cells	249	77.6	49	35.64
17 x 17 cells	299.64	98.9	58.1	45.8
20 x 20 cells	325	126.3	88	71.82
25 x 25 cells	514.47	267.7	176.8	135.9

Table 6.2: Average number of timesteps needed for the last agent to reach the destination (Phase Two)

	Number of Agents			
Maze Dimension	1	3	5	7
10 x 10 cells	88	42.6	45.4	49.55
15 x 15 cells	249	103.2	107.8	99.15
17 x 17 cells	299.64	177.9	143.16	124.7
20 x 20 cells	325	234.3	188.6	166.1
25 x 25 cells	514.47	311.7	297.1	254

Table 6.3: Minimum and maximum number of timesteps recorded in the test results

		Number of Agents			
Maze Dimension		1	3	5	7
10 x 10 cells	phase 1	2/130	9/37	11/46	1/43
	phase 2	-	19/56	31/58	21/140
15 x 15 cells	phase 1	118/444	3/177	1/157	2/104
	phase 2	-	34/185	36/342	23/348
17 x 17 cells	phase 1	8/568	16/285	1/211	5/100
	phase 2	-	60/372	49/522	73/320
20 x 20 cells	phase 1	195/417	4/321	21/123	1/186
	phase 2	-	23/594	135/225	74/372
25 x 25 cells	phase 1	24/1240	40/396	22/399	14/268
	phase 2	-	23/594	116/410	183/368

Table 6.4: Median of the number of timesteps recorded in the test results

		Number of Agents			
Maze Dimension		1	3	5	7
10 x 10 cells	phase 1	109	30	22	14
	phase 2	-	45	46	47
15 x 15 cells	phase 1	163	43	48	24
	phase 2	-	86	99	94
17 x 17 cells	phase 1	327	83	50	46
	phase 2	-	145	130	103
20 x 20 cells	phase 1	344	92	104	59
	phase 2	-	227	197	156.5
25 x 25 cells	phase 1	565	299.5	154	150
	phase 2	-	334.5	325	249

Table 6.5: Standard deviation of the number of timesteps recorded in the test results

		Number of Agents			
Maze Dimension		1	3	5	7
10 x 10 cells	phase 1	45.78	9.91	14.88	11.53
	phase 2	-	13.31	9.13	19.81
15 x 15 cells	phase 1	128.39	68.61	33.08	28.47
	phase 2	-	55.8	44.28	45.13
17 x 17 cells	phase 1	172.46	70.9	44.66	31.25
	phase 2	-	86.32	67.61	70.75
20 x 20 cells	phase 1	85.57	101.27	39.48	50.75
	phase 2	-	119.39	33.69	60.39
25 x 25 cells	phase 1	339.37	115.44	130.37	77.33
	phase 2	-	85.43	90.87	53.0

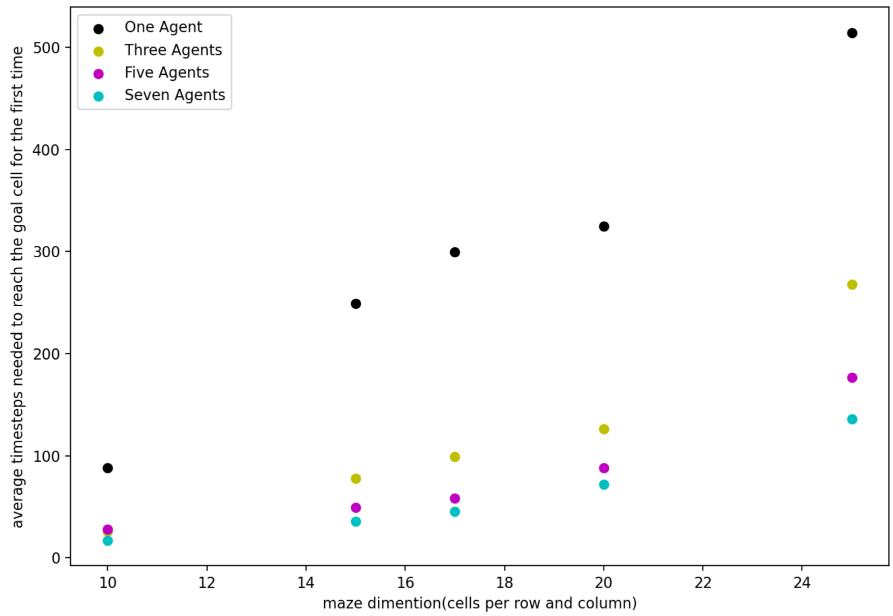


Figure 6.3: Results of the simulation for Phase One

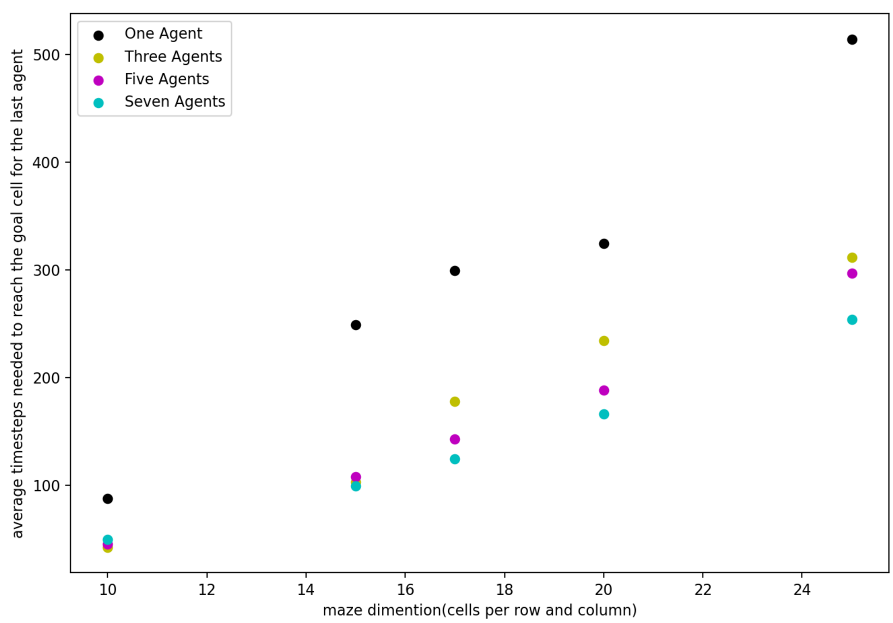


Figure 6.4: Results of the simulation for Phase Two

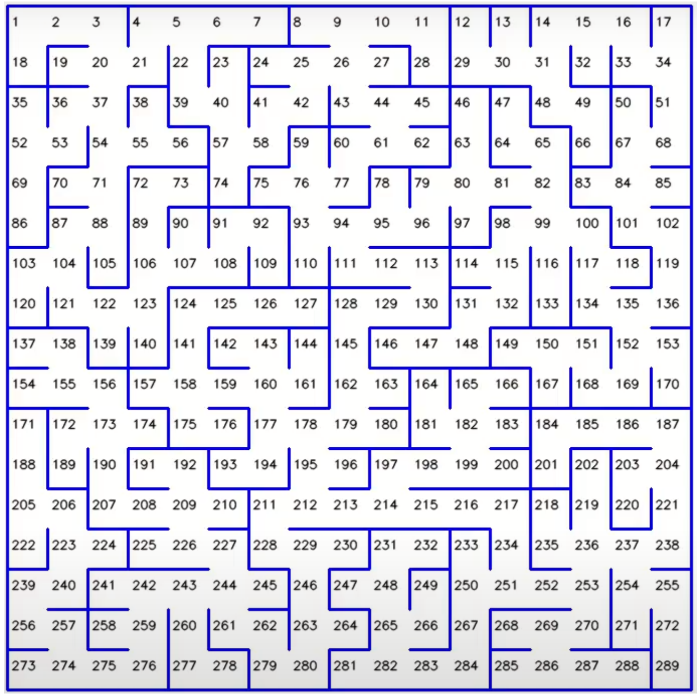


Figure 6.5: A random maze developed for the purpose of running a simulation. The *goal cell* was randomly assigned to the sixth row and sixth column.

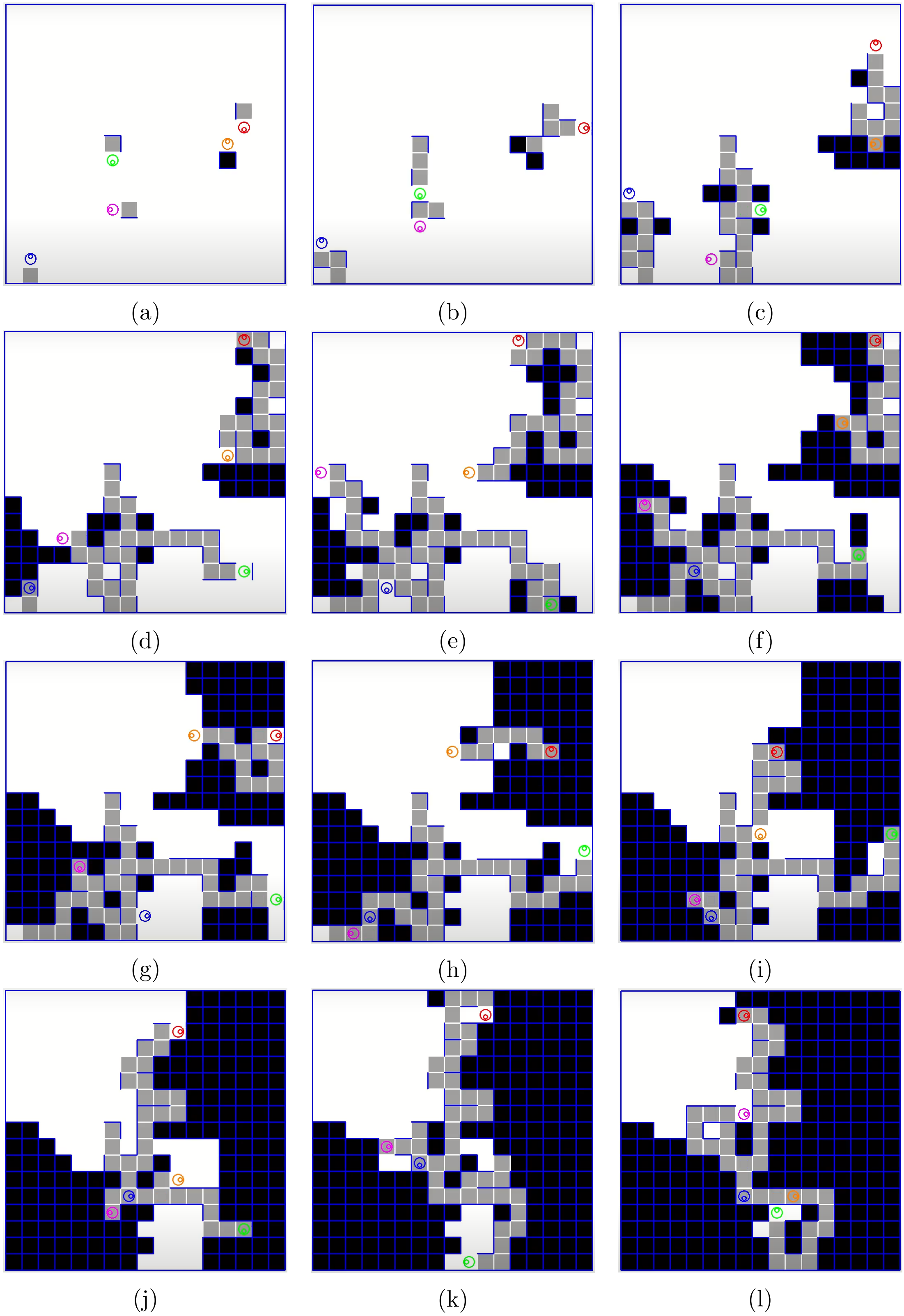


Figure 6.6: The timeline of solving the maze in Figure 6.6 for five agents from the starting point to reaching the *goal cell* by all agents.(Part A)

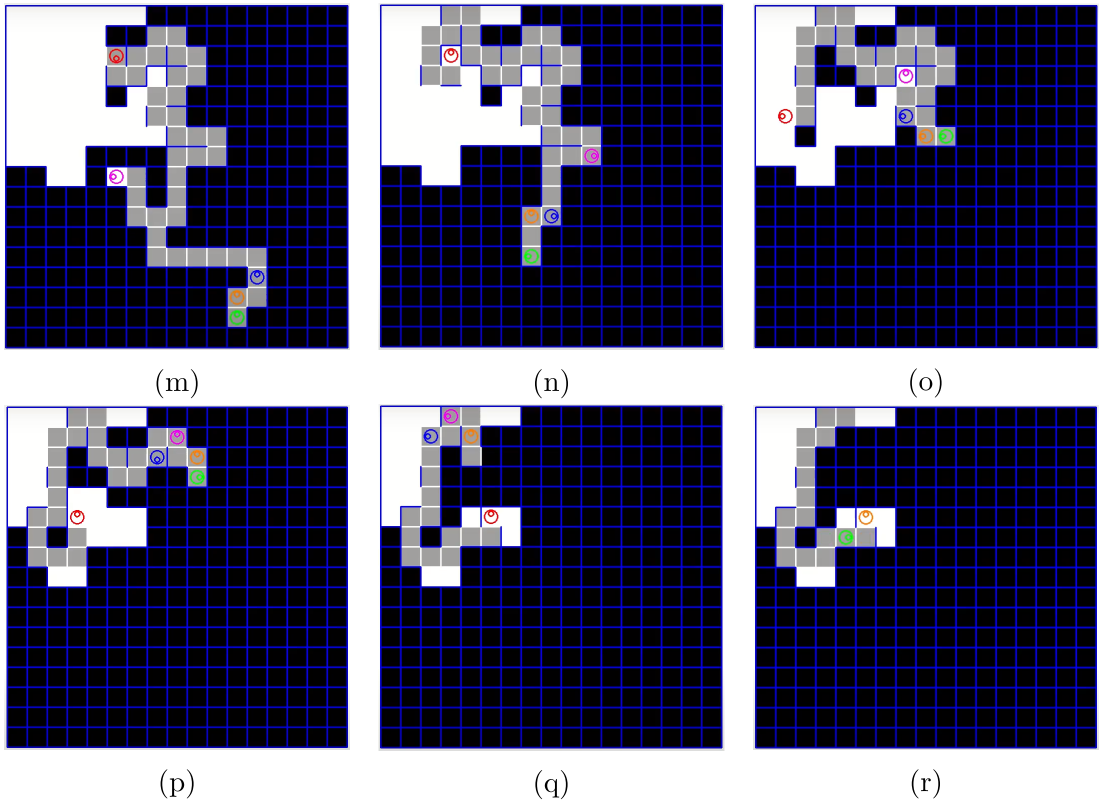


Figure 6.6: The timeline of solving the maze in Figure 6.6 for five agents from the starting point to reaching the *goal cell* by all agents.(Part B)

6.3 Physical test

To experimentally validate the result our algorithm was implemented using a maze-like field of $\{10 \times 10\}$ cells, and a group of three mobile robots working cooperatively. A floor mounted camera was used to monitor the location of each agent (Figure 6.7). We now describe each system component in more detail.

Figure 6.7: Our physical test formation

6.3.1 Agent

The mobile robots used as agents are GCtronic E-puck2 (Figure 6.8), a $70mm$ diameter and $45mm$ height robot with an $1800mAh$ rechargeable battery with a highest speed of $15.4cm/s$. Each E-puck2 robot contains the following onboard chips:

1. Main microcontroller, which handles sensors and actuators.

2. Programmer, which allows the user to configure the robot via a USB hub

3. Radio module, which permits communication with the robot via BlueTooth or WiFi.

Each E-puck2 robot has eight proximity sensors and a ToF sensor distributed as shown in Figure 6.9 [4]. We use $Prox2$, $Prox5$ and the ToF to detect right side walls, left side walls and front walls respectively. In our case, we use WiFi to communicate with the robots. All robots connect to a single hotspot and then, via the main system, commands are being executed (Figure 6.10).

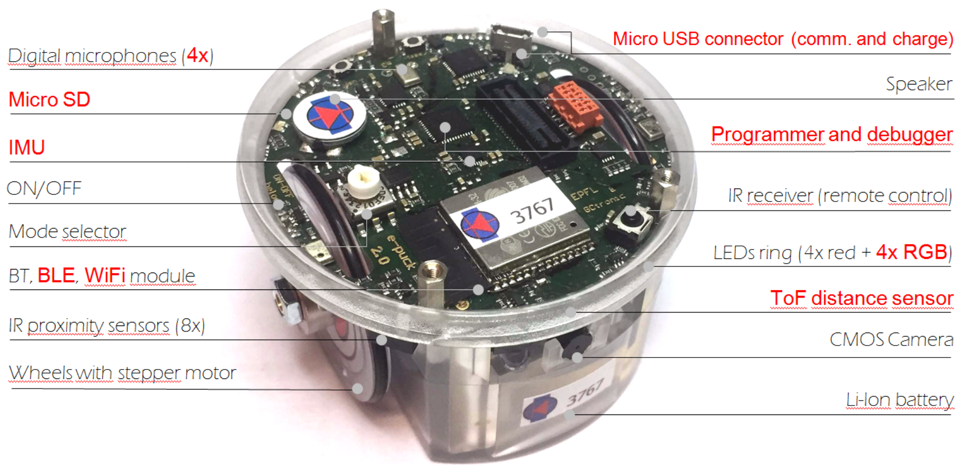


Figure 6.8: e-puck2 overview [4]

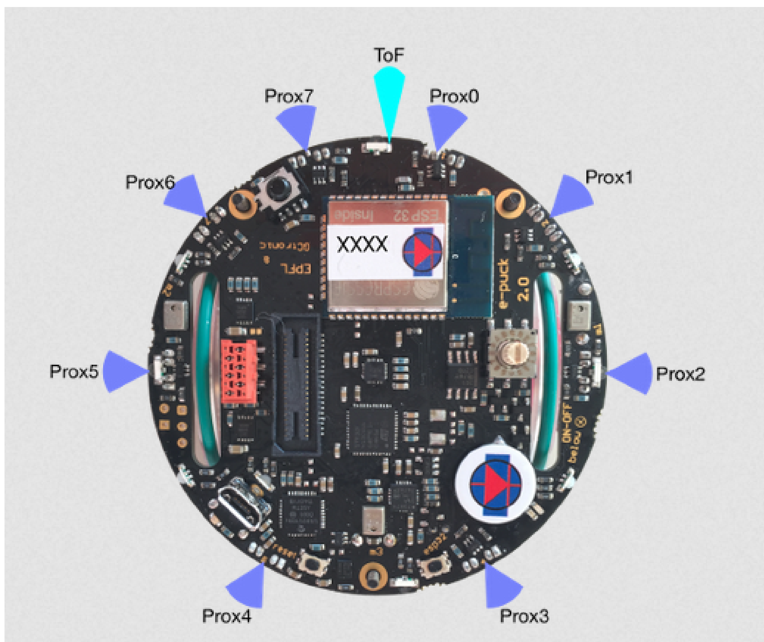


Figure 6.9: epuck2 proximity sensors [4]

Figure 6.10: Robots in the maze

6.3.2 Maze

We consider a maze-like wooden field with 120 cm width and 120 cm height. The dimension of each cell is $12cm \times 12cm$. The architecture consists of a one-way maze without cycles, *i.e.* the maze is a single tree, in the terminology of graph theory (Figure 6.11).

Figure 6.11: Maze used for the test

6.3.3 Camera

To avoid noisy odometry measurements calculated by each robot, we use an overhead camera to accurately localize robots in the maze. We use a StereoLabs ZED camera (Figure 6.12), which has dual 4MP lenses capable of 100FPS streaming and video recording. [55] The camera is mounted at a distance of 1.2m above the maze in order to cover the entire field.

Figure 6.12: ZED camera used for localization

To detect each robot in the field, we use specific markers to separate robots and other components, and use image processing to localize robots. We use ArUco markers [56] which are binary square fiducial markers used for camera pose estimation as shown in Figure 6.13. Distinct markers are attached on top of each robot to detect their position and orientation.

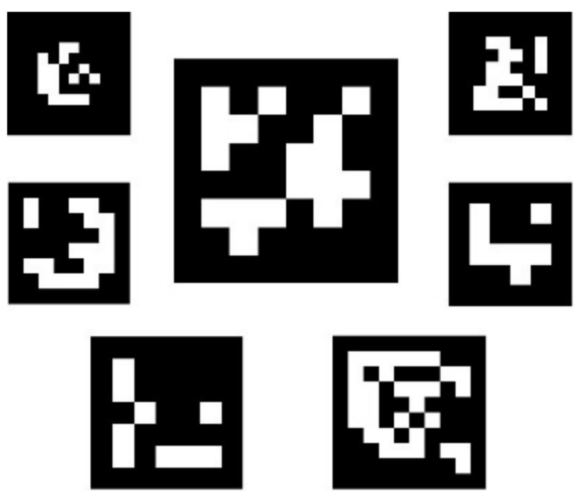


Figure 6.13: Examples for aruco markers used for localization[5]

6.3.4 Platform

ROS Kinetic crame (robot operating system) is used to facilitate communication between the robots and the camera, get sensor data and implement commands. ROS is a flexible framework that permits writing robot software while benefiting from a

collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behaviour across a wide variety of robotic platforms, [57]. Figures 6.14 and 6.15 show a general overview of the ROS environment and an abstract view of how sensor data and command data are connected.
Two main concepts exist in the ROS environment: ROS nodes and ROS topics.

- **Node**: A node is a program that interacts with other nodes using the ROS environment.

- **Topic**: Nodes may both publish messages to and subscribe to a topic.

- **Messages**: ROS data type, which are the presented data in topics.[57]

Each oval box in Figure 6.15 represents a Node, whereas each rectangular box represents a Topic in the ROS environment. The arrows linking the nodes and topics indicate a message that has been published. The agent's thread produces a node in the ROS environment corresponding to the E-puck robot that represents the agent. In Figure 6.15, the node *"/epuck2 robot 0"* is an example of a ROS node. Additionally, all rectangular boxes prefixed with *"proximity"* are the topics to which the robot's node is published.

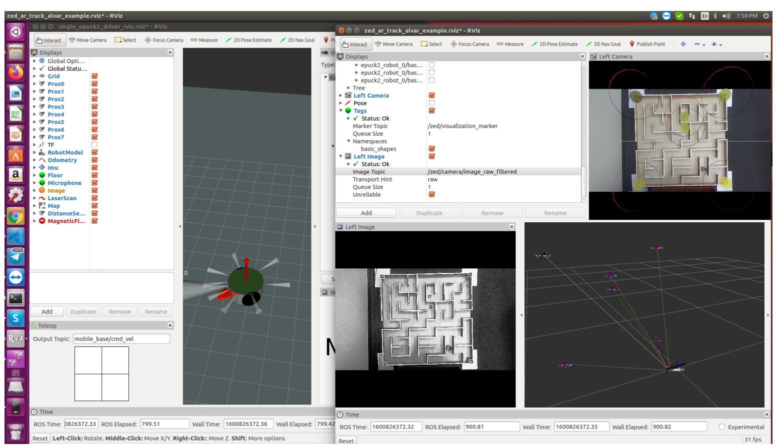


Figure 6.14: Overview of ROS Rviz environment

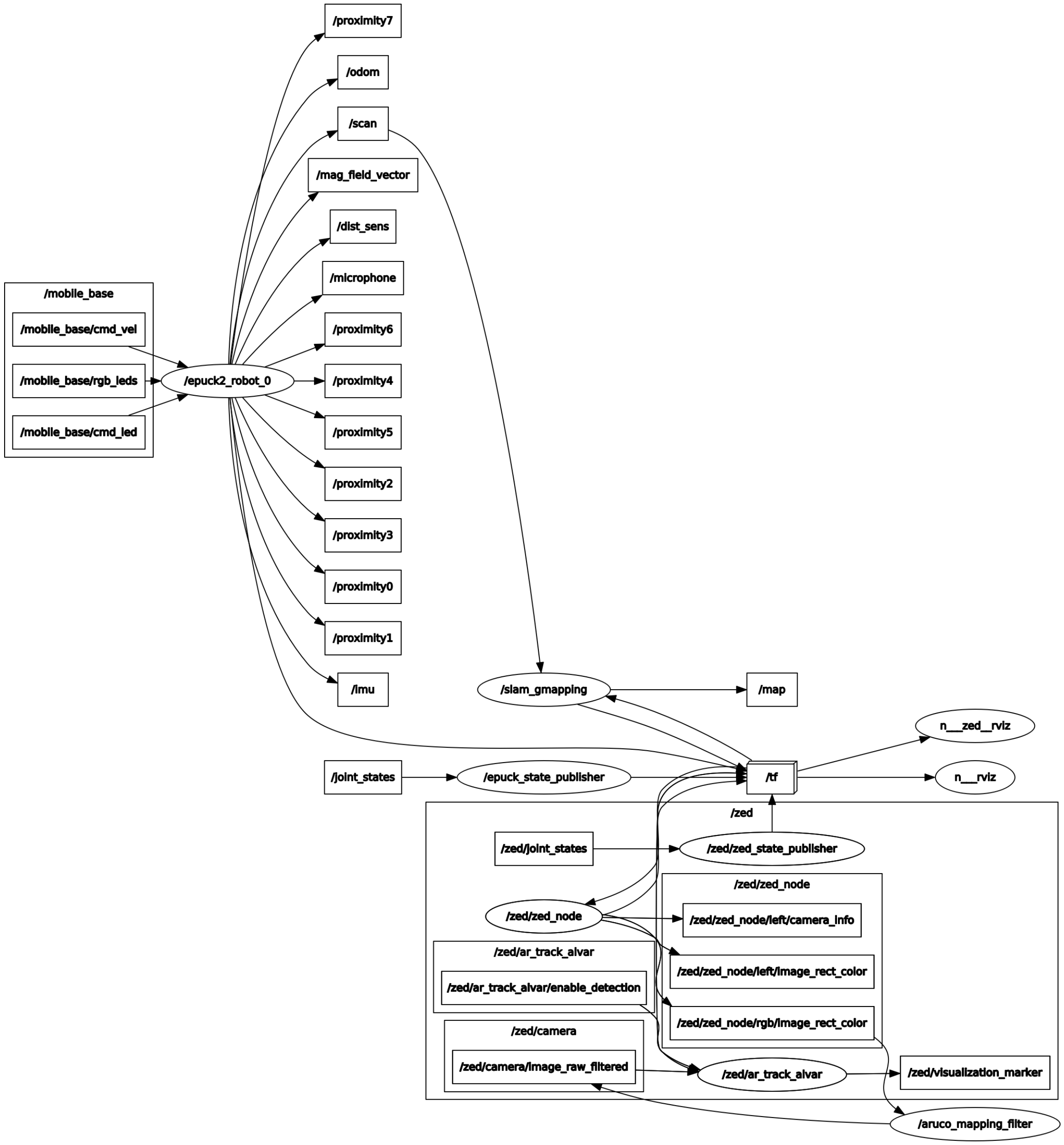
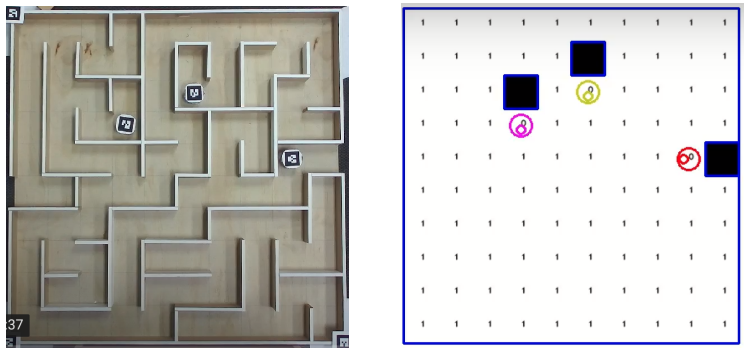


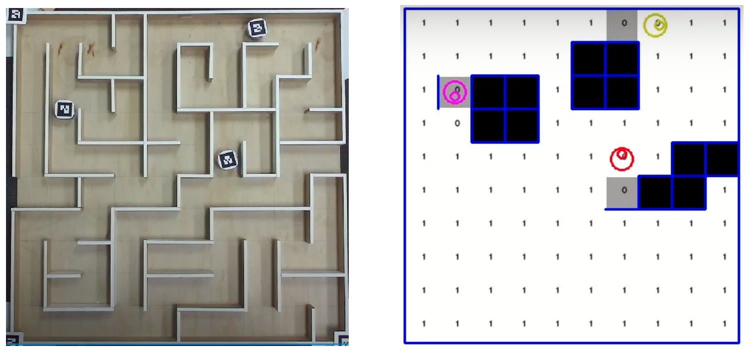
Figure 6.15: Overview of ROS graph of nodes

6.3.5 Results

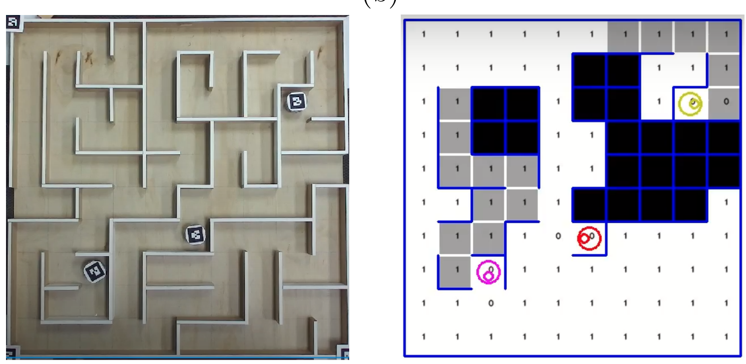
Previously, in Section 6.2 we show that for a 10×10 maze, using three agents is advisable. To verify the results in the experiment, we successfully tested our algorithm using a single robot and then using three robots working cooperatively. In the first case, it took 55 minutes for a single robot to complete the task of reaching the goal cell. Working cooperatively, it took 24 minutes for the three robots to reach the goal cell. It may raise confusion that it needs one-third time for three robots to explore the maze compared to one robot. This time margin is caused by the fact that robots may spend timesteps in their mutual subgraphs with other robots in phases one and two of the exploration. Figures 1 and 2 illustrate the timeline of the physical assessment for which we obtained the results. These figures represent images from the whole test and may help convey the nature of our algorithm's execution on a physical system. This physical test result allows us to generalize the simulation results for all sizes of mazes and a various number of robots in any given physical system. It also satisfies two main objectives: simultaneous movement of all robots and the ability of all robots to make decisions in real-time rather than sequentially. Figures 6.16 illustrate the test's timeline of events. It can provide us with a detailed report of what occurred throughout the test.



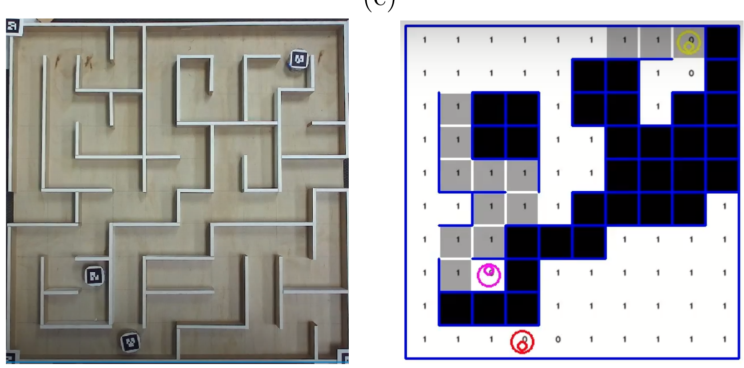
(a)



(b)

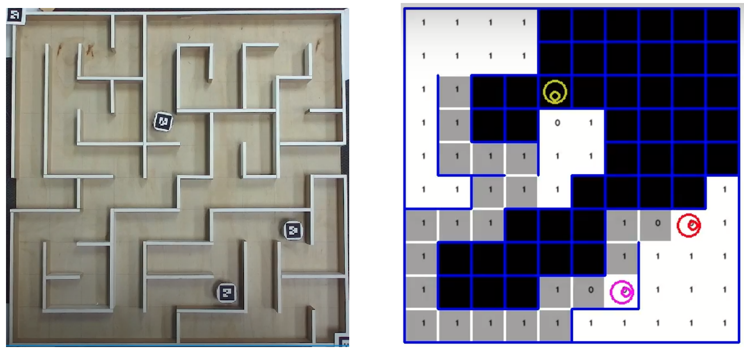


(c)

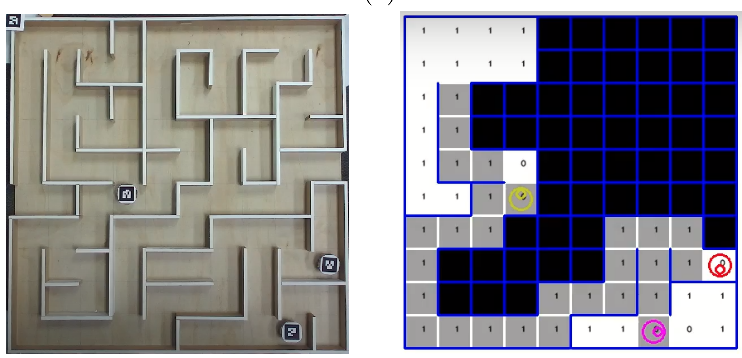


(d)

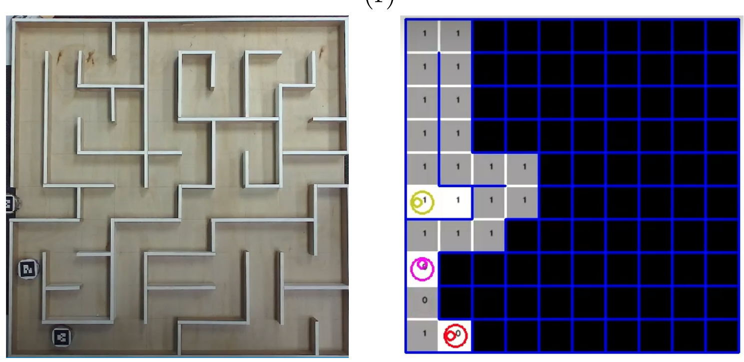
Figure 6.16: The timeline of solving the maze in Figure 6.6 for three agents from the starting point to reaching the destination by all agents.(Part A)



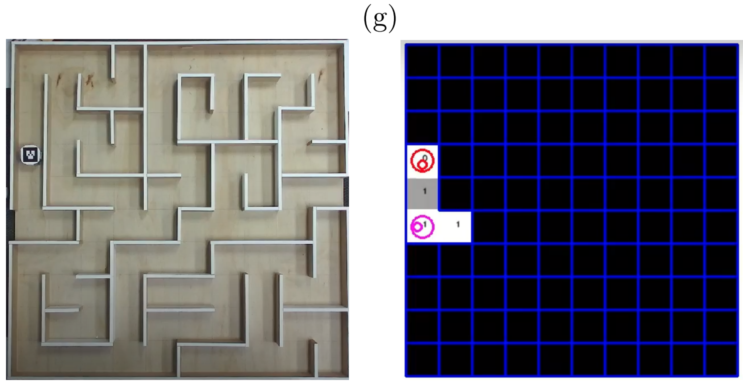
(e)



(f)



(g)



(h)

Figure 6.16: The timeline of solving the maze in Figure 6.6 for three agents from the starting point to reaching the destination by all agents.(Part B)

Chapter 7

Summary and Conclusions

This thesis considers the problem of cooperatively solving a maze by a group of multiple agents distributed randomly in an arbitrary rectangular maze without prior knowledge of the maze. We propose as a solution an algorithm for the agents, working cooperatively, that finds the hidden destination. We divide our algorithm into two main phases and presented rules for each one, where each agent is separately programmed to observe these rules. We tested our algorithm in a simulation using computer-generated square-shaped mazes with different sizes and a varying number of agents. The algorithm works well in the simulation, and it is effective in displaying the advantage of using multiple agents. We provide the results in terms of average required timesteps for both phases of the solution. Our solution mainly relies on working all agents cooperatively and in parallel, which distinguishes our algorithm from the algorithm of reference [35], in which much of this work was inspired, which discusses the solution in a way that agents move one at a time. We discusse the implementation of the algorithm by addressing the challenges we have, so the agents can follow the algorithm in such a way that all agents are allowed to move simultaneously. We introduce a trade-off between the number of agents, resources they take, and time to solve the maze for better decision-making inquiries. Finally, a physical system with a maze-like field and a floor-mounted camera were used to validate the simulation results. We successfully tested our algorithm using a single robot and then using three robots working cooperatively. This validation method demonstrates that our algorithm can be expanded to different maze sizes and varying numbers of robots as a physical system, hence validating our algorithm's real-world use by eliminating the discrete-time barriers associated with this challenge. Chapter 2 discusses the fundamental graph theory notations and algorithms that we employed throughout this thesis. These notions are essential in order to comprehend the solution given for the described problem. Chapter 3 describes the problem for which this thesis suggested a solution by using predefined ideas. Chapter 4 discusses the suggested solution, concentrating on the problem's theoretical aspects. The implementation part has been excluded since it requires a detailed structure to meet the requirements necessary for executing the solution. Chapter 5 attempts to address the implementation issues and the discussion around agent conflicts that our solution encounters. Chapter 6 validates and tests the theoretical solution and implementation structure outlined in Chapters 4 and 5. The results are presented and analyzed. Additionally, the physical system was described, which we utilize to conduct a feasibility study on the tested simulation.

7.1 Directions for Future Work

- For future studies, we may address the remaining issues of this topic by eliminating the "discrete field" barrier. To this end we may mesh and grid any surface to simulate a maze; the smaller the grid dimensions, the more accurate our maze model will be. This improvement can face some challenges that need to be solved. For instance, in the present problem definition, we considered the agent to be a "point" with no dimensions and to occupy an entire cell. In contrast, when confronted with the "discrete field" challenge, the dimension of the agent becomes an issue that must be addressed in order to find a suitable solution.

- While the algorithm used in phase two of our solution accomplishes the objective, it is unclear whether it is optimal. More research may be conducted by developing and executing phase two algorithms and further evaluating the results.

- This thesis discusses rectangular mazes and assigns Cartesian dimensions to each cell. One may do research on circular mazes using Polar dimensions to address each cell. Conducting research on other non-square-shaped mazes may also be challenging.

- The solution proposed in this thesis assumes that all agents are aware of the shared map's contents; agents may request access to the shared map's details as needed. One may propose solutions to challenges that arise as a result of constraints in the agents' communication protocol. Examples of the constraints that agents may encounter include the following:

- An agent may keep a subset of the map, or the subset of the map can be restricted to a certain distance from the agent.
- There is no central system that collects all of the agents' data from their explorations. The agents may communicate and exchange information only when they are next to each other in two neighbouring cells.
- Each agent's data transmissions and receptions are vulnerable to attack. It might be a denial of service or incorrect information. The agents' response to these circumstances may be examined.

Bibliography

[1] "kruskalwiki." https://en.wikipedia.org/wiki/Kruskal

[2] "Dijkstra wikipedia." url=https://en.wikipedia.org/wiki/Dijkstra(Date last accessed 4-Dec-2021).

[3] "A* algorithm in geekforgeeks website." url=https://www.geeksforgeeks.org/a-search-algorithm/, (Date last accessed 5-Dec-2021).

[4] "e-puck2." url=https://www.gctronic.com/doc/index.php/e-puck2, (Date last accessed 16-Feb-2021).

[5] "Detection of aruco markers." url=https://docs.opencv.org/master/d5/dae/tutorial_aruco_detection.html, (Date last accessed 18-Feb-2021).

[6] H. Kern, J. Saward, M. Schons, R. Ferré, A. Clay, S. Thomson, and K. Velder, *Through the Labyrinth: Designs and Meanings Over 5,000 Years*. Art and Design Series, Prestel, 2000.

[7] "The hedge maze at traquair house by sir gawain." url=https://www.flickr.com/photos/12173006@N08/2937635448, (Date last accessed 9-Dec-2021).

[8] "Maze-wiki." url=https://en.wikipedia.org/wiki/Maze, (Date last accessed 17-Nov-2021).

[9] "Longleat hedge maze, wiltshire, england, the longest hedge maze in the world.." url=https://www.atlasobscura.com/places/longleat-hedge-maze, (Date last accessed 9-Dec-2021).

[10] V. J. Lumelsky, "A comparative study on the path length performance of maze-searching and robot motion planning algorithms," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 1, pp. 57–66, 1991.

[11] P. J. Werbos and Xiaozhong Pang, "Generalized maze navigation: Srn critics solve what feedforward or hebbian nets cannot," in *1996 IEEE International Conference on Systems, Man and Cybernetics. Information Intelligence and Systems (Cat. No.96CH35929)*, vol. 3, pp. 1764–1769 vol.3, 1996.

[12] A. J. Bagnall and Z. V. Zatuchna, "On the classification of maze problems," *null*, 2005.

[13] A. M. J. Sadik, M. A. Dhali, H. M. A. B. Farid, T. U. Rashid, and A. Syeed, "A comprehensive and comparative study of maze-solving techniques by implementing graph theory," *null*, 2010.

[14] A. Felner, R. Stern, S. E. Shimony, E. Boyarski, M. Goldenberg, G. Sharon, N. Sturtevant, G. Wagner, and P. Surynek, "Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges," in *Tenth Annual Symposium on Combinatorial Search*, 2017.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd ed., 2009.

[16] S. Mishra and P. Bande, "Maze solving algorithms for micro mouse," *null*, 2008.

[17] M. Sharma, "Algorithms for micro-mouse," *null*, 2009.

[18] G. Law, "Quantitative comparison of flood fill and modified flood fill algorithms," *International Journal of Computer Theory and Engineering*, 2013.

[19] H. Dang, J. Song, and Q. Guo, "An efficient algorithm for robot maze-solving," *null*, 2010.

[20] J. Cai, X. Wan, M. Huo, and J. Wu, "An algorithm of micromouse maze solving," *null*, 2010.

[21] I. Sutherland, "A method for solving arbitrary-wall mazes by computer," *IEEE Transactions on Computers*, 1969.

[22] L. Wyard-Scott and Q.-H. Meng, "A potential maze solving algorithm for a micromouse robot," *null*, 1995.

[23] M. O. A. Aqel, A. Issa, M. Khdair, M. ElHabbash, M. AbuBaker, and M. Massoud, "Intelligent maze solving robot based on image processing and graph theory algorithms," *null*, 2017.

[24] R. Siegwart, I. Nourbakhsh, and D. Scaramuzza, "Introduction to autonomous mobile robots," *null*, 2004.

[25] M. Wooldridge, "An introduction to multiagent systems," *null*, 2002.

[26] J. Liu and J. Wu, *Multiagent robotic systems.* CRC press, 2018.

[27] B. Rahnama, M. C. Ozdemir, Y. Kiran, and A. Elçi, "Design and implementation of a novel weighted shortest path algorithm for maze solving robots," *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*, 2013.

[28] J. Ferber, J. Ferber, O. Gutknecht, O. Gutknecht, and F. Michel, "From agents to organizations: an organizational view of multi-agent systems," *AOSE*, 2003.

[29] P. Surynek, "A novel approach to path planning for multiple robots in bi-connected graphs," in *2009 IEEE International Conference on Robotics and Automation*, pp. 3613–3619, 2009.

[30] N. Sariff and N. Buniyamin, "An overview of autonomous mobile robot path planning algorithms," in *2006 4th Student Conference on Research and Development*, pp. 183–188, 2006.

[31] D. K. Grady, K. E. Bekris, and L. E. Kavraki, *Asynchronous Distributed Motion Planning with Safety Guarantees under Second-Order Dynamics*, pp. 53–70. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[32] I. Arel, C. Liu, T. Urbanik, and A. G. Kohls, "Reinforcement learning-based multi-agent system for network traffic signal control," *IET Intelligent Transport Systems*, vol. 4, no. 2, pp. 128–135, 2010.

[33] J. T. SCHWARTZ and M. SHARIR, "Chapter 8 - algorithmic motion planning in robotics," in *Algorithms and Complexity* (J. VAN LEEUWEN, ed.), Handbook of Theoretical Computer Science, pp. 391–430, Amsterdam: Elsevier, 1990.

[34] A. Elçi and B. Rahnama, "Human-robot interactive communication using semantic web tech. in design and implementation of collaboratively working robots," *null*, 2007.

[35] E. H. Kivelevitch and K. Cohen, "Multi-agent maze exploration," *Journal of Aerospace Computing, Information, and Communication*, vol. 7, no. 12, pp. 391–405, 2010.

[36] A. Elçi, B. Rahnama, and S. Kamran, "Defining a strategy to select either of closed/open world assumptions on semantic robots," in *2008 32nd Annual IEEE International Computer Software and Applications Conference*, pp. 417–423, 2008.

[37] D. B. West, *Introduction to Graph Theory.* Prentice Hall, 2 ed., September 2000.

[38] "graph prelims." url=https://www.baeldung.com/cs/graph-theory-intro, (Date last accessed 30-Nov-2021).

[39] "graph-theory-basics." url=https://www.geeksforgeeks.org/mathematics-graph-theory-basics-set-1/, (Date last accessed 1-Dec-2021).

[40] N. Robertson, P. Seymour, N. Foundation, and U. Research, *Graph Structure Theory: Proceedings of the AMS-IMS-SIAM Joint Summer Research Conference on Graph Minors, Held June 22 to July 5, 1991, with Support from the National Science Foundation and the Office of Naval Research.* Contemporary mathematics - American Mathematical Society, American Mathematical Society, 1993.

[41] E. Williamson, *Lists, Decisions and Graphs.* S. Gill Williamson.

[42] "graph-theory-basics." url=https://www.geeksforgeeks.org/mathematics-graph-theory-basics-set-1/, (Date last accessed 18-Dec-2021).

[43] F. Harary, *Graph Theory (on Demand Printing Of 02787).* CRC Press, 2018.

[44] "depth first search."
url=https://www.tutorialspoint.com/
data_structures_algorithms/depth_first_traversal.htm, (Date last accessed 2-Dec-2021).

[45] "breadth first search."
url=https://www.tutorialspoint.com/
data_structures_algorithms/breadth_first_traversal.htm, (Date last accessed 2-Dec-2021).

[46] R. C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, 1957.

[47] J. E. Hopcroft and J. D. Ullman, "Set merging algorithms," *SIAM Journal on Computing*, vol. 2, no. 4, pp. 294–303, 1973.

[48] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *null*, 1956.

[49] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, 1959.

[50] W. Zeng and R. L. Church, "Finding shortest paths on real road networks: the case for A*," Apr. 2009.

[51] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[52] "A* algorithm in wikipedia." https://en.wikipedia.org/wiki/A*_search_algorithm, (Date last accessed 5-Dec-2021).

[53] S. Jose and A. Antony, "Mobile robot remote path planning and motion control in a maze environment," in *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, pp. 207–209, 2016.

[54] R. Stern, N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, E. Boyarski, and R. Barták, "Multi-agent pathfinding: Definitions, variants, and benchmarks," *CoRR*, vol. abs/1906.08291, 2019.

[55] "Zed camera." url=https://www.stereolabs.com/zed/.

[56] S. Garrido-Jurado, R. Muñoz-Salinas, F. Madrid-Cuevas, and M. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition*, vol. 47, no. 6, pp. 2280–2292, 2014.

[57] "robot operating system." url=https://www.ros.org/about-ros/, (Date last accessed 18-Feb-2021).

Appendix A

Simulation Codes

Listing A.1: Main code for simulation and extracting data

```
from Classes.maze import Maze
from Classes.cam import Detector
from Classes.robot import Robot
import math
import random
import time
import threading
import numpy as np
from cv_bridge import CvBridge
import cv2 as cv
import pickle

def build_global_maze(dim):
    global_maze = Maze(dim, dim)
    global_maze.build_maze()
    File = open("mmm.pickle", 'wb')
    File.truncate()
    pickle.dump(global_maze, File)
    File.close()
    return global_maze

def build_robots(k, dim):
    maze = Maze(dim, dim)
    maze.build()
    goal_cell = maze.generate_random_cell()
    robots = []
    for i in range(k):
        robots.append(Robot(10 + i, colors[i], maze, goal_cell))
    maze.add_robots(robots)
    return robots, maze

def import_global_maze():
    File = open("mmm.pickle", "rb")
    global_maze = pickle.load(File)
    File.close()
    return global_maze
```

```python

def build_cam():
    robot_cam = Detector(maze, "robots", lock, robots)
    global_cam = Detector(global_maze, "global", lock)
    for i in range(k):
        robots[i].add_robot_cam(robot_cam)
    return robot_cam, global_cam

def main(global_maze):
    t = []
    for i in range(k):
        t.append(threading.Thread(target = robots[i].maze_exploration, args = (maze, global_maze, robot_cam, lock)))
    for i in range(len(t)):
        t[i].start()
    for i in range(len(t)):
        t[i].join()

    for i in range(k):
        robots[i].add_global_maze(global_maze)
        robots[i].goal_cell = maze.return_cell((10, 1))
        t.append(threading.Thread(target = robots[i].follow_path, args = (paths[i], maze, lock)))
    for i in range(len(t)):
        t[i].start()
    for i in range(len(t)):
        t[i].join()

if __name__ == '__main__':
    infile = open("dict", "rb")
    dic = pickle.load(infile)
    infile.close()
    for dim in [10, 15, 17, 20, 25]:
        for k in [1, 3, 5, 7]:
            global_maze = build_global_maze(dim)
            robots, maze = build_robots(k, dim)
            lock = threading.Lock()
            robot_cam, global_cam = build_cam()
            main(global_maze)
            max_overall = 0
            max_phase_one = 0
            for robot in robots:
                max_phase_one = max(robot.phase_one_counter, max_phase_one)
                max_overall = max(robot.overall_counter, max_overall)
            dic[k][dim].append((max_phase_one, max_overall))
            outfile = open("dict", "wb")
            pickle.dump(dic, outfile)
            outfile.close()
```

Listing A.2: Robot object code

```
import random
import time
from datetime import datetime
from collections import defaultdict
import math
import sys
from dijkstar import Graph, find_path

def euc_dist(cell1, cell2):
    return math.sqrt((cell1.Row - cell2.Row)**2 + (cell1.Col - cell2.Col)**2)
class Robot:
    def __init__(self, color, rgb, main_maze, goal_cell, initial_pos = None):
        self.Xpos = 0
        self.Ypos = 0
        self.RowInd = 0 #
        self.ColInd = 0 #
        self.Front = 1
        self.Left = 1
        self.Right = 1
        self.Back = 1
        self.direction = 0 # 0 left # 1 up # 2 right # 3 down
        self.visited = defaultdict(int)
        self.color = color
        self.rgb = rgb #
        self.goal_cell = goal_cell
        self.initializePos(main_maze, initial_pos)
        self.WL = defaultdict(float)
        self.key = "sum_of_distance"
        self.destination = None
        self.next_Cell = None
        self.changed = ""
        self.phase_one_counter = 0
        self.overall_counter = 0

    def CurrentCell(self, maze): # done
        return maze.cell[self.RowInd][self.ColInd]

    def LeftCell(self, maze): #done
        if self.direction == 0: #left
            return maze.down_cell(self.CurrentCell(maze))
        if self.direction == 1: #up
            return maze.left_cell(self.CurrentCell(maze))
        if self.direction == 2: #right
            return maze.up_cell(self.CurrentCell(maze))
        if self.direction == 3: #down
            return maze.right_cell(self.CurrentCell(maze))

    def RightCell(self, maze): # done
        if self.direction == 0: #left
            return maze.up_cell(self.CurrentCell(maze))
```

```
        if self.direction == 1: #up
            return maze.right_cell(self.CurrentCell(maze))
        if self.direction == 2: #right
            return maze.down_cell(self.CurrentCell(maze))
        if self.direction == 3: #down
            return maze.left_cell(self.CurrentCell(maze))

    def BackCell(self, maze): # done
        if self.direction == 0: #left
            return maze.right_cell(self.CurrentCell(maze))
        if self.direction == 1: #up
            return maze.down_cell(self.CurrentCell(maze))
        if self.direction == 2: #right
            return maze.left_cell(self.CurrentCell(maze))
        if self.direction == 3: #down
            return maze.up_cell(self.CurrentCell(maze))

    def FrontCell(self, maze): # done
        if self.direction == 0: #left
            return maze.left_cell(self.CurrentCell(maze))
        if self.direction == 1: #up
            return maze.up_cell(self.CurrentCell(maze))
        if self.direction == 2: #right
            return maze.right_cell(self.CurrentCell(maze))
        if self.direction == 3: #down
            return maze.down_cell(self.CurrentCell(maze))

    def FrontWall(self, maze): # done
        if self.direction == 0: # left
            return self.FrontCell(maze).rightWall
        if self.direction == 1: # up
            return self.FrontCell(maze).downWall
        if self.direction == 2: # right
            return self.CurrentCell(maze).rightWall
        if self.direction == 3: # down
            return self.CurrentCell(maze).downWall

    def BackWall(self, maze): # done
        if self.direction == 0: # left
            return self.CurrentCell(maze).rightWall
        if self.direction == 1: # up
            return self.CurrentCell(maze).downWall
        if self.direction == 2: # right
            return self.BackCell(maze).rightWall
        if self.direction == 3: # down
            return self.BackCell(maze).downWall

    def LeftWall(self, maze): # done
        if self.direction == 0: # left
            return self.CurrentCell(maze).downWall
        if self.direction == 1: # up
            return self.LeftCell(maze).rightWall
        if self.direction == 2: # right
            return self.LeftCell(maze).downWall
```

```
        if self.direction == 3: # down
            return self.CurrentCell(maze).rightWall

    def RightWall(self, maze): # done
        if self.direction == 0: # left
            return self.RightCell(maze).downWall
        if self.direction == 1: # up
            return self.CurrentCell(maze).rightWall
        if self.direction == 2: # right
            return self.CurrentCell(maze).downWall
        if self.direction == 3: # down
            return self.RightCell(maze).rightWall

    def TurnLeft(self): # done
        self.direction = (self.direction - 1) % 4

    def TurnRight(self): # done
        self.direction = (self.direction + 1) % 4

    def ForwardOneCell(self, maze, speed = 2): # done
        front_cell = self.FrontCell(maze)
        self.change_pos_to(front_cell)

    def change_pos_to(self, cell): # done
        self.RowInd = cell.Row
        self.ColInd = cell.Col
        self.Xpos = cell.xpos
        self.Ypos = cell.ypos

    def initializePos(self, maze, initial_pos): # done
        random.seed(datetime.now())
        if initial_pos is None:
            first_cell = maze.generate_random_cell()
            while first_cell == self.goal_cell:
                first_cell = maze.generate_random_cell()
        else: first_cell = initial_pos
        if first_cell.color != 0:
            first_cell = maze.generate_random_cell()
        self.change_pos_to(first_cell)
        self.direction = random.randint(0, 3)
        self.CurrentCell(maze).set_color(self.color)
        self.CurrentCell(maze).set_OC_flag(True)
        self.add_to_visited_cell(self.CurrentCell(maze))

    def add_to_visited_cell(self, next_Cell): # done
        self.visited[next_Cell.id] += 1

    def explore(self, maze, global_maze, robot_cam, lock): # done
        self.AssignWall(maze, global_maze, robot_cam)
        next_Cell = self.Choose_direction(maze)
        self.Move(maze, next_Cell)

    def Move(self, maze, next_Cell): # done
        self.overall_counter += 1
```

```
        if next_Cell == self.CurrentCell(maze):
            return
        prev_Cell = self.CurrentCell(maze)
        next_Cell.OC_flag = True
        if next_Cell == self.LeftCell(maze):
            self.TurnLeft()
        if next_Cell == self.RightCell(maze):
            self.TurnRight()
        if next_Cell == self.FrontCell(maze):
            pass
        if next_Cell == self.BackCell(maze):
            self.TurnLeft()
            self.TurnLeft()
        self.color_assign(maze, next_Cell)
        self.ForwardOneCell(maze)
        prev_Cell.OC_flag = False
        self.add_to_visited_cell(next_Cell)

    def color_assign(self, maze, next_Cell): # done
        if next_Cell.get_color() == 0:
            if self.direction_sum(maze) == 1:
                self.dead_end(maze)
            else:
                self.CurrentCell(maze).set_color(1)
                self.robot_cam.set_color(self.CurrentCell(maze), 1)
        elif next_Cell.get_color() == 1:
            if self.direction_sum(maze) == 1:
                self.dead_end(maze)
            else:
                self.CurrentCell(maze).set_color(1)
                self.robot_cam.set_color(self.CurrentCell(maze), 1)
        next_Cell.set_color(self.color)
        self.robot_cam.set_color(next_Cell, self.color, self)

    def dead_end(self, maze): # done
        self.CurrentCell(maze).set_color(2)
        self.robot_cam.set_color(self.CurrentCell(maze), 2)
        maze.add_obstacle(self.CurrentCell(maze), maze.up_cell(self.CurrentCell(maze)), self.robot_cam)
        maze.add_obstacle(self.CurrentCell(maze), maze.left_cell(self.CurrentCell(maze)), self.robot_cam)
        maze.add_obstacle(self.CurrentCell(maze), maze.right_cell(self.CurrentCell(maze)), self.robot_cam)
        maze.add_obstacle(self.CurrentCell(maze), maze.down_cell(self.CurrentCell(maze)), self.robot_cam)

    def direction_sum(self, maze): # done
        return self.FrontWall(maze) + self.BackWall(maze) + self.LeftWall(maze) + self.RightWall(maze)

    def AssignWall(self, maze, global_maze, robot_cam): # done
        if self.LeftWall(global_maze) == 0 and self.LeftWall(maze) == 1:
            maze.add_obstacle(self.CurrentCell(maze), self.LeftCell(maze), robot_cam)
```

```
        elif self.LeftWall(global_maze) == 1:
            maze.add_to_graph(self.CurrentCell(maze), self.LeftCell(maze))
        if self.FrontWall(global_maze) == 0 and self.FrontWall(maze) == 1:
            maze.add_obstacle(self.CurrentCell(maze), self.FrontCell(maze), robot_cam)
        elif self.FrontWall(global_maze) == 1:
            maze.add_to_graph(self.CurrentCell(maze), self.FrontCell(maze))
        if self.RightWall(global_maze) == 0 and self.RightWall(maze) == 1:
            maze.add_obstacle(self.CurrentCell(maze), self.RightCell(maze), robot_cam)
        elif self.RightWall(global_maze) == 1:
            maze.add_to_graph(self.CurrentCell(maze), self.RightCell(maze))
        if self.BackWall(global_maze) == 0 and self.BackWall(maze) == 1:
            maze.add_obstacle(self.CurrentCell(maze), self.BackCell(maze), robot_cam)
        elif self.BackWall(global_maze) == 1:
            maze.add_to_graph(self.CurrentCell(maze), self.BackCell(maze))
        robot_cam.present()

    def Cell_categorize(self, maze, next_Cell = None): # done
        white_cells = []
        grey_cells = []
        if self.LeftCell(maze) and self.LeftWall(maze) == 1 and self.LeftCell(maze).get_OC_flag() == False:
            if self.LeftCell(maze).get_color() == 0:
                white_cells.append(self.LeftCell(maze))
            if self.LeftCell(maze).get_color() == 1:
                grey_cells.append(self.LeftCell(maze))
        if self.FrontCell(maze) and self.FrontWall(maze) == 1 and self.FrontCell(maze).get_OC_flag() == False:
            if self.FrontCell(maze).get_color() == 0:
                white_cells.append(self.FrontCell(maze))
            if self.FrontCell(maze).get_color() == 1:
                grey_cells.append(self.FrontCell(maze))
        if self.RightCell(maze) and self.RightWall(maze) == 1 and self.RightCell(maze).get_OC_flag() == False:
            if self.RightCell(maze).get_color() == 0:
                white_cells.append(self.RightCell(maze))
            if self.RightCell(maze).get_color() == 1:
                grey_cells.append(self.RightCell(maze))
        if self.BackCell(maze) and self.BackWall(maze) == 1 and self.BackCell(maze).get_OC_flag() == False:
            if self.BackCell(maze).get_color() == 0:
                white_cells.append(self.BackCell(maze))
            if self.BackCell(maze).get_color() == 1:
                grey_cells.append(self.BackCell(maze))

        for cell in white_cells:
```

```
            if self.goal_cell.color == 0:
                self.WL[cell] = float('inf')
            else:
                self.WL[cell] = self.heuristic(cell, maze, key=self.key)
        print(self.color, "wl", [cell.id for cell in self.WL.keys()], self.CurrentCell(maze).id, self.changed)
        return white_cells, grey_cells

    def heuristic(self, cell, maze, key):
        if key == "sum_of_distance":
            return euc_dist(self.goal_cell, cell) + euc_dist(cell, self.CurrentCell(maze))

        if key == "closest_white_to_robot":
            return euc_dist(cell, self.CurrentCell(maze))
        if key == "closest_white_to_goal":
            return euc_dist(self.goal_cell, cell)

    def Choose_direction(self, maze):
        white_cells, grey_cells = self.Cell_categorize(maze)
        if len(white_cells):
            next_Cell = random.choice(white_cells)
            del self.WL[next_Cell]
            print(self.color, "wl", [cell.id for cell in self.WL.keys()], self.CurrentCell(maze).id)
            return next_Cell
        if len(grey_cells) == 0:
            return self.CurrentCell(maze)
        new_grey_cells = self.sort_cell_list(grey_cells)
        next_Cell = new_grey_cells[0]
        return next_Cell

    def sort_cell_list(self, cell_list):
        n = len(cell_list)
        for i in range(n):
            for j in range(n - i - 1):
                if self.visited[cell_list[j].id] > self.visited[cell_list[j + 1].id]:
                    cell_list[j], cell_list[j + 1] = cell_list[j + 1], cell_list[j]
        new_list = [cell_list[0]]
        val = self.visited[cell_list[0]]
        for i in range(1, n):
            if self.visited[cell_list[i]] == val:
                new_list.append(cell_list[i])
        return new_list

    def maze_exploration(self, maze, global_maze, robot_cam, lock):
        print("start")
        lock.acquire()
        self.global_maze = global_maze
        lock.release()
        while self.goal_cell.color == 0:
            lock.acquire()
```

```
            start = time.time()
            self.explore(maze, global_maze, robot_cam, lock)
            self.phase_one_counter += 1
            lock.release()
            while time.time() - start < .25:
                pass

        lock.acquire()
        self.AssignWall(maze, global_maze, robot_cam)
        self.Cell_categorize(maze)
        for cell in self.WL:
            self.WL[cell] = self.heuristic(cell, maze, key=self.key)
        path = find_path(maze.graph, self.CurrentCell(maze), self.goal_cell)
        lock.release()
        while not path:
            lock.acquire()
            print(self.color, "no path", [cell.id for cell in self.WL.keys()], self.CurrentCell(maze).id)
            if not len(self.WL):
                print(self.color, "empty white", self.CurrentCell(maze).id)
                self.explore(maze, global_maze, robot_cam, lock)
                path = find_path(maze.graph, self.CurrentCell(maze), self.goal_cell)
                lock.release()
                time.sleep(.01)
                continue
            closest_white = min(self.WL, key=self.WL.get)
            if closest_white.color != 0:
                del self.WL[closest_white]
                lock.release()
                continue
            path = find_path(maze.graph, self.CurrentCell(maze), closest_white)
            lock.release()
            self.follow_path(path[0], maze, lock)
            lock.acquire()
            self.AssignWall(maze, global_maze, robot_cam)
            self.Cell_categorize(maze)
            #assign_wall
            path = find_path(maze.graph, self.CurrentCell(maze), self.goal_cell)
            lock.release()
        self.follow_path(path[0], maze, lock)
        self.CurrentCell(maze).color = 1
        self.CurrentCell(maze).OC_flag = False

    def follow_path(self, path, maze, lock):
        lock.acquire()
        self.destination = path[-1]
        print(self.color, "heading to", self.destination.id)
        lock.release()
        for i in range(1, len(path)):
```

```
lock.acquire()
start = time.time()
if self.destination.color == 3:
    print(self.color, "destination explored before, break!")
    lock.release()
    break
self.next_Cell = path[i]
self.AssignWall(maze, self.global_maze, self.robot_cam)
self.Cell_categorize(maze)
lock.release()
counter = 0
while self.next_Cell.OC_flag == True:
    time.sleep(.1)
    lock.acquire()
    if counter > 3:
        front_robot = maze.robot_ref(self.next_Cell)
        if front_robot: print(self.color, "front robot color is", front_robot.color)
        if front_robot.next_Cell == self.CurrentCell(maze):
            lock.release()
            maze.switch_destination(self, front_robot, lock)
            lock.acquire()
            print("destination switched", self.color, "heading to", self.destination.id)
            #bell()
    lock.release()
    lock.acquire()
    if not self.destination:
        self.explore(maze, self.global_maze, self.robot_cam)
        print(self.color, "empty dest, explore")
        lock.release()
        return

    if self.destination != path[-1]:
        print(self.color, "heading to new destination", self.destination.id)
        self.changed = "changed"
        #lock.acquire()
        path = find_path(maze.graph, self.CurrentCell(maze), self.destination)
        lock.release()
        self.follow_path(path[0], maze, lock)
        return
    print(self.color, "wait next cell is", self.next_Cell.id, "counter is", counter)
    lock.release()
    time.sleep(.1)
    counter += 1
lock.acquire()
self.Move(maze, self.next_Cell)
if self.CurrentCell(maze) in self.WL:
    del self.WL[self.CurrentCell(maze)]
    print(self.color, "wl", [cell.id for cell in self.WL.keys()], self.CurrentCell(maze).id)
```

```
            lock.release()
            while(time.time() - start < .25):
                pass
        lock.acquire()
        self.AssignWall(maze, self.global_maze, self.robot_cam)
        self.Cell_categorize(maze)
        print(self.color, "follow path done, current is ", self.
            CurrentCell(maze).id)
        self.destination = None
        lock.release()

    def extract_path(self, maze, cell_a, cell_b):
        path = find_path(maze.graph, cell_a, cell_b)
        return path[0]

    def add_robot_cam(self, robot_cam):
        self.robot_cam = robot_cam

    def add_global_maze(self, global_maze):
        self.global_maze = global_maze
```

Listing A.3: Maze object code

```
import random
from datetime import datetime
from collections import defaultdict
import collections
import marshal
import os
from copy import copy
from dijkstar import Graph, find_path
class Cell:
    def __init__(self, Row, Col, id):
        self.Row = Row
        self.Col = Col
        self.id = id
        self.xpos = 0
        self.ypos = 0
        self.downWall = 1 # 1 for way
        self.rightWall = 1 # 0 for wall
        self.color = 0
        self.par = self
        self.OC_flag = False
        self.graph = Graph()

    def set_color(self, color):
        self.color = color

    def get_color(self):
        return self.color

    def set_OC_flag(self, flag):
        self.OC_flag = flag

    def get_OC_flag(self):
        return self.OC_flag
class Maze:
    def __init__(self, Row, Col, StartRow = 0, StartCol = 0, FinishRow =
        6, FinishCol = 1):
        self.StartRow = StartRow
        self.StartCol = StartCol
        self.FinishRow = FinishRow
        self.FinishCol = FinishCol
        self.Row = Row + 1
        self.Col = Col + 1
        self.cell = [[Cell(row, col, (self.Col - 1) * (row - 1) + col)
            for col in range(self.Col)] for row in range(self.Row)]
        self.connected_components = (len(self.cell) - 1) * (len(self.
            cell[0]) - 1)
        self.graph = Graph()

    def add_robots(self, robots):
        self.robots = robots

    def get_id_from_cell(self, cell):
```

```
        return cell.id

    def get_cell_from_id(self, id):
        row = id // (self.Col - 1) + 1
        col = id - (self.Col - 1) * (row - 1) - col
        return self.cell[row][col]

    def up_cell(self, cell_u):
        row_u, col_u = cell_u.Row, cell_u.Col
        if row_u == 0: return None
        return self.cell[row_u - 1][col_u]

    def up_wall(self, cell_u):
        row_u, col_u = cell_u.Row, cell_u.Col
        if row_u == 0: return None
        return self.up_cell(cell_u).downWall

    def down_cell(self, cell_u):
        row_u, col_u = cell_u.Row, cell_u.Col
        if row_u == self.Row - 1: return None
        return self.cell[row_u + 1][col_u]

    def down_wall(self, cell_u):
        row_u, col_u = cell_u.Row, cell_u.Col
        if row_u == self.Row - 1: return None
        return cell_u.downWall

    def left_cell(self, cell_u):
        row_u, col_u = cell_u.Row, cell_u.Col
        if col_u == 0: return None
        return self.cell[row_u][col_u - 1]

    def left_wall(self, cell_u):
        row_u, col_u = cell_u.Row, cell_u.Col
        if col_u == 0: return None
        return self.left_cell(cell_u).rightWall

    def right_cell(self, cell_u):
        row_u, col_u = cell_u.Row, cell_u.Col
        if col_u == self.Col - 1: return None
        return self.cell[row_u][col_u + 1]

    def right_wall(self, cell_u):
        row_u, col_u = cell_u.Row, cell_u.Col
        if col_u == self.Col - 1: return None
        return cell_u.rightWall

    def set_obstacle(self, cell_u, cell_v, robot_cam, val):
        if not cell_u: return
        if not cell_v: return
        row_u, col_u = cell_u.Row, cell_u.Col
        row_v, col_v = cell_v.Row, cell_v.Col
        if row_u == row_v:
            if col_u < col_v:
```

```
                cell_u.rightWall = val
                if not val: robot_cam.add_right_wall(cell_u)
            else:
                cell_v.rightWall = val
                if not val: robot_cam.add_right_wall(cell_v)
        else:
            if row_u < row_v:
                cell_u.downWall = val
                if not val: robot_cam.add_down_wall(cell_u)
            else:
                cell_v.downWall = val
                if not val: robot_cam.add_down_wall(cell_v)

    def add_obstacle(self, cell_u, cell_v, robot_cam = None):
        self.set_obstacle(cell_u, cell_v, robot_cam, 0)

    def remove_obstacle(self, cell_u, cell_v, robot_cam = None):
        self.set_obstacle(cell_u, cell_v,robot_cam, 1)

    def add_to_graph(self, cell_u, cell_v):
        self.graph.add_edge(cell_u, cell_v, 1)
        self.graph.add_edge(cell_v, cell_u, 1)

    def build(self):
        Row = len(self.cell)
        Col = len(self.cell[0])
        self.cell[Row - 1][Col - 1].xpos = -2.25
        self.cell[Row - 1][Col - 1].ypos = -2.25
        self.cell[1][1].xpos = 2.25
        self.cell[1][1].ypos = 2.25
        self.cell[Row - 1][1].xpos = -2.25
        self.cell[Row - 1][1].ypos = 2.25
        self.cell[1][Col - 1].xpos = 2.25
        self.cell[1][Col - 1].ypos = 2.25
        for r in range(1, Row):
            for c in range(1, Col):
                self.cell[r][c].xpos = 2.25 - .5 * (r - 1)
                self.cell[r][c].ypos = 2.25 - .5 * (c - 1)
        for i in range(1, Row):
            self.cell[i][Col - 1].rightWall = 0
            self.cell[i][0].rightWall = 0
        for i in range(1, Col):
            self.cell[0][i].downWall = 0
            self.cell[Row - 1][i].downWall = 0

    def find(self, u):
        if u != u.par:
            u.par = self.find(u.par)
        return u.par

    def union(self, cell_u, cell_v):
        if not cell_u or cell_u.Row == 0 or cell_u.Col == 0 or cell_u.
            Row == self.Row or cell_u.Col == self.Col:
            return False
```

```
        if not cell_v or cell_v.Row == 0 or cell_v.Col == 0 or cell_v.Row == self.Row or cell_v.Col == self.Col:
            return False
        par_u = self.find(cell_u)
        par_v = self.find(cell_v)
        if par_u == par_v:
            return False
        par_u.par = par_v
        self.remove_obstacle(cell_u, cell_v) #
        self.connected_components -= 1
        return True

    def build_maze(self):
        self.build()
        self.make_all_obstacle()
        self.generate_random_obstacles()

    def make_all_obstacle(self):
        for i in range(1, self.Row):
            for j in range(1, self.Col):
                self.cell[i][j].rightWall = 0
                self.cell[i][j].downWall = 0

    def generate_random_cell(self):
        i = random.randint(1, self.Row - 1)
        j = random.randint(1, self.Col - 1)
        return self.cell[i][j]

    def return_cell(self, pos):
        return self.cell[pos[0]][pos[1]]

    def generate_random_obstacles(self):
        random.seed(datetime.now())
        while(self.connected_components > 1):
            cur_cell = self.generate_random_cell()
            lst = [self.up_cell(cur_cell), self.down_cell(cur_cell), self.left_cell(cur_cell), self.right_cell(cur_cell)]
            random_cell = random.choice(lst)
            if self.union(cur_cell, random_cell):
                pass

    def build_graph(self, global_maze):
        for i in range(1, self.Row):
            for j in range(1, self.Col):
                curr_cell = self.cell[i][j]
                global_cell = global_maze.cell[i][j]
                if self.right_cell(global_cell) and self.right_wall(global_cell):
                    self.add_to_graph(curr_cell, self.right_cell(curr_cell))
                if self.down_cell(global_cell) and self.down_wall(global_cell):
                    self.add_to_graph(curr_cell, self. down_cell(curr_cell))
```

```
    def get_robots_pos(self):
        return [self.robots[i].CurrentCell(self) for i in range(len(
            robots))]

    def robot_ref(self, cell):
        for robot in self.robots:
            if robot.CurrentCell(self) == cell:
                return robot
        return None

    def switch_destination(self, robot1, robot2, lock):
        print("here")
        lock.acquire()

        if robot1.color > robot2.color:
            lock.release()
            return
        robot1.destination, robot2.destination = robot2.destination,
            robot1.destination
        print("in switch")
        lock.release()
```

Listing A.4: Camera object code

```python
import cv2 as cv
import numpy as np
import time
class Detector:
    def __init__(self, maze, name, lock, robots = []):
        self.robots = robots
        self.maze = maze
        self.name = name
        self.robot_count = 0
        self.thickness = 2
        self.w = 40
        self.lock = lock
        for i in robots:
            if i:
                self.robot_count += 1
        self.initialize()

    def initialize(self):
        start = time.time()
        w = 40
        maze = self.maze
        Row = len(self.maze.cell)
        Col = len(self.maze.cell[0])
        self.img = np.zeros([(Row + 1) * w, (Col + 3) * w, 3], dtype = np.uint8)
        self.img.fill(255)
        for i in range(Row):
            for j in range(Col):
                if self.name == "global" and 0 < i < Row and 0 < j < Col:
                    font = cv.FONT_HERSHEY_SIMPLEX
                    xx = int(w * (j + .1))
                    yy = int(w * (i + .5))
                    cv.putText(self.img, str(maze.cell[i][j].id), (xx, yy), font, .4, (0, 0, 0), self.thickness // 10, cv.LINE_AA)

                if self.maze.cell[i][j].downWall == 0:
                    x1 = w * j
                    y1 = w * (i + 1)
                    x2 = w * (j + 1)
                    y2 = w * (i + 1)
                    cv.line(self.img, (x1, y1), (x2, y2), (255, 0, 0), self.thickness, lineType = 8)
                if self.maze.cell[i][j].rightWall == 0:
                    x1 = w * (j + 1)
                    y1 = w * i
                    x2 = w * (j + 1)
                    y2 = w * (i + 1)
                    cv.line(self.img, (x1, y1), (x2, y2), (255, 0, 0), self.thickness, lineType = 8)

```

```
        for robot in self.robots:
            self.set_color(robot.CurrentCell(self.maze), robot.color, robot)
        self.present()

    def present(self):
        if self.name == "global":
            cv.imshow(self.name, self.img)
            cv.moveWindow(self.name,1000,60)
            cv.waitKey(1)
        else:

            cv.moveWindow(self.name,100,60)
            cv.imshow(self.name, self.img)
            cv.waitKey(1)

    def renew_image(self, key, last_cell, next_cell):
        pass

    def add_right_wall(self, cell):
        i = cell.Row
        j = cell.Col
        w = self.w
        x1 = w * (j + 1)
        y1 = w * i
        x2 = w * (j + 1)
        y2 = w * (i + 1)
        cv.line(self.img, (x1, y1), (x2, y2), (255, 0, 0), self.thickness, lineType = 8)

    def add_left_wall(self, cell):
        i = cell.Row
        j = cell.Col
        w = self.w
        x1 = w * j
        y1 = w * i
        x2 = w * j
        y2 = w * (i + 1)
        cv.line(self.img, (x1, y1), (x2, y2), (255, 0, 0), self.thickness, lineType = 8)

    def add_up_wall(self, cell):
        i = cell.Row
        j = cell.Col
        w = self.w
        x1 = w * j
        y1 = w * (i + 1)
        x2 = w * (j + 1)
        y2 = w * (i + 1)
        cv.line(self.img, (x1, y1), (x2, y2), (255, 0, 0), self.thickness, lineType = 8)

    def add_down_wall(self, cell):
        i = cell.Row
```

```
        j = cell.Col
        w = self.w
        x1 = w * j
        y1 = w * (i + 1)
        x2 = w * (j + 1)
        y2 = w * (i + 1)
        cv.line(self.img, (x1, y1), (x2, y2), (255, 0, 0), self.thickness, lineType = 8)

    def set_color(self, cell, color, robot = None):
        i = cell.Row
        j = cell.Col
        w = self.w
        if robot == None:
            if color == 1:
                rgb_color = (160, 160, 160)
            if color == 2:
                rgb_color = (0, 0, 0)
            start_point = (w * j + self.thickness, w * i + self.thickness)
            end_point =  (w * (j + 1) - self.thickness, w * (i + 1) - self.thickness)
            cv.rectangle(self.img, start_point, end_point, rgb_color, -1)
        else:
            color = robot.rgb
            dirr = robot.direction
            r = cell.Row
            c = cell.Col
            center = (int(w * c + w / 2), int(w * r + w / 2))
            if r:
                cv.circle(self.img, center, int(w / 3), color, self.thickness)
                if dirr == 0:
                    rr, cc = -1, 0
                elif dirr == 1:
                    rr, cc = 0, -1
                elif dirr == 2:
                    rr, cc = 1, 0
                else:
                    rr, cc = 0, 1
                cent = (int(center[0] + rr * w / 7), int(center[1] + cc * w / 7))
                cv.circle(self.img, cent, int(w / 8), color, self.thickness)
        self.present()

    def visualize(self):
        start = time.time()
        w = self.w
        thickness = 2
        maze = self.maze
        self.img = np.zeros([(Row + robot_count + 1) * w, (Col + 3) * w, 3], dtype = np.uint8)
```

```
        self.img.fill(255)
        if maze:
            Row = len(maze.cell)
            Col = len(maze.cell[0])
            img = np.zeros([(Row + self.robot_count + 1) * w, (Col + 3) * w,3], dtype = np.uint8)
            img.fill(255)
            for i in range(Row):
                for j in range(Col):
                    if maze.cell[i][j].downWall == 0:
                        x1 = w * j
                        y1 = w * (i + 1)
                        x2 = w * (j + 1)
                        y2 = w * (i + 1)
                        cv.line(img, (x1, y1), (x2, y2), (255, 0, 0), thickness, lineType = 8)
                    if maze.cell[i][j].rightWall == 0:
                        x1 = w * (j + 1)
                        y1 = w * i
                        x2 = w * (j + 1)
                        y2 = w * (i + 1)
                        cv.line(img, (x1, y1), (x2, y2), (255, 0, 0), thickness, lineType = 8)
                    if maze.cell[i][j].color == 1:
                        start_point = (w * j + thickness, w * i + thickness)
                        end_point =  (w * (j + 1) - thickness, w * (i + 1) - thickness)
                        cv.rectangle(img, start_point, end_point, (160, 160, 160), -1)
                    if maze.cell[i][j].color == 2:
                        start_point = (w * j + thickness, w * i + thickness)
                        end_point =  (w * (j + 1) - thickness, w * (i + 1) - thickness)
                        cv.rectangle(img, start_point, end_point, (0, 0, 0), -1)
            count = 0
            for robot in self.robots:
                if robot:
                    color = robot.rgb
                    dirr = robot.direction
                    txt = str(robot.color) + ": current cell: " + str(robot.RowInd) + " " + str(robot.ColInd) + " "
                    font = cv.FONT_HERSHEY_SIMPLEX
                    cv.putText(img, txt,(w,(Row + 1 + count) * w), font, .5, (0, 0, 0), 1, cv.LINE_AA)
                    r = robot.CurrentCell(maze).Row
                    c = robot.CurrentCell(maze).Col
                    center = (int(w * c + w / 2), int(w * r + w / 2))
                    if r:
                        cv.circle(img, center, int(w / 3), color, thickness)
                        if dirr == 0:
```

```
                    rr, cc = -1, 0
                elif dirr == 1:
                    rr, cc = 0, -1
                elif dirr == 2:
                    rr, cc = 1, 0
                else:
                    rr, cc = 0, 1
                cent = (int(center[0] + rr * w / 7), int(center[1] + cc * w / 7))
                cv.circle(img, cent, int(w / 8), color, thickness)
            count += 1
        if self.name == "global":
            cv.imshow(self.name, img)
            cv.moveWindow(self.name,1000,60)
            cv.waitKey(1)
        else:
            cv.moveWindow(self.name,100,60)
            cv.imshow(self.name, img)
            cv.waitKey(1)
```

Appendix B

Physical Test Codes

Listing B.1: ROS inputted script

```python
#!/usr/bin/env python

import rospy
import threading
from std_msgs.msg import String
from nav_msgs.msg import Odometry
from sensor_msgs.msg import LaserScan
from sensor_msgs.msg import Range, Image
from geometry_msgs.msg import Point, Twist
from ar_track_alvar_msgs.msg import AlvarMarkers
from tf.transformations import euler_from_quaternion
from dijkstar import Graph, find_path
import math
import random
import time
import sys
import numpy as np
from copy import copy
from datetime import datetime
from collections import defaultdict
import collections
import marshal
from cv_bridge import CvBridge
sys.path.remove('/opt/ros/melodic/lib/python2.7/dist-packages') # in order to import cv2 under python3
import cv2 as cv
sys.path.append('/opt/ros/melodic/lib/python2.7/dist-packages') # append back in order to import


right_up = 1
right_down = 0
left_up = 8
left_down = 2

def SlopeDeg(z1, y1, z2, y2):
```

```
    m = (z2 - z1) / (y2 - y1)
    slopedeg = math.degrees(math.atan(m))
    if slopedeg < -60:
        slopedeg += 180
    return slopedeg

def interpolation(pos, pos1, pos2, m1, m2):
    return ((pos - pos1) / (pos2 - pos1)) * (m2 - m1) + m1

def Average(lst):
    return sum(lst) / len(lst)

class Cell:
    def __init__(self, Row, Col):
        self.Row = Row
        self.Col = Col
        self.id = Row * 10 + Col
        self.xpos = 0
        self.ypos = 0
        self.downWall = 1 # 1 for way
        self.rightWall = 1 # 0 for wall
        self.color = 0
        self.parent = None
        self.tag = 1
        self.changed_x = 0
        self.changed_y = 0

class Maze:
    def __init__(self, Row, Col, StartRow = 0, StartCol = 0, FinishRow = 1, FinishCol = 1):
        self.StartRow = StartRow
        self.StartCol = StartCol
        self.FinishRow = FinishRow
        self.FinishCol = FinishCol
        self.cell = [[Cell(row, col) for col in range(Col)] for row in range(Row)]
        self.Row = Row
        self.Col = Col
        self.graph = Graph()

    def add_to_graph(self, cell_u, cell_v):
        self.graph.add_edge(cell_u, cell_v, 1)
        self.graph.add_edge(cell_v, cell_u, 1)

    def remove_from_graph(self, cell_u, cell_v):
        self.graph.remove_edge(cell_u, cell_v)
        self.graph.remove_edge(cell_v, cell_u)


    def cell_ref(self, i, j):
        return self.cell[i][j]

    def present(self):
        matrix = [[(self.cell[i][j].xpos, self.cell[i][j].ypos) for j in
```

```
            range(self.Col)] for i in range(self.Row)]
        rospy.loginfo("maze dimention:")
        rospy.loginfo(matrix)

class Robot:
    def __init__(self, id, color, rgb, dist_treshhold = 3.8):
        self.Xpos = 0
        self.Ypos = 0
        self.Xorient = 0
        self.Yorient = 0
        self.Zorient = 0
        self.Worient = 0

        self.roll = 0
        self.pitch = 0
        self.theta = 0
        self.alpha = 0
        self.RowInd = 0
        self.ColInd = 0


        self.prox0 = 0
        self.Qprox0 = []
        self.prox0first = 0
        self.prox0Num = 0
        self.prox0MaxSize = 3

        self.prox7 = 0
        self.Qprox7 = []
        self.prox7first = 0
        self.prox7Num = 0
        self.prox7MaxSize = 3

        self.dist = 0
        self.Qdist = []
        self.distfirst = 0
        self.distNum = 0
        self.distMaxSize = 2


        self.prox2 = 0
        self.Qprox2 = []
        self.prox2first = 0
        self.prox2Num = 0
        self.prox2MaxSize = 6

        self.prox5 = 0
        self.Qprox5 = []
        self.prox5first = 0
        self.prox5Num = 0
        self.prox5MaxSize = 6

        self.Sensor_treshhold = 4.7
        self.dist_treshhold = dist_treshhold
```

```
        self.WallMargin = 0.025
        self.direction = 0 # 0 left # 1 up # 2 right # 3 down
        self.InitializeFlag = 0

        self.MH = 0
        self.MV = 90

        self.odom_flag = 0 # if 0 set from camera/ if 1 set from odom
        self.id = str(id)
        self.RowInd = 0
        self.ColInd = 0
        self.visited = defaultdict(int)
        self.color = color
        self.velocity_publisher = None
        self.rgb = rgb

    def CurrentCell(self, maze):
        return maze.cell[self.RowInd][self.ColInd]

    def LeftCell(self, maze):
        return maze.cell[self.RowInd][self.ColInd - 1]

    def UpCell(self, maze):
        return maze.cell[self.RowInd - 1][self.ColInd]

    def RightCell(self, maze):
        return maze.cell[self.RowInd][self.ColInd + 1]

    def RightCell_robot(self, maze): # done
        if self.direction == 0: #left
            return self.UpCell(maze)
        if self.direction == 1: #up
            return self.RightCell(maze)
        if self.direction == 2: #right
            return self.DownCell(maze)
        if self.direction == 3: #down
            return self.LeftCell(maze)

    def DownCell(self, maze):
        return maze.cell[self.RowInd + 1][self.ColInd]

    def ParentCell(self, maze):
        return maze.cell[self.RowInd][self.ColInd].parent

    def frontCell(self, maze):
        if self.direction == 0:
            return self.LeftCell(maze)
        if self.direction == 1:
            return self.UpCell(maze)
        if self.direction == 2:
            return self.RightCell(maze)
        if self.direction == 3:
            return self.DownCell(maze)

```

```
    def front_wall(self, maze):
        if self.direction == 0: # left
            return self.LeftCell(maze).rightWall
        if self.direction == 1: # up
            return self.UpCell(maze).downWall
        if self.direction == 2: # right
            return self.CurrentCell(maze).rightWall
        if self.direction == 3: # down
            return self.CurrentCell(maze).downWall

    def left_wall(self, maze):
        if self.direction == 0: # left
            return self.CurrentCell(maze).downWall
        if self.direction == 1: # up
            return self.LeftCell(maze).rightWall
        if self.direction == 2: # right
            return self.UpCell(maze).downWall
        if self.direction == 3: # down
            return self.CurrentCell(maze).rightWall

    def right_wall(self, maze):
        if self.direction == 0: # left
            return self.UpCell(maze).downWall
        if self.direction == 1: # up
            return self.CurrentCell(maze).rightWall
        if self.direction == 2: # right
            return self.CurrentCell(maze).downWall
        if self.direction == 3: # down
            return self.LeftCell(maze).rightWall

    def LeftSensor(self):
        return self.prox5

    def RightSensor(self):
        return self.prox2

    def FrontProximity(self):
        return self.dist

    def LeftAlignSensor(self):
        return self.prox7

    def RightAlignSensor(self):
        return self.prox0

    def FrontSensor(self):
        return self.dist

    def vel_assign(self, velocity_publisher):
        self.velocity_publisher = velocity_publisher

    def distance_callback(self, data):
        if self.distNum < self.distMaxSize:
            self.Qdist.append(data.range)
```

```
                self.distNum += 1
            else:
                self.Qdist[self.distfirst] = data.range
                self.distfirst = (self.distfirst + 1) % self.distMaxSize
                self.distNum += 1
            self.dist = (sum(self.Qdist) / len(self.Qdist)) * 100

        def prox0_callback(self, data):
            if self.prox0Num < self.prox0MaxSize:
                self.Qprox0.append(data.range)
                self.prox0Num += 1
            else:
                self.Qprox0[self.prox0first] = data.range
                self.prox0first = (self.prox0first + 1) % self.prox0MaxSize
                self.prox0Num += 1
            self.prox0 = 100 * sum(self.Qprox0) / len(self.Qprox0)

        def prox2_callback(self, data):
            if self.prox2Num < self.prox2MaxSize:
                self.Qprox2.append(data.range)
                self.prox2Num += 1
            else:
                self.Qprox2[self.prox2first] = data.range
                self.prox2first = (self.prox2first + 1) % self.prox2MaxSize
                self.prox2Num += 1
            self.prox2 = 100 * sum(self.Qprox2) / len(self.Qprox2)

        def prox5_callback(self, data):
            if self.prox5Num < self.prox5MaxSize:
                self.Qprox5.append(data.range)
                self.prox5Num += 1
            else:
                self.Qprox5[self.prox5first] = data.range
                self.prox5first = (self.prox5first + 1) % self.prox5MaxSize
                self.prox5Num += 1
            self.prox5 = 100 * sum(self.Qprox5) / len(self.Qprox5)

        def prox7_callback(self, data):
            if self.prox7Num < self.prox7MaxSize:
                self.Qprox7.append(data.range)
                self.prox7Num += 1
            else:
                self.Qprox7[self.prox7first] = data.range
                self.prox7first = (self.prox7first + 1) % self.prox7MaxSize
                self.prox7Num += 1
            self.prox7 = 100 * sum(self.Qprox7) / len(self.Qprox7)

        def align_0(self, speed = .4):
            start = time.time()
            while time.time() - start < 3:
                vel_msg = Twist()
                vel_msg.linear.x = 0
                vel_msg.linear.y = 0
                vel_msg.linear.z = 0
```

```
            vel_msg.angular.x = 0
            vel_msg.angular.y = 0
            if self.theta < 0:
                vel_msg.angular.z = abs(speed) / 4
            else:
                vel_msg.angular.z = - abs(speed) / 4
            self.velocity_publisher.publish(vel_msg)


    def align_1(self, speed = .4):
        start = time.time()
        while time.time() - start < 3:
            vel_msg = Twist()
            vel_msg.linear.x = 0
            vel_msg.linear.y = 0
            vel_msg.linear.z = 0
            vel_msg.angular.x = 0
            vel_msg.angular.y = 0
            if self.theta < -90:
                vel_msg.angular.z = abs(speed) / 4
            else:
                vel_msg.angular.z = -abs(speed) / 4
            self.velocity_publisher.publish(vel_msg)

    def align_3(self, speed = .4):
        start = time.time()
        while time.time() - start < 3:
            vel_msg = Twist()
            vel_msg.linear.x = 0
            vel_msg.linear.y = 0
            vel_msg.linear.z = 0
            vel_msg.angular.x = 0
            vel_msg.angular.y = 0
            if self.theta < 90:
                vel_msg.angular.z = abs(speed) / 4
            else:
                vel_msg.angular.z = -abs(speed) / 4
            self.velocity_publisher.publish(vel_msg)


    def align_2(self, speed = .4):
        start = time.time()
        while time.time() - start < 3:
            vel_msg = Twist()
            vel_msg.linear.x = 0
            vel_msg.linear.y = 0
            vel_msg.linear.z = 0
            vel_msg.angular.x = 0
            vel_msg.angular.y = 0
            if self.theta < 0:
                vel_msg.angular.z = -abs(speed) / 4
            else:
                vel_msg.angular.z = abs(speed) / 4
            self.velocity_publisher.publish(vel_msg)
```

```
    def TurnLeft(self, speed = .4, degree = 180):
        if self.direction == 0:
            final_dir = 3
        if self.direction == 1:
            final_dir = 0
        if self.direction == 2:
            final_dir = 1
        if self.direction == 3:
            final_dir = 2
        rospy.loginfo(str(self.id) + " turn left")
        oldTheta = self.theta
        AbsDiffDeg = 0
        while AbsDiffDeg < degree and not rospy.is_shutdown():
            middleTheta = self.theta
            vel_msg = Twist()
            vel_msg.linear.x = 0
            vel_msg.linear.y = 0
            vel_msg.linear.z = 0
            vel_msg.angular.x = 0
            vel_msg.angular.y = 0
            vel_msg.angular.z = abs(speed)
            normDeg = (oldTheta - self.theta) % 360
            AbsDiffDeg = min(normDeg, 360 - normDeg)
            left = 5.5
            right = 0
            if self.alpha > 76 and self.direction == final_dir and AbsDiffDeg > 45:
                self.align_0(speed)
                rospy.loginfo(str(self.id) + " breakx0left " + "new: " + str(self.theta) + " old: " + str(oldTheta) + " dif: " + str(AbsDiffDeg))
                break
            if final_dir == 1:
                if abs(self.alpha) < 13 and self.direction == final_dir and AbsDiffDeg > 45:
                    self.align_1(speed)
                    rospy.loginfo(str(self.id) + " breaky0up " + "new: " + str(self.theta) + " old: " + str(oldTheta) + " dif: " + str(AbsDiffDeg))
                    break
            if final_dir == 3:
                if abs(self.alpha) < 13 and self.direction == final_dir and AbsDiffDeg > 45:
                    self.align_3(speed)
                    rospy.loginfo(str(self.id) + " breaky0down " + "new: " + str(self.theta) + " old: " + str(oldTheta) + " dif: " + str(AbsDiffDeg))
                    break
            if self.alpha < -76 and self.direction == final_dir and AbsDiffDeg > 45:
                self.align_2(speed)
```

```
                rospy.loginfo(str(self.id) + " breakx0right " + "new: " + str(self.theta) + " old: " + str(oldTheta) + " dif: " + str(AbsDiffDeg))
                break
            if ((self.alpha < -71) or (71 < self.alpha) or abs(self.alpha) < 19) and AbsDiffDeg > 45:
                if (self.alpha < -87) or (87 < self.alpha) or (abs(self.alpha) < 1.5):
                    rospy.loginfo(str(self.id) + " breakalpha " + "new: " + str(self.theta) + " old: " + str(oldTheta) + " dif: " + str(AbsDiffDeg))
                    break
                else:
                    vel_msg.angular.z = abs(speed) / 3
                    rospy.loginfo(str(self.id) + " fine " + str(self.alpha) + " " + str(self.theta))
                    time.sleep(.03)
            self.velocity_publisher.publish(vel_msg)
        self.move_stop()
        rospy.loginfo(str(self.id) + " " + str(self.alpha))
        time.sleep(.1)

    def TurnRight(self, speed = .4, degree = 180):
        if self.direction == 0:
            final_dir = 1
        if self.direction == 1:
            final_dir = 2
        if self.direction == 2:
            final_dir = 3
        if self.direction == 3:
            final_dir = 0
        rospy.loginfo(str(self.id) + " turn right")
        oldTheta = self.theta
        AbsDiffDeg = 0
        while AbsDiffDeg < degree and not rospy.is_shutdown():
            middleTheta = self.theta
            vel_msg = Twist()
            vel_msg.linear.x = 0
            vel_msg.linear.y = 0
            vel_msg.linear.z = 0
            vel_msg.angular.x = 0
            vel_msg.angular.y = 0
            vel_msg.angular.z = -abs(speed)
            normDeg = (oldTheta - self.theta) % 360
            AbsDiffDeg = min(normDeg, 360 - normDeg)
            left = 0
            right = 5.5
            if self.alpha > 76 and self.direction == final_dir and AbsDiffDeg > 45:
                self.align_0(speed)
                rospy.loginfo(str(self.id) + " breakx0left " + "new: " + str(self.theta) + " old: " + str(oldTheta) + " dif: " + str(AbsDiffDeg))
                break
```

```python
            if final_dir == 1:
                if abs(self.alpha) < 13 and self.direction == final_dir and AbsDiffDeg > 45:
                    self.align_1(speed)
                    rospy.loginfo(str(self.id) + " breaky0up " + "new: " + str(self.theta) + " old: " + str(oldTheta) + " dif: " + str(AbsDiffDeg))
                    break
            if final_dir == 3:
                if abs(self.alpha) < 13 and self.direction == final_dir and AbsDiffDeg > 45:
                    self.align_3(speed)
                    rospy.loginfo(str(self.id) + " breaky0down " + "new: " + str(self.theta) + " old: " + str(oldTheta) + " dif: " + str(AbsDiffDeg))
                    break
            if self.alpha < -76 and self.direction == final_dir and AbsDiffDeg > 45:
                self.align_2(speed)
                rospy.loginfo(str(self.id) + " breakx0right " + "new: " + str(self.theta) + " old: " + str(oldTheta) + " dif: " + str(AbsDiffDeg))
                break
            if ((self.alpha < -71) or (71 < self.alpha) or abs(self.alpha) < 19) and AbsDiffDeg > 45:
                if (self.alpha < -86) or (86 < self.alpha) or (abs(self.alpha) < 1.5):
                    rospy.loginfo(str(self.id) + " breakalpha " + "new: " + str(self.theta) + " old: " + str(oldTheta) + " dif: " + str(AbsDiffDeg))
                    break
                else:
                    vel_msg.angular.z = -abs(speed) / 3
                    rospy.loginfo(str(self.id) + " fine " + str(self.alpha) + " " + str(self.theta))
                    time.sleep(.01)
            self.velocity_publisher.publish(vel_msg)
        self.move_stop()
        rospy.loginfo(str(self.id) + " " + str(self.alpha))
        time.sleep(.1)

    def euclidean_distance(self, front_cell):
        r = abs(front_cell.Row - self.RowInd)
        c = abs(front_cell.Col - self.ColInd)
        return math.sqrt(math.pow(c * (front_cell.xpos - self.Xpos), 2) + math.pow(r * (front_cell.ypos - self.Ypos), 2))

    def linear_vel(self, front_cell, speed = .5):
        l = speed * self.euclidean_distance(front_cell)
        if l > 3: return 2.3
        elif l < 3: return 1
        else: return l

    def steer_angle2(self, front_cell):
```

```
        if self.direction == 0:
            return -self.theta
        if self.direction == 1:
            return (-90 - self.theta)
        if self.direction == 3:
            return (90 - self.theta)
        if self.direction == 2:
            if self.theta > 0:
                return (180 - self.theta)
            else:
                return -(self.theta + 180)

    def angular_vel2(self, front_cell, speed = 1):
        return speed * self.steer_angle2(front_cell) / 100

    def move_to_cell(self, front_cell, maze, lock, speed):
        distance_tolerance = .5
        while self.euclidean_distance(front_cell) >= distance_tolerance and not rospy.is_shutdown() and self.dist > 4:
            vel_msg = Twist()
            vel_msg.linear.x = self.linear_vel(front_cell)
            vel_msg.linear.y = 0
            vel_msg.linear.z = 0
            if vel_msg.linear.x > 1: #2
                m = 1
            else:
                m = 0
            vel_msg.angular.x = 0
            vel_msg.angular.y = 0
            if self.LeftAlignSensor() < 2 or self.LeftSensor() < 2.1:
                vel_msg.angular.z = -0.1
                if self.LeftAlignSensor() < .3:
                    vel_msg.linear.x = 0
            elif self.RightSensor() < 2.1 or self.RightAlignSensor() < 2:
                vel_msg.angular.z = 0.1
                if self.RightAlignSensor() < .3:
                    vel_msg.linear.x = 0
            else:
                vel_msg.angular.z = self.angular_vel2(front_cell) * m
            self.velocity_publisher.publish(vel_msg)
        self.RowInd = front_cell.Row
        self.ColInd = front_cell.Col
        lock.acquire()
        if maze.cell[self.RowInd][self.ColInd].changed_x == 0:
            maze.cell[self.RowInd][self.ColInd].xpos = self.Xpos
            maze.cell[self.RowInd][self.ColInd].changed_x = 1
        if maze.cell[self.RowInd][self.ColInd].changed_y == 0:
            maze.cell[self.RowInd][self.ColInd].ypos = self.Ypos
            maze.cell[self.RowInd][self.ColInd].changed_y = 1
            for i in range(1, 11):
                if maze.cell[self.RowInd][i].changed_y == 0:
                    maze.cell[self.RowInd][i].ypos = self.Ypos
                    maze.cell[self.RowInd][i].changed_y = 1
```

```python
                if maze.cell[i][self.ColInd].changed_x == 0:
                    maze.cell[i][self.ColInd].xpos = self.Xpos
                    maze.cell[i][self.ColInd].changed_x = 1
        lock.release()
        if self.dist_treshhold < self.dist < 6 and not (self.front_wall(maze) == 1 and self.frontCell(maze).color >= 10):
            #rospy.loginfo('align')
            while self.dist > (self.dist_treshhold + .0) and not rospy.is_shutdown():
                vel_msg = Twist()
                vel_msg.linear.x = self.linear_vel(front_cell)
                vel_msg.linear.y = 0
                vel_msg.linear.z = 0
                vel_msg.angular.x = 0
                vel_msg.angular.y = 0
                vel_msg.angular.z = 0
                self.velocity_publisher.publish(vel_msg)
        rospy.loginfo(str(self.id) + " dist is: " + str(self.dist))

    def ForwardOneCell(self, maze, lock, speed = 1):
        if self.direction == 0:
            final_dir = 0
        if self.direction == 1:
            final_dir = 1
        if self.direction == 2:
            final_dir = 2
        if self.direction == 3:
            final_dir = 3
        rospy.loginfo(str(self.id) + ' forward one cell, dir is ' + str(self.direction))
        front_cell = self.frontCell(maze)
        self.move_to_cell(front_cell, maze, lock, speed)
        if final_dir == 0:
            self.align_0()
        if final_dir == 1:
            self.align_1()
        if final_dir == 2:
            self.align_2()
        if self.direction == 3:
            self.align_3()
        self.move_stop()
        time.sleep(1)

    def initializePos(self, maze, r, c):
        self.RowInd = r
        self.ColInd = c
        self.CurrentCell(maze).color = self.color
        self.add_to_visited_cell(self.CurrentCell(maze))
        rospy.loginfo(str(self.id) + ' initialize done')

    def explore(self, maze, lock):
        lock.acquire()
        self.AssignWall(maze)
        next_Cell = self.Choose_direction(maze)
```

```
        lock.release()
        rospy.loginfo(str(self.id) + ' next cell is ' + str(next_Cell.
           Row) + '  ' + str(next_Cell.Col))
        time.sleep(1)
        self.Move(maze, next_Cell, lock)

    def double_Check(self):
        if self.FrontSensor() < self.Sensor_treshhold * 2:
            rospy.loginfo(str(self.id) + " sorry")
            return False
        else:
            return True

    def add_to_visited_cell(self, next_Cell):
        self.visited[next_Cell.id] += 1

    def Move(self, maze, next_Cell, lock):
        start_cell = self.CurrentCell(maze)
        finish_cell = next_Cell
        if next_Cell == self.CurrentCell(maze):
            rospy.loginfo(str(self.id) + " don't move")
            return
        next_dir = self.next_cell_direction(maze, next_Cell)
        rospy.loginfo(str(self.id) + ' next dir is ' + next_dir)
        lock.acquire()
        start_cell.tag = 0
        finish_cell.tag = 0
        lock.release()
        if next_dir == 'left':
            self.TurnLeft()
            if self.double_Check():
                lock.acquire()
                self.color_assign(maze, next_Cell)
                maze.add_to_graph(self.CurrentCell(maze), next_Cell)
                lock.release()
                self.ForwardOneCell(maze, lock)
                self.add_to_visited_cell(next_Cell)
            else:
                lock.acquire()
                finish_cell.tag = 1
                self.AssignWall(maze)
                lock.release()
                self.TurnRight()
                self.Move(maze, self.Choose_direction(maze), lock)
        if next_dir == 'right':
            self.TurnRight()
            if self.double_Check():
                lock.acquire()
                self.color_assign(maze, next_Cell)
                maze.add_to_graph(self.CurrentCell(maze), next_Cell)
                lock.release()
                self.ForwardOneCell(maze, lock)
                self.add_to_visited_cell(next_Cell)
            else:
```

```
                lock.acquire()
                finish_cell.tag = 1
                self.AssignWall(maze)
                lock.release()
                self.TurnLeft()
                self.Move(maze, self.Choose_direction(maze), lock)
        if next_dir == 'forward':
            lock.acquire()
            self.color_assign(maze, next_Cell)
            maze.add_to_graph(self.CurrentCell(maze), next_Cell)
            lock.release()
            self.ForwardOneCell(maze, lock)
            self.add_to_visited_cell(next_Cell)
        if next_dir == 'backward':
            self.TurnLeft()
            self.TurnLeft()
            if self.double_Check():
                lock.acquire()
                self.color_assign(maze, next_Cell)
                lock.release()
                self.ForwardOneCell(maze, lock)
                self.add_to_visited_cell(next_Cell)
            else:
                lock.acquire()
                finish_cell.tag = 1
                self.AssignWall(maze)
                lock.release()
                self.TurnLeft()
                self.TurnLeft()
                self.Move(maze, self.Choose_direction(maze), lock)
        lock.acquire()
        start_cell.tag = 1
        lock.release()
        self.move_stop()

    def move_stop(self, xx = 0):
        vel_msg = Twist()
        vel_msg.linear.x = xx
        vel_msg.linear.y = 0
        vel_msg.linear.z = 0
        vel_msg.angular.x = 0
        vel_msg.angular.y = 0
        vel_msg.angular.z = 0
        self.velocity_publisher.publish(vel_msg)
        rospy.loginfo(str(self.id) + " finish " + str(self.Xpos) + " " +
            str(self.Ypos))

    def color_assign(self, maze, next_Cell):
        if next_Cell.color == 0:
            next_Cell.parent = self.CurrentCell(maze)
        if self.dir_sum(maze) == 1:
            self.dead_end(maze)
            next_Cell.color = self.color
        else:
```

```python
            self.CurrentCell(maze).color = 1
            next_Cell.color = self.color

    def dead_end(self, maze):
        self.CurrentCell(maze).color = 2
        self.CurrentCell(maze).downWall = 0
        self.CurrentCell(maze).rightWall = 0
        self.UpCell(maze).downWall = 0
        self.LeftCell(maze).rightWall = 0
        rospy.loginfo(str(self.id) + ' dead end reached: return to parent')

    def dir_sum(self, maze):
        return self.CurrentCell(maze).downWall + self.CurrentCell(maze).rightWall + self.UpCell(maze).downWall + self.LeftCell(maze).rightWall

    def next_cell_direction(self, maze, next_Cell):
        if self.direction == 0: # left
            if next_Cell.Row == self.RowInd and next_Cell.Col == self.ColInd - 1:
                return 'forward'
            if next_Cell.Row == self.RowInd and next_Cell.Col == self.ColInd + 1:
                return 'backward'
            if next_Cell.Row == self.RowInd + 1 and next_Cell.Col == self.ColInd:
                return 'left'
            if next_Cell.Row == self.RowInd - 1 and next_Cell.Col == self.ColInd:
                return 'right'
        if self.direction == 1: # up
            if next_Cell.Row == self.RowInd - 1 and next_Cell.Col == self.ColInd:
                return 'forward'
            if next_Cell.Row == self.RowInd + 1 and next_Cell.Col == self.ColInd:
                return 'backward'
            if next_Cell.Row == self.RowInd and next_Cell.Col == self.ColInd - 1:
                return 'left'
            if next_Cell.Row == self.RowInd and next_Cell.Col == self.ColInd + 1:
                return 'right'
        if self.direction == 2: # right
            if next_Cell.Row == self.RowInd and next_Cell.Col == self.ColInd + 1:
                return 'forward'
            if next_Cell.Row == self.RowInd and next_Cell.Col == self.ColInd - 1:
                return 'backward'
            if next_Cell.Row == self.RowInd - 1 and next_Cell.Col == self.ColInd:
                return 'left'
```

```
            if next_Cell.Row == self.RowInd + 1 and next_Cell.Col == self.ColInd:
                return 'right'
        if self.direction == 3: # down
            if next_Cell.Row == self.RowInd + 1 and next_Cell.Col == self.ColInd:
                return 'forward'
            if next_Cell.Row == self.RowInd - 1 and next_Cell.Col == self.ColInd:
                return 'backward'
            if next_Cell.Row == self.RowInd and next_Cell.Col == self.ColInd + 1:
                return 'left'
            if next_Cell.Row == self.RowInd and next_Cell.Col == self.ColInd - 1:
                return 'right'

    def Assign_LeftWall(self, maze):
        if self.LeftCell(maze).rightWall and self.LeftCell(maze).tag == 1:
            self.LeftCell(maze).rightWall = 0
            rospy.loginfo(str(self.id) + " assign leftwall " + str(self.FrontSensor()))

    def Assign_UpWall(self, maze):
        if self.UpCell(maze).downWall and self.UpCell(maze).tag == 1:
            self.UpCell(maze).downWall = 0
            rospy.loginfo(str(self.id) + " assign upwall " + str(self.RightSensor()))

    def Assign_RightWall(self, maze):
        if self.CurrentCell(maze).rightWall and self.RightCell(maze).tag == 1:
            self.CurrentCell(maze).rightWall = 0
            rospy.loginfo(str(self.id) + " assign rightwall " + str(self.RightSensor()))

    def Assign_DownWall(self, maze):
        if self.CurrentCell(maze).downWall and self.DownCell(maze).tag == 1:
            self.CurrentCell(maze).downWall = 0
            rospy.loginfo(str(self.id) + " assign downwall " + str(self.LeftSensor()))

    def AssignWall(self, maze):
        if self.direction == 0: # left
            if self.LeftSensor() < self.Sensor_treshhold: # downwall
                self.Assign_DownWall(maze)
            if self.RightSensor() < self.Sensor_treshhold: # upwall
                self.Assign_UpWall(maze)
            if self.FrontSensor() < self.Sensor_treshhold * 2: # leftwall
                self.Assign_LeftWall(maze)
        if self.direction == 1: # up
```

```
            if self.LeftSensor() < self.Sensor_treshhold: # leftwall
                self.Assign_LeftWall(maze)
            if self.RightSensor() < self.Sensor_treshhold: # rightwall
                self.Assign_RightWall(maze)
            if self.FrontSensor() < self.Sensor_treshhold * 2: # upwall
                self.Assign_UpWall(maze)
        if self.direction == 2: # right
            if self.LeftSensor() < self.Sensor_treshhold: # upwall
                self.Assign_UpWall(maze)
            if self.RightSensor() < self.Sensor_treshhold: # downwall
                self.Assign_DownWall(maze)
            if self.FrontSensor() < self.Sensor_treshhold * 2: #
                rightwall
                self.Assign_RightWall(maze)
        if self.direction == 3: # down
            if self.LeftSensor() < self.Sensor_treshhold: # rightwall
                self.Assign_RightWall(maze)
            if self.RightSensor() < self.Sensor_treshhold: # leftwall
                self.Assign_LeftWall(maze)
            if self.FrontSensor() < self.Sensor_treshhold * 2: #
                downwall
                self.Assign_DownWall(maze)

    def sort_cell_list(self, cell_list): #done
        n = len(cell_list)
        for i in range(n):
            for j in range(n - i - 1):
                if self.visited[cell_list[j].id] > self.visited[
                    cell_list[j + 1].id]:
                    cell_list[j], cell_list[j + 1] = cell_list[j + 1],
                        cell_list[j]
        new_list = [cell_list[0]]
        val = self.visited[cell_list[0]]
        for i in range(1, n):
            if self.visited[cell_list[i]] == val:
                new_list.append(cell_list[i])
        return new_list

    def Choose_direction(self, maze):
        white_cells = []
        grey_cells_visited = []
        grey_cells_not_visited = []
        rospy.loginfo(str(self.id) + ' choosing direction: cur row '+
            str(self.CurrentCell(maze).Row) + ' cur col '+ str(self.
            CurrentCell(maze).Col))
        if self.LeftCell(maze).rightWall == 1 and self.LeftCell(maze).
            tag:
            if self.LeftCell(maze).color == 0 : # left
                white_cells.append(self.LeftCell(maze))
            if self.LeftCell(maze).color == 1: #
                if self.LeftCell(maze).id in self.visited:
                    grey_cells_visited.append(self.LeftCell(maze))
                else:
                    grey_cells_not_visited.append(self.LeftCell(maze))
```

```
        if self.CurrentCell(maze).rightWall == 1 and self.RightCell(maze).tag:
            if self.RightCell(maze).color == 0:  # right
                white_cells.append(self.RightCell(maze))
            if self.RightCell(maze).color == 1:  # right
                if self.RightCell(maze).id in self.visited:
                    grey_cells_visited.append(self.RightCell(maze))
                else:
                    grey_cells_not_visited.append(self.RightCell(maze))
        if self.CurrentCell(maze).downWall == 1 and self.DownCell(maze).tag:
            if self.DownCell(maze).color == 0: # down
                white_cells.append(self.DownCell(maze))
            if self.DownCell(maze).color == 1: # down
                if self.DownCell(maze).id in self.visited:
                    grey_cells_visited.append(self.DownCell(maze))
                else:
                    grey_cells_not_visited.append(self.DownCell(maze))
        if self.UpCell(maze).downWall == 1 and self.UpCell(maze).tag:
            if self.UpCell(maze).color == 0: # up
                white_cells.append(self.UpCell(maze))
            if self.UpCell(maze).color == 1: # up
                if self.UpCell(maze).id in self.visited:
                    grey_cells_visited.append(self.UpCell(maze))
                else:
                    grey_cells_not_visited.append(self.UpCell(maze))
        if self.LeftCell(maze).rightWall == 1 and self.LeftCell(maze).tag:
            if self.LeftCell(maze).color == 0 : # left
                white_cells.append(self.LeftCell(maze))
            if self.LeftCell(maze).color == 1: #
                if self.LeftCell(maze).id in self.visited:
                    grey_cells_visited.append(self.LeftCell(maze))
                else:
                    grey_cells_not_visited.append(self.LeftCell(maze))
        if self.CurrentCell(maze).rightWall == 1 and self.RightCell(maze).tag:
            if self.RightCell(maze).color == 0:  # right
                white_cells.append(self.RightCell(maze))
            if self.RightCell(maze).color == 1:  # right
                if self.RightCell(maze).id in self.visited:
                    grey_cells_visited.append(self.RightCell(maze))
                else:
                    grey_cells_not_visited.append(self.RightCell(maze))
        if self.CurrentCell(maze).downWall == 1 and self.DownCell(maze).tag:
            if self.DownCell(maze).color == 0: # down
                white_cells.append(self.DownCell(maze))
            if self.DownCell(maze).color == 1: # down
                if self.DownCell(maze).id in self.visited:
                    grey_cells_visited.append(self.DownCell(maze))
                else:
                    grey_cells_not_visited.append(self.DownCell(maze))
        if self.UpCell(maze).downWall == 1 and self.UpCell(maze).tag:
```

```python
            if self.UpCell(maze).color == 0: # up
                white_cells.append(self.UpCell(maze))
            if self.UpCell(maze).color == 1: # up
                if self.UpCell(maze).id in self.visited:
                    grey_cells_visited.append(self.UpCell(maze))
                else:
                    grey_cells_not_visited.append(self.UpCell(maze))
        if len(white_cells):
            rospy.loginfo(str(self.id) + " whites: ")
            for celll in white_cells:
                rospy.loginfo(str(celll.Row) + " " + str(celll.Col))
            rospy.loginfo(str(self.id) + " white_done")
        rospy.loginfo(str(self.id) + " g not vis: " + str([x.id for x in grey_cells_not_visited]) + " " + str(len(grey_cells_not_visited)))
        rospy.loginfo(str(self.id) + " g vis: " + str([(x.id, self.visited[x.id]) for x in grey_cells_visited]) + " " + str(len(grey_cells_visited)))
        if len(white_cells) > 0:
            next_Cell = random.choice(white_cells)
            if self.id == "5":
                if self.RightCell_robot(maze) in white_cells:
                    rospy.loginfo("eee")
                    next_Cell = self.RightCell_robot(maze)
            if maze.cell_ref(5, 7) in white_cells: next_Cell = maze.cell_ref(5, 7)
            if maze.cell_ref(10, 5) in white_cells: next_Cell = maze.cell_ref(10, 5)
            if maze.cell_ref(6, 5) in white_cells: next_Cell = maze.cell_ref(6, 5)
            if maze.cell_ref(3, 3) in white_cells: next_Cell = maze.cell_ref(3, 3)
            if maze.cell_ref(1, 3) in white_cells: next_Cell = maze.cell_ref(1, 3)
            rospy.loginfo(str(self.id) + ' white cell choice')
            return next_Cell
        if len(grey_cells_visited) + len(grey_cells_not_visited) == 0:
            rospy.loginfo(str(self.id) + ' halt')
            return self.CurrentCell(maze)
        if len(grey_cells_not_visited) > 0:
            next_Cell = random.choice(grey_cells_not_visited)
            rospy.loginfo(str(self.id) + ' gray not visited choice')
            return next_Cell
        if len(grey_cells_visited) > 0:
            new_grey_cells = self.sort_cell_list(grey_cells_visited)
            next_Cell = new_grey_cells[0]
            if self.id == "5":
                if maze.cell_ref(5, 4) in grey_cells_visited: next_Cell = maze.cell_ref(5, 4)
            rospy.loginfo(str(self.id) + ' gray visited choice')
            return next_Cell
        rospy.loginfo('error choice')
        return self.CurrentCell(maze)

```

```
    def maze_exploration(self, maze, lock):
        while maze.cell[6][1].color == 0 and not rospy.is_shutdown():
            self.explore(maze, lock)
            time.sleep(1)
        lock.acquire()
        path = find_path(maze.graph, self.CurrentCell(maze), maze.cell[6][1])
        lock.release()
        while not path:
            self.explore(maze, lock)
            lock.acquire()
            path = find_path(maze.graph, self.CurrentCell(maze), maze.cell[6][1])
            lock.release()
        self.follow_path(path[0], maze, lock)
        self.TurnRight()
        self.move_stop(2)
        time.sleep(4)
        self.move_stop()
        self.CurrentCell(maze).tag = 1

    def follow_path(self, path, maze, lock):
        rospy.loginfo(str(self.id) + " path following")
        for i in range(1, len(path)):
            self.next_Cell = path[i]
            while not self.next_Cell.tag:
                time.sleep(.1)
            self.Move(maze, self.next_Cell, lock)

class detector:
    def __init__(self, robots):
        self.marker_info = {
            "cam": None,
            "14": None,
            "10": None,
            "1": None,
            "9": None,
            "5": None
        }
        self.robot_info = {
            "0": None,
            "1": None,
            "2": None,
            "3": None,
            "4": robots[0],
            "5": robots[2],
            "6": robots[3],
            "7": robots[1],
            "8": None,
            "14": None,
            "10": None,
            "11": None, #
            "12": None, #
            "9": None,
```

```
        }

        self.Left_Down_Q_Y = []
        self.Left_Down_Y = 0
        self.Left_Down_Q_Z = []
        self.Left_Down_Z = 0
        self.Left_Up_Q_Y = []
        self.Left_Up_Y = 0
        self.Left_Up_Q_Z = []
        self.Left_Up_Z = 0
        self.Right_Down_Q_Y = []
        self.Right_Down_Y = 0
        self.Right_Down_Q_Z = []
        self.Right_Down_Z = 0
        self.Right_Up_Q_Y = []
        self.Right_Up_Y = 0
        self.Right_Up_Q_Z = []
        self.Right_Up_Z = 0
        self.robot_Q_Y = []
        self.robot_Y = 0
        self.robot_Q_Z = []
        self.robot_Z = 0

        self.Mleft = 90
        self.Mright = 90
        self.Mup = 0
        self.Mdown = 0

        self.InitializeFlag = 0
        self.maze = None
        self.robot_count = 0
        for i in robots:
            if i:
                self.robot_count += 1

    def initial(self, maze):
        while len(self.Right_Down_Q_Y) < 50000 and not rospy.is_shutdown():
            self.Right_Down_Q_Y.append(self.get_position(right_down).y)
            self.Right_Down_Q_Z.append(self.get_position(right_down).z)
            self.Left_Up_Q_Y.append(self.get_position(left_up).y)
            self.Left_Up_Q_Z.append(self.get_position(left_up).z)
            self.Left_Down_Q_Y.append(self.get_position(left_down).y)
            self.Left_Down_Q_Z.append(self.get_position(left_down).z)
            self.robot_Q_Y.append(self.get_position(0).y)
            self.robot_Q_Z.append(self.get_position(0).z)
        self.Right_Down_Y = Average(self.Right_Down_Q_Y) * 100
        self.Right_Down_Z = Average(self.Right_Down_Q_Z) * 100
        self.Left_Down_Y = Average(self.Left_Down_Q_Y) * 100 - self.Right_Down_Y
        self.Left_Down_Z = Average(self.Left_Down_Q_Z) * 100 - self.Right_Down_Z
        self.Left_Up_Y = Average(self.Left_Up_Q_Y) * 100 - self.Right_Down_Y
```

```
        self.Left_Up_Z = Average(self.Left_Up_Q_Z) * 100 - self.Right_Down_Z
        self.robot_Y = Average(self.robot_Q_Y) * 100 - self.Right_Down_Y
        self.robot_Z = Average(self.robot_Q_Z) * 100 - self.Right_Down_Z
        self.Mright = 0
        self.Mleft = SlopeDeg(self.Left_Down_Z, self.Left_Down_Y, self.Left_Up_Z, self.Left_Up_Y)
        self.Mup = 0
        self.Mdown = SlopeDeg(0, 0, self.Left_Down_Z, self.Left_Down_Y)
        rospy.loginfo("RD: y: " + str(self.Right_Down_Y) + " z: " + str(self.Right_Down_Z))
        rospy.loginfo("LU: y: " + str(self.Left_Up_Y) + " z: " + str(self.Left_Up_Z))
        rospy.loginfo("LD: y: " + str(self.Left_Down_Y) + " z: " + str(self.Left_Down_Z))
        rospy.loginfo("R : y: " + str(self.robot_Y) + " z: " + str(self.robot_Z))
        rospy.loginfo( " Mdown: " + str(self.Mdown))
        rospy.loginfo("Mleft: " + str(self.Mleft))
        self.make_maze(maze)
        self.maze = maze
        self.InitializeFlag = 1


    def make_maze(self, maze):
        Row = len(maze.cell)
        Col = len(maze.cell[0])
        maze.cell[Row - 1][Col - 1].xpos = self.Left_Up_Y / (2 * (Col - 1))
        maze.cell[Row - 1][Col - 1].ypos = self.Left_Up_Z / (2 * (Row - 1))
        maze.cell[1][1].xpos = self.Left_Up_Y - self.Left_Up_Y / (2 * (Col - 1))
        maze.cell[1][1].ypos = self.Left_Up_Z - self.Left_Up_Z / (2 * (Row - 1))
        maze.cell[Row - 1][1].xpos = self.Left_Up_Y - self.Left_Up_Y / (2 * (Col - 1))
        maze.cell[Row - 1][1].ypos = self.Left_Up_Z / (2 * (Row - 1))
        maze.cell[1][Col - 1].xpos = self.Left_Up_Y / (2 * (Col - 1))
        maze.cell[1][Col - 1].ypos = self.Left_Up_Z - self.Left_Up_Z / (2 * (Row - 1))
        for r in range(1, Row):
            for c in range(1, Col):
                maze.cell[r][c].xpos = maze.cell[Row - 1][Col - 1].xpos + (Col - 1 - c) * (maze.cell[Row - 1][1].xpos - maze.cell[Row - 1][Col - 1].xpos)  / (Col - 2)
                maze.cell[r][c].ypos = maze.cell[Row - 1][Col - 1].ypos + (Row - 1 - r) * (maze.cell[1][Col - 1].ypos - maze.cell[Row - 1][Col - 1].ypos)  / (Row - 2)
        for i in range(1, Row):
            maze.cell[i][Col - 1].rightWall = 0
            maze.cell[i][0].rightWall = 0
        for i in range(1, Col):
            maze.cell[0][i].downWall = 0
```

```
            maze.cell[Row - 1][i].downWall = 0
        maze.cell[5][10].downWall = 0
        maze.cell[4][10].downWall = 0

    def renew_position_callback(self, data):
        for marker in data.markers:
            self.marker_info[self.get_id(marker)] = marker
            if self.robot_info[self.get_id(marker)] != None:
                self.set_pos(self.robot_info[self.get_id(marker)])
        self.visualize()

    def get_id(self, marker):
        return str(marker.id)

    def get_pose(self, id):
        return self.marker_info[str(id)].pose.pose

    def get_position(self, id):
        return self.marker_info[str(id)].pose.pose.position

    def get_orientation(self, id):
        return self.marker_info[str(id)].pose.pose.orientation

    def set_pos(self, robot):
        marker_data = self.marker_info[robot.id]
        robot.Xpos = marker_data.pose.pose.position.y * 100 - self.InitializeFlag * self.Right_Down_Y
        robot.Ypos = marker_data.pose.pose.position.z * 100 - self.InitializeFlag * self.Right_Down_Z
        robot.Xorient = marker_data.pose.pose.orientation.x
        robot.Yorient = marker_data.pose.pose.orientation.y
        robot.Zorient = marker_data.pose.pose.orientation.z
        robot.Worient = marker_data.pose.pose.orientation.w
        roll, pitch, yaw = euler_from_quaternion([robot.Xorient, robot.Yorient, robot.Zorient, robot.Worient])
        robot.alpha = math.degrees(pitch)
        if yaw < 0:
            yawSign = - 1
        else:
            yawSign = 1
        robot.theta = (robot.alpha - 90) * yawSign
        if -135 < robot.theta < -45:
            robot.direction = 1
        if -45 < robot.theta < 45:
            robot.direction = 0
        if 45 < robot.theta < 135:
            robot.direction = 3
        if robot.theta > 135 or robot.theta < -135:
            robot.direction = 2
        robot.alpha = math.degrees(pitch)
        if self.InitializeFlag == 1:
            pass
```

```python
    def visualize(self):
        w = 40
        thickness = 2
        maze = self.maze
        if maze:
            Row = len(maze.cell)
            Col = len(maze.cell[0])
            img = np.zeros([(Row + self.robot_count + 1) * w, (Col + 3) * w,3], dtype = np.uint8)
            img.fill(255)
            for i in range(Row):
                for j in range(Col):
                    if maze.cell[i][j].downWall == 0:
                        x1 = w * j
                        y1 = w * (i + 1)
                        x2 = w * (j + 1)
                        y2 = w * (i + 1)
                        cv.line(img, (x1, y1), (x2, y2), (255, 0, 0), thickness, lineType = 8)
                    if maze.cell[i][j].rightWall == 0:
                        x1 = w * (j + 1)
                        y1 = w * i
                        x2 = w * (j + 1)
                        y2 = w * (i + 1)
                        cv.line(img, (x1, y1), (x2, y2), (255, 0, 0), thickness, lineType = 8)
                    if maze.cell[i][j].color == 1:
                        start_point = (w * j + thickness, w * i + thickness)
                        end_point =  (w * (j + 1) - thickness, w * (i + 1) - thickness)
                        cv.rectangle(img, start_point, end_point, (160, 160, 160), -1)
                    if maze.cell[i][j].color == 2:
                        start_point = (w * j + thickness, w * i + thickness)
                        end_point =  (w * (j + 1) - thickness, w * (i + 1) - thickness)
                        cv.rectangle(img, start_point, end_point, (0, 0, 0), -1)
                    if i != 0 and j!= 0:
                        cv.putText(img, str(maze.cell[i][j].tag),(int((j + .5) * w), int((i + .5) * w)), cv.FONT_HERSHEY_SIMPLEX, .3, (0, 0, 0), 1, cv.LINE_AA)
            count = 0
            for tag in self.robot_info:
                if self.robot_info[tag]:
                    robot = self.robot_info[tag]
                    color = robot.rgb
                    dirr = robot.direction
                    r = robot.CurrentCell(maze).Row
                    c = robot.CurrentCell(maze).Col
                    center = (w * c + w / 2, w * r + w / 2)
```

```
                    if r:
                        cv.circle(img, center, w / 3, color, thickness)
                        if dirr == 0:
                            rr, cc = -1, 0
                        elif dirr == 1:
                            rr, cc = 0, -1
                        elif dirr == 2:
                            rr, cc = 1, 0
                        else:
                            rr, cc = 0, 1
                        cent = (center[0] + rr * w / 7, center[1] + cc * w / 7)
                        cv.circle(img, cent, w / 8, color, thickness)
                    count += 1
            cv.imshow("hello", img)
            cv.waitKey(3)

def main():
    Row = 11
    Col = 11

    maze = Maze(Row, Col) # define our maze object
    robot0 = Robot(4, 11, (0, 0, 255), 3.5) #tag4
    robot1 = None #Robot(7, 12, (200, 200, 0), 3) #tag7
    robot2 = Robot(5, 13, (0, 200, 200), 3.3) #tag5
    robot3 = Robot(6, 14, (255, 0, 255), 3.5) #tag6
    robots = [robot0, robot1, robot2, robot3]
    camera = detector(robots)
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber('/zed/ar_pose_marker', AlvarMarkers, camera.renew_position_callback)
    if robot0: #tag4
        rospy.Subscriber("/epuck_robot_0/dist_sens", Range, robot0.distance_callback)
        rospy.Subscriber("/epuck_robot_0/proximity1", Range, robot0.prox0_callback)
        rospy.Subscriber("/epuck_robot_0/proximity2", Range, robot0.prox2_callback)
        rospy.Subscriber("/epuck_robot_0/proximity5", Range, robot0.prox5_callback)
        rospy.Subscriber("/epuck_robot_0/proximity6", Range, robot0.prox7_callback)
        velocity_publisher0 = rospy.Publisher('/epuck_robot_0/mobile_base/cmd_vel', Twist, queue_size=10)
        robot0.vel_assign(velocity_publisher0)
    if robot1: #tag7
        rospy.Subscriber("/epuck_robot_1/dist_sens", Range, robot1.distance_callback)
        rospy.Subscriber("/epuck_robot_1/proximity1", Range, robot1.prox0_callback)
        rospy.Subscriber("/epuck_robot_1/proximity2", Range, robot1.prox2_callback)
        rospy.Subscriber("/epuck_robot_1/proxiRobotmity5", Range, robot1.prox5_callback)
```

```
        rospy.Subscriber("/epuck_robot_1/proximity6", Range, robot1.prox7_callback)
        velocity_publisher1 = rospy.Publisher('/epuck_robot_1/mobile_base/cmd_vel', Twist, queue_size=10)
        robot1.vel_assign(velocity_publisher1)
    if robot2: #tag5
        rospy.Subscriber("/epuck_robot_2/dist_sens", Range, robot2.distance_callback)
        rospy.Subscriber("/epuck_robot_2/proximity1", Range, robot2.prox0_callback)
        rospy.Subscriber("/epuck_robot_2/proximity2", Range, robot2.prox2_callback)
        rospy.Subscriber("/epuck_robot_2/proximity5", Range, robot2.prox5_callback)
        rospy.Subscriber("/epuck_robot_2/proximity6", Range, robot2.prox7_callback)
        velocity_publisher2 = rospy.Publisher('/epuck_robot_2/mobile_base/cmd_vel', Twist, queue_size=10)
        robot2.vel_assign(velocity_publisher2)
    if robot3: #tag6
        rospy.Subscriber("/epuck_robot_3/dist_sens", Range, robot3.distance_callback)
        rospy.Subscriber("/epuck_robot_3/proximity1", Range, robot3.prox0_callback)
        rospy.Subscriber("/epuck_robot_3/proximity2", Range, robot3.prox2_callback)
        rospy.Subscriber("/epuck_robot_3/proximity5", Range, robot3.prox5_callback)
        rospy.Subscriber("/epuck_robot_3/proximity6", Range, robot3.prox7_callback)
        velocity_publisher3 = rospy.Publisher('/epuck_robot_3/mobile_base/cmd_vel', Twist, queue_size=10)
        robot3.vel_assign(velocity_publisher3)

    # allow  the robot for a second to initialize the sensors:
    time.sleep(.5)
    camera.initial(maze)
    #maze.present()
    time.sleep(1)
    #robot2.TurnRight()
    # position initialization in the first
    comment = 1
    if comment == 1:
        if robot0:
            robot0.initializePos(maze, 5, 10)
        if robot1:
            robot1.initializePos(maze, 8, 5)
        if robot2:
            robot2.initializePos(maze, 2, 6)
        if robot3:
            robot3.initializePos(maze, 3, 4)
        lock = threading.Lock()
        t0 = threading.Thread(target=robot0.maze_exploration, args=(maze, lock) )
```

```
        #t1 = threading.Thread(target=robot1.maze_exploration, args=(maze, lock) )
        t2 = threading.Thread(target=robot2.maze_exploration, args=(maze, lock) )
        t3 = threading.Thread(target=robot3.maze_exploration, args=(maze, lock) )
        time.sleep(1)
        t0.start()
        time.sleep(1)
        #t1.start()
        t2.start()
        time.sleep(1)
        t3.start()
        t0.join()
        #t1.join()
        t2.join()
        t3.join()


if __name__ == '__main__':

    try:
    #Testing our function
        main()
    except rospy.ROSInterruptException: pass
```