FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Simulation of Centralized Algorithms for Multi-Agent Path Finding on Real Robots
Student:	Ján Chudý
Supervisor:	doc. RNDr. Pavel Surynek, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2020/21

Instructions

The task is to simulate selected multi-agent pathfinding (MAPF) algorithms on real robots. It is expected that a group of faculty's OZOBOTs will be used for simulation. The important challenge is how to adapt existing algorithms for the laboratory case of the multi-agent pathfinding problems for real robotic hardware as we expect that the capabilities of OZOBOTs are limited. A direction we would like to investigate is an on-line drawing of the environment and the planned path on the screen so that robots can use their build-in localization functions when moving on the surface of the screen. The student should complete the following tasks:

1. Study relevant literature on multi-agent pathfinding with a special focus on pathfinding in a physical environment.
2. Explore the hardware and software capabilities of OZOBOTs.
3. Find a technique on how to implement the selected algorithm for a group of OZOBOTs.
4. Perform experimental evaluation and simulations.

References

[1] Guni Sharon, Roni Stern, Meir Goldenberg, Ariel Felner: The increasing cost tree search for optimal multi-agent pathfinding. Artif. Intell. 195: 470-495 (2013)

[2] Jiaoyang Li, Pavel Surynek, Ariel Felner, Hang Ma, T. K. Satish Kumar, Sven Koenig: Multi-Agent Path Finding for Large Agents. SOCS 2019: 186-187

[3] Anton Andreychuk, Konstantin S. Yakovlev, Dor Atzmon, Roni Stern: Multi-Agent Pathfinding with Continuous Time. IJCAI 2019: 39-45

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 7, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Simulation of Centralized Algorithms for Multi-Agent Path Finding on Real Robots

Ján Chudý

Department of Theoretical Computer Science
Supervisor: doc. RNDr. Pavel Surynek, Ph.D.

May 26, 2020

Acknowledgements

Firstly, I would like to thank my supervisor for his invaluable insight into the field of multi-agent pathfinding and active guidance. Whenever I needed help, he gave me advice or expressed his opinion based on experience. My biggest thanks go to my parents, who always led me to education and discovery. Without their support and sacrifices, I would not be able to study away from home and reach this milestone. Last but not least, thanks to my girlfriend, who was very supportive and tolerated the buzzing sound of the robots with which I experimented.

Declaration

In Prague on May 26, 2020 ....

Citation of this thesis

Chudý, Ján. *Simulation of Centralized Algorithms for Multi-Agent Path Finding on Real Robots*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstract

The simulation of multi-agent pathfinding solutions is essential for research but also in educational demonstrations. Most of the time, the simulation is only displayed on a screen without the use of robotic agents. If robots are used, they get a sequence of commands they need to execute, or they receive the commands gradually, to follow their planned paths correctly. This work proposes a novel approach to simulation of centralized multi-agent pathfinding algorithms on physical agents called *ESO-Nav*. In this approach, the agents are not part of the planning process, nor do they have any information about their paths. The agents have a simple predetermined behavior in an environment and navigate in it based on the environment outputs. A working prototype of a simulator that utilizes this novel approach was implemented for a group of Ozobot Evo robots.

Keywords multi-agent pathfinding, MAPF, simulation, navigation by environment outputs, centralized algorithms, ozobot, real robot, grid maps

Abstrakt

Simulace řešení multi-agentího hledání cest je nezbytná pro výzkum, ale také pro demonstrace v akademickém prostředí. Většinou se simulace pouze zobrazuje na obrazovce bez použití robotických agentů. Používají-li se roboty, obdrží posloupnost příkazů, které potřebují provést, nebo příkazy obdrží postupně, aby správně sledovaly své naplánované cesty. Tato práce navrhuje nový přístup k simulaci centralizovaných multi-agentných algoritmů pro hledání cest na fyzických agentech s názvem *ESO-Nav*. V tomhle přístupu agenti nejsou součástí plánovacího procesu, ani nemají o svých cestách žádné informace. Agenti mají jednoduché předdefinované chování v prostředí, v kterém navigují na základě jeho podnetů. Pro skupinu robotů Ozobot Evo byl implementován funkční prototyp simulátoru, který využívá tento nový přístup.

Klíčová slova multi-agentní hledání cest, MAPF, simulace, navigace pomocí výstupů prostředí, centralizované algoritmy, ozobot, skutečný robot, mřížkové mapy

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Multi-agent pathfinding problem is a task of finding paths for multiple agents from their initial positions to their goal positions while ensuring that the agents do not collide at any point in their path execution. Due to its real-world applications, multi-agent pathfinding has been deeply researched topic over recent years. New variations of this problem, as well as approaches to solving them, emerged, and new solving algorithms have been designed.

However, most of these solvers are tested and benchmarked in the virtual environment of the computer's memory, where there are no moving parts involved, nor any laws of physics affecting the agents. Even though this type of simulation is great for theoretical research and agile solver benchmarking, it might not be the most exciting for students or someone interested in understanding this topic. Moreover, it is far from using physical robots, which usually requires a fundamentally different approach to solving multi-agent pathfinding problems.

This thesis aims to test a new approach to simulating the existing multi-agent pathfinding algorithms using small robots. A group of Ozobots was chosen for this task. If this approach proves capable of simulating multi-agent pathfinding sufficiently, it could be utilized by different groups of people. It will mostly help educators on their mission to showcase multi-agent pathfinding more entertainingly, but can also be beneficial for researchers. They will be able to simulate their findings and theoretical approaches in the physical world, to improve their methods before utilizing the expensive machinery for which their solution is intended. Hopefully, this approach will also unlock new research paths that are yet to be explored.

Aim of the thesis

The main objective of this thesis is to create a prototype of a novel simulation approach that is intended for simulation of centralized multi-agent pathfinding algorithms on physical robots like Ozobots. This objective can be broken down to several tasks like so:

- Explore Ozobots and their capabilities.
- Explain the selected approach and compare it to the already existing work.
- Create a simulation prototype.

It is also important to mention that this thesis does not aim to develop a new multi-agent pathfinding algorithm, nor modify existing algorithms to work with real robots.

Expected outcome

The hypothesis is that the novel simulation approach proposed in this thesis will prove to be versatile enough to simulate various types of multi-agent pathfinding problems.

Structure of the thesis

The thesis firstly provides the reader with some theoretical background in the first chapter. The second chapter explains Ozobots and their capabilities, and how their behavior can be altered via their supported coding techniques. It is a selection of valuable information from the Ozobot documentation, but also a summary of crucial findings gathered from testing and playing with Ozobots. The third chapter describes the chosen approach and compares it to previous similar work. The chapter also anticipates some problems that might arise with the selected approach. The fourth chapter is dedicated to the realization of the simulation. All modules, as well as their interactions, are outlined, and the simulation is iteratively refined. At the end of the chapter, the final version of the simulator is presented. The fifth chapter then performs and evaluates experiments on this simulation prototype. Lastly, the conclusion sums up the work and findings of the thesis, evaluates the completion of goals, and proposes future steps or research.

Chapter 1

Theoretical background

In this chapter, the reader is provided with the required theoretical background concerning the task at hand. Firstly, a selection of terms from graph theory is explained, and then the multi-agent pathfinding (MAPF) problem can be formalized. Later sections compare different approaches to defining and solving of MAPF problems, discuss complications that come with the real-world application of MAPF, and present two state-of-the-art MAPF solvers.

1.1 Graph theory

Since MAPF problems are usually modeled on graphs, and some of the solving approaches extensively use the tools and methods of graph theory, it would be helpful that the reader was familiar with some of the terms that will be used in this work. Because this thesis does not exploit any MAPF approach or algorithm to great detail, it will be satisfactory to familiarize the reader with only a handful of basic terms. Additional concepts can be found in [1], from where most of the following definitions are taken.

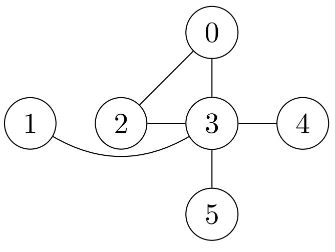
1.1.1 Basic terminology

1.1.1.1 Graph

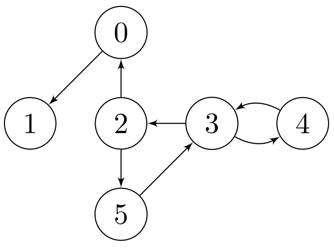
A *graph*[1] is the fundamental concept of graph theory. In the most common form, it is defined as an ordered pair $G = (V, E)$ of sets such that $E \subseteq [V]^2$, where:

- V is a non-empty finite set of *vertices*[2] of the graph G.
- E is a set of *edges* of the graph G.

[1]Usually meant as an undirected graph, but the word "undirected" is omitted.
[2]Sometimes also referred to as nodes.



(a) Undirected graph G



(b) Directed graph H

Figure 1.1: Example of undirected and directed graphs

As denoted by $E \subseteq [V]^2$, E is a subset of all 2-element subsets of V. In other words, edges are unordered pairs of vertices and an edge $\{x, y\}$ is usually written as xy. Two vertices are called *adjacent* or *neighboring* if they are connected with an edge. The vertex set of a graph G is referred to as $V(G)$ and its edge set as $E(G)$.

1.1.1.2 Directed graph

A *directed graph* is very similar to the undirected one, except for edges being directed and usually represented as ordered pairs of vertices. Formally, the directed graph is an ordered pair $G = (V, E)$ together with two maps, $init : E \to V$ and $ter : E \to V$, assigning to every edge an initial vertex $init(e)$ and a terminal vertex $ter(e)$.

1.1.1.3 Graph visualization

Graphs are usually pictured by drawing a dot for each vertex and connecting pairs of these dots with a line[3] if the corresponding vertices form an edge. An example of undirected graph

$$G = (\{0, \ldots, 5\}, \{\{0, 2\}, \{0, 3\}, \{1, 3\}, \{2, 3\}, \{3, 4\}, \{3, 5\}\})$$

can be found in Figure 1.1a and an example of directed graph

$$H = (\{0, \ldots, 5\}, \{(0, 1), (2, 0), (2, 5), (3, 2), (3, 4), (4, 3), (5, 3)\})$$

in Figure 1.1b. Often, there are many ways of picturing the same graph, and how the dots and lines are drawn is considered irrelevant. The most important information is which pairs of vertices form an edge and which do not.

[3]For directed graphs, an arrow is used.

1.1.1.4 Subgraph

Let there be two graphs $G = (V, E)$ and $G' = (V', E')$. If $V' \subseteq V$ and $E' \subseteq E$, then G' is a *subgraph*[4] of G, written as $G' \subseteq G$.

1.1.2 Graph types

There are different types of graphs, each having their specific properties and uses. For this thesis, however, the most relevant graphs are a path and a two-dimensional grid graph.

1.1.2.1 Path graph

A *path* is a non-empty graph $P = (V, E)$ of the form

$$V = \{x_0, x_1, \dots, x_k\} \qquad E = \{x_0x_1, x_1x_2, \dots, x_{k-1}x_k\},$$

where the x_i are all distinct. A path is often referred to by the sequence of its vertices, written as $P = x_0x_1 \dots x_k$, and called as a path from x_0 to x_k. The number of edges of a path is its length, and the path of length k is denoted by P^k.

1.1.2.2 Two-dimensional grid graph

An n-dimensional p_0–p_1–…–p_n-grid graph can be defined as a product of n paths $P^{p_0}, P^{p_1}, \dots, P^{p_n}$. The definition of a 2-dimensional grid graph [2] needs to be slightly modified for more convenient use in this thesis. This modification will prove useful later when working with grid size and vertex positions.

A *two-dimensional grid graph*[5] $G_{m,n}$ is a product of two paths P^m and P^n, written as $G_{m,n} = P^m \square P^n$. Note that in this text, the grid graph $G_{m,n}$ will be referred to as $x \times y$ grid, where $x = m + 1$ is the width of the grid, and $y = n + 1$ is the height of the grid. A 4×3 grid is pictured in Figure 1.2. When referencing a specific vertex, notion (x, y) will be used, where x is the column, and y is the row of the grid. Both coordinates start at 0. For example, the index of vertex $(0, 0)$ from Figure 1.2 is 0, and vertex 6 is at position $(2, 1)$.

1.1.3 Searching in graphs

From the algorithmic point of view, searching in graphs is a well-researched problem with countless applications in computer science and the real world. There are numerous graph search algorithms, from simple graph traversals algorithms like Depth-first search (DFS) and Breadth-first search (BFS), to more sophisticated ones like the Bellman-Ford algorithm or Dijkstra's algorithm. All

[4] And G is a *supergraph* of G'.

[5] From now on, only "grid graph" will be used.

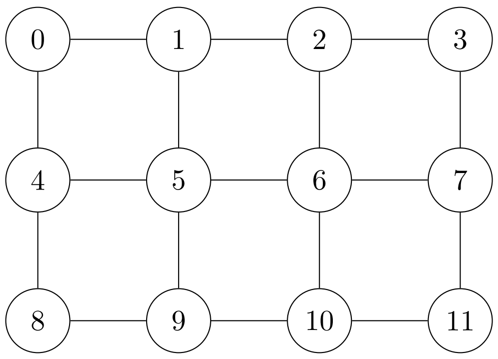


Figure 1.2: Grid graph $G_{3,2}$

of these algorithms can be found in [3], and all can be used for finding a path between two vertices in a graph. From the set of graph search algorithms, A* algorithm [4] and its modern variations are extensively used for solving pathfinding problems.

1.1.3.1 Pathfinding

A *pathfinding* problem [5, 6, 7] is closely related to the *shortest path* problem, which is defined in several variants in [3]. It is a problem of finding a path between two points, often vertices in a graph, such that the distance or cost[6] is minimized. This problem can represent a situation where an agent wants to move from its current position to its goal destination as efficiently as possible. The *agent* can be a person, a robot, or an entity in a video game. Pathfinding problems are being solved continuously all over the world, for example, in online maps.

1.2 Multi-agent pathfinding

Multi-agent pathfinding [8, 9, 10] is a generalization of the pathfinding problem from Section 1.1.3.1 for multiple agents with distinct positions and destinations. It can be inviting to divide such a problem into a set of simple single-agent pathfinding problems, solve each of them using one of the graph searching algorithms mentioned in Section 1.1.3, and execute the planned paths simultaneously. However, this approach does not take into account other agents and their positions in time, resulting in unwanted collisions. Therefore, solving MAPF problems requires different techniques and modified algorithms.

At the beginning of this section, a MAPF problem is formalized. Then some of the variations and state-of-the-art approaches are compared, and

[6]In graphs where the edges are weighted.

lastly, some MAPF complications that arise from the real-world applications are mentioned.

1.2.1 Problem formalization

In [11] and later in [12], Surynek provides a formal definition of two motion problems on a graph. First, he formalizes the problem of *pebble motion on a graph* [13, 14, 15], which is also known as the sliding box puzzle, and its best-known variant is the 15-puzzle. He then provides a definition of the *multi-robot path planning*[7] problem [16, 17], representing a relaxed variant of the pebble motion problem. Both problems are defined within an abstraction of environments being modeled as undirected graphs. Each vertex of these graphs is occupied by at most one agent, and agents can move between vertices along the undirected edges. The definition of the multi-robot path planning problem can be viewed as a formal definition of classical MAPF.

Definition (problem of multi-robot path planning [11]). *Let $G = (V, E)$ be an undirected graph. Next, let $R = \{\overline{r}_1, \overline{r}_2, \dots, \overline{r}_v\}$ where $v < |V|$ be a set of robots. The graph models an environment in which the robots are moving. The* initial configuration *of the robots is defined by a uniquely invertible function $S_R^0 : R \to V$ (that is, $S_R^0(r) \neq S_R^0(s)$ for every $r, s \in R$ with $r \neq s$). The* goal configuration *of the robots is defined by another uniquely invertible function $S_R^+ : R \to V$ (that is, $S_R^+(r) \neq S_R^+(s)$ for every $r, s \in R$ with $r \neq s$). A problem of multi-robot path planning is the task to find a number μ and a sequence $S_R = [S_R^0, S_R^1, \dots, S_R^\mu]$ where $S_R^k : R \to V$ is a uniquely invertible function for every $k = 1, 2, \dots, \mu$. The following conditions must hold for the sequence S_R:*

(i) *$S_R^\mu = S_R^+$; that is, all the robots reach their destination vertices.*

(ii) *Either $S_R^k(r) = S_R^{k+1}(r)$ or $\{S_R^k(r), S_R^{k+1}(r)\} \in E$ for every $r \in R$ and $k = 1, 2, \dots, \mu - 1$; that is, a robot can either* wait *in a vertex or* move *to the neighboring vertex at each time step.*

(iii) *If $S_R^k(r) \neq S_R^{k+1}(r)$ (that is, the robot r moves between time steps k and $k+1$) and $S_R^k(s) \neq S_R^{k+1}(r) \forall s \in R$ such that $s \neq r$ (that is, no other robot s occupies the target vertex at time step k), then the move of r at the time step k is called to be* allowed *(that is, the robot r moves into an unoccupied neighbouring vertex - a leading robot). If $S_R^k(r) \neq S_R^{k+1}(r)$ and there is $s \in R$ such that $s \neq r \wedge S_R^k(s) = S_R^{k+1}(r) \wedge S_R^k(s) \neq S_R^{k+1}(s)$ (that is, the robot r moves into a vertex that is being left by the robot s) and the move of s at time step k is allowed, then the move of r at the time step k is also allowed. All the moves of robots at all time steps must*

[7] In literature, the same concept can also be referred to as *cooperative pathfinding* (CPF) or MAPF, as it is usually used today.

be allowed. Analogically, this condition, together with the requirement on unique invertibility of functions forming S_R, implies that no two robots can enter the same target vertex at the same time step.

The instance of the problem of multi-robot path planning is formally a quadruple $\Sigma = (G, R, S_R^0, S_R^+)$. The solution of the problem Σ will be sometimes denoted as $S_R(\Sigma) = [S_R^0, S_R^1, \ldots, S_R^\mu]$.

If the multi-robot path planning problem is viewed as a MAPF problem, the robots are the agents[8]. In Figure 1.3, a simple MAPF problem instance is depicted. The number μ in the definition is called the *makespan* [11] of the solution. Makespan is the number of time steps required for all agents to reach their goal destinations, and it is one of the most common objective functions that is used to evaluate MAPF solutions. Another conventional objective function is the *sum of costs* [18, 19], which is the sum of time steps required by each agent to reach its target.

At each time step of the solution, every agent has two possible actions from which it can choose. The agent can either *wait* in its current vertex or *move* to one of the neighboring vertices if the move is allowed. In classical MAPF, these are the only two actions that agents can perform. The solution of a MAPF problem can also be represented as a sequence of moves for each agent. These sequences of actions lead the agents through the graph from their initial configuration to their goal configuration. Note that the definition does not prohibit agents from staying in the same vertex for several time steps, nor to visit one vertex multiple times. Therefore, an agent's path is not necessarily a path subgraph in the underlying graph that models the environment.

The problem of pebble motion on a graph is almost the same as the multi-robot path planning problem. It has one additional constraint, which is that agents can enter only currently unoccupied vertices. In other words, moving into a vertex, from which another agent is leaving, is not considered an allowed move. This constraint can create delays in the plans of the agents if they follow the same path. A valid solution for an instance of the pebble motion problem would also be a correct solution for the multi-robot path planning problem on the same graph and with the same agent configurations. Although this solution could have a larger makespan and thus be less efficient, it is considered to be more *robust* [20].

A solution of a MAPF problem is *valid* if all the moves in every time step are allowed and if no collision occurs. Different types of collisions can be defined for different variations of MAPF problems. In this case, a *collision* would occur if two agents try to enter the same vertex[9], which is prevented by definition. Another type of collision occurs when two agents attempt to

[8]For the problem of pebble motion on a graph, agents would be pebbles.

[9]This is referred to as a *vertex collision*.

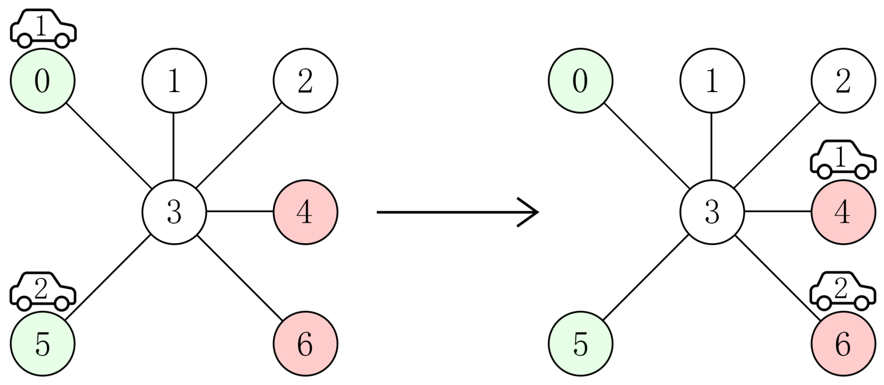


Figure 1.3: MAPF problem instance with two agents (cars), moving from their initial positions (left) to their goal positions (right)

use the same graph edge at the same time step[10], usually because they plan to swap positions.

1.2.2 Variations

All the variations of MAPF problems and approaches to solving these problems can often be divided into different categories. Each developed solver or set of algorithms can then choose to solve a specific problem, which can be defined with a subset of these categories. This section presents a few of these categorizations of MAPF.

1.2.2.1 Optimal and suboptimal approach

The apparent goal of MAPF solvers would be to minimize the objective function and thus find the best possible solution for a given problem. Such a solution that is valid, and there is no other solution of lower cost, is called an *optimal solution*. Many of the state-of-the-art MAPF solvers are *optimal*, meaning that they yield optimal solutions to given MAPF problems. For example, Operator Decomposition and Independent Subproblems [19], Increasing Cost Tree Search (ICTS) [21], Conflict-based Search (CBS) [22], and boolean satisfiability problem (SAT) based MAPF solvers like solvers from [23] and MDD-SAT [24], are all optimal solvers. However, finding an optimal solution to this type of problem is known to be *NP-hard* [15, 25, 26], so these optimal solvers are not very scalable. With the increasing number of agents and large environments, these solvers usually take too long or fail to find a solution.

However, often it is not required to find an optimal solution, but a near-optimal solution that can be found quickly is needed. It is possible to sacrifice some of the solution quality in exchange for fast or efficient computation.

[10]Usually referred to as an *edge collision*.

Therefore, several *suboptimal solvers* that yield *suboptimal solutions* to MAPF problems are available. These solvers are usually relaxations or other modifications of existing optimal solvers, like suboptimal CBS [27] and suboptimal SAT solver [28]. Sometimes, it is needed to guarantee some quality of a suboptimal solution, or the user wants to experiment with the trade-off of solution quality and computational load of the solver. *Bounded suboptimal solvers* can take a parameter that represents the guaranteed quality of the solution compared to the cost of an optimal solution.

1.2.2.2 Centralized and decentralized setting

MAPF solvers can run in two distinct settings: centralized and decentralized. In the *centralized* setting, the solver gets an instance of a MAPF problem and performs a global search of the state space to find a solution from the initial state to the goal state. The agents, if they were some entities that were then to execute the found solution, would each get their pre-planned path from this central solver. Performing a centralized search of a state-space can be computationally exhaustive for large instances of a MAPF problem. There are many MAPF algorithms with this centralized setting, and they can be divided into three categories: search-based, reduction-based, and rule-based. In the *search-based* category are algorithms like CBS [22] and its improved variants [29], ICTS [21], or other algorithms based on A*. *Reduction-based* solvers reduce the MAPF problem to another known problem that already has a high-quality solver. These solvers utilize reduction to SAT problems [23], answer set programming [30], or integer linear programming [31]. *Rule-based* solvers work with specific agent movement rules, and examples are the algorithms Push-and-Swap [32], BIBOX [33], and Push-and-Rotate [34].

On the other hand, the *decentralized* solvers decompose the problem into a set of smaller searches, which provides better scalability for larger instances of MAPF problem. However, these solvers usually do not guarantee to find a solution, and the plans can be far from optimal. Examples of decentralized algorithms would be Local Repair A*, Cooperative A*, Windowed Hierarchical Cooperative A* [9], Flow Annotation Replanning [10], and MAPP algorithm [35].

1.2.2.3 Discrete and continuous approach

Almost all of the previous MAPF research and proposed solvers were built on top of several assumptions about time. First, time is not continuous, but rather *discretized* into time steps. Second, all actions that agents perform take the same amount of time to execute, precisely one time step. Moreover, a significant portion of the research was done on simple grid graphs, and agents are usually entities of the same shape and size that fit into one graph vertex. In [36], the authors propose a Continuous-time Conflict-based Search (CCBS) algorithm that supports *continuous time* and agent actions that are of different durations. Agents even can have different speeds. The collision detection is geometry-aware so that the algorithm can handle agents of different sizes and shapes. The algorithm is optimal and complete.

1.2.2.4 Other approaches

Some other approaches have also been researched over the last years. Opposed to the *cooperative pathfinding* (CPF) [9, 37], which is the same concept as the classical MAPF, where agents aim to fulfill one global goal[11] as effectively as possible, there is *adversarial cooperative pathfinding*[12] (ACPF) [38, 39]. In ACPF, agents are divided into a finite number of teams that alter in turns between time steps. The goal of ACPF is to find a winning solution for one selected team of agents that reacts to moves of other, adversarial, groups of agents. Additionally, ACPF can follow different tactics like offense or defense. The adversarial approach to MAPF provided additional opportunities for future research.

Another interesting variation of MAPF is *multi-agent evacuation* (MAE), or evacuation planning. The goal in MAE is to move agents from a danger zone into a safe zone. Agents usually do not have their specified goal positions, meaning that they can finish at any location in the safe area, but they should not prevent other agents from entering the zone. One of the recently published MAE algorithms is Local Cooperative Multi-agent Evacuation [40].

1.2.3 Real-world complications

MAPF has many real-world applications in robotics [41], warehousing [42], transport [43], or video games [44, 45], and each of the possible applications most likely has its specific solution. However, most of the MAPF research is done in theory and works within a specified abstraction. Unfortunately, between theoretical study and real-world MAPF problems, there is a gap that often needs to be overcome in order to apply existing solutions to these situations. This gap is mainly a product of the *abstraction* used in MAPF, which usually does not correspond to reality.

The environment is usually modeled as an unordered graph or a tiled grid. Even though this can often be sufficient even for a real-world scenario, the reality is not generally that easy to model. For example, maps in some video games can benefit from not being grid-based but instead use polygons [44].

There are several *assumptions for the agents* in MAPF. Point agents are usually used, where they are of the same size and shape and occupy a single point in the environment representation. However, physical agents are geometrical. They have their specific form, and they can collide in many differ-

[11]That is usually reaching their specified destinations in the environment.

[12]Adversarial cooperative pathfinding can also be referred to as adversarial pathfinding or adversarial MAPF.

ent ways. Different agent representation and collision detection are therefore needed if pursuing a more realistic model. The CCBS algorithm mentioned in Section 1.2.2.3 tries to combat this with supporting agents of different shapes and using geometry-aware collision detection. Also, a study [46] solely around this concept of geometrical agents was done. In that work, authors refer to such agents as *large agents*, and they formalize and study a *MAPF for large agents*, then propose a new algorithm for this problem.

The *time assumptions* mentioned in Section 1.2.2.3 are also a problem. Actions of mechanical agents do not usually take the same time, and they do not merely snap between positions in the environment instantaneously. The agents require continuous movement to reposition. Continuous MAPF can adequately accommodate this problem, but also other approaches might sufficiently simulate the fluid transfer of agents. For example, solutions that are finely discretized based on the agent movement and wait times, or that use weighted graph edges as a representation of different time durations may imitate continuity. Another assumption of abstract models is that all agent movements are synchronous, but in reality, there can be many factors that might introduce *desynchronization* into the plan execution. One of these factors is the mentioned variety of move durations. Because every agent is executing a different sequence of actions, their movements are desynchronized quickly. Fortunately, this factor can be mitigated by using a suitable abstract solver, but many factors cannot be anticipated. Some kind of monitoring of the execution is needed, when weather, terrain, or other unexpected circumstances might cause desynchronization.

It is essential to mention that a collision of agents in the real world could damage them or have other real-world consequences. Moreover, the ability of an agent to follow the planned path successfully can be dependant on the environment or its hardware. The plans should, therefore, account for unexpected mistakes and delays to prevent unanticipated collisions. For this, *k-robustness* [20] can be introduced to the plan. A k-robust plan, besides the classical MAPF plan, requires that after an agent leaves a position, no other agent can enter it for the next k time steps. If agents were to move in a train-like formation, there would be empty spaces between them.

In classical MAPF, the agents can move in any direction for every time step of the plan. Even though this might be true for some agents like drones, this is not true for most wheeled agents moving on the ground. When an agent wants to change the direction of movement, it is required to rotate what takes some time and adds to the plan desynchronization. These rotation movements can also be incorporated in the MAPF abstraction, as proposed in [47]. The authors suggest splitting *position vertices* into *directional vertices*, which represent the direction the agent is facing. Edges between these new vertices represent rotation actions and original edges movements between the original vertices. This change also requires a modification in the solver, namely in conflict detection. The study's primary purpose was to test the behavior

Algorithm 1: High-level search of CBS algorithm

```
Input: MAPF problem instance $\Sigma$
1  $R.constraints \leftarrow \varnothing$
2  $R.solution \leftarrow$ paths from low-level search
3  $R.cost \leftarrow \text{cost}(R.solution)$
4  insert $R$ into $OPEN$
5  while $OPEN \neq \varnothing$ do
6      $N \leftarrow$ node from $OPEN$ with lowest cost
7      validate $N.solution$ until a conflict is found
8      if $N.solution$ has no conflicts then
9          return $N.solution$ // it is a valid solution
10     $C \leftarrow$ first conflict $(a_i, a_j, v, t)$ found in $N.solution$
11     foreach $a$ in $\{a_i, a_j\}$ do
12         $N' \leftarrow$ new constraint tree node
13         $N'.constraints \leftarrow N.constraints \cup \{(a, v, t)\}$
14         $N'.solution \leftarrow N.solution$
15         update $N'.solution$ for agent $a$ with low-level search
16         if $N'.solution$ was found then
17             $N'.cost \leftarrow \text{score}(N'.solution)$
18             insert $N'$ into $OPEN$
```

of several defined MAPF models when executed on physical robots. The experiments concluded that classical MAPF plans are not suitable for such use, and some of the other proposed models yielded better results than the classical one. Since this study used Ozobots to simulate the execution of the plans, it is further discussed in Section 3.1.

1.2.4 Optimal centralized algorithms in discrete space

In its practical part, this work is utilizing only discrete centralized MAPF algorithms that are optimal. In this section, two state-of-the-art representative solutions are presented in a high-level overview, namely the principles of CBS and SAT-based solver.

1.2.4.1 CBS algorithm

The Conflict-based Search [22] is a two-level optimal MAPF algorithm that decomposes the MAPF problem into several constrained single-agent pathfinding problems that are easier to solve. The algorithm is composed of two searches: high-level search and low-level search.

The *high-level search* is performed on a binary *constraint tree* that is expanding during the search until a valid solution is found. Each node of the constraint tree holds a set of constraints, a found solution, and a cost. All constraints are tuples (a_i, v, t), meaning that the agent a_i cannot be at the vertex v at time step t. Each node inherits the set of constraints from its parent and adds only one extra constraint for a specific agent. The solution in a node is a set of paths for all agents found by the low-level search. Each of the paths is restricted by the constraints for the given agent. The cost of the node is the cost of the found solution. In Algorithm 1, the pseudocode of the high-level search of CBS is shown. At the beginning of the search, the root of the constraint tree is initialized. The set of constraints in the root node is empty, and the solution is a set of shortest paths for each agent. The tree is searched in a best-first manner, meaning that in each iteration, the lowest-cost node from the open nodes is processed and expanded. A solution of the currently processed node is checked for conflicts between agents in their paths. If there is no conflict, the solution is valid, and these paths are returned. If a conflict is found, two new constraints are added to the constraint tree for this conflict. The conflict is a tuple (a_i, a_j, v, t), meaning that both agents a_i and a_j occupy vertex v at time step t. The node is split into two child nodes, each introducing a new constraint for one of the conflicted agents and hold an updated solution. For example, for agent a_i, a new constraint (a_i, v, t) is added. Then the path of the agent a_i from the previous solution is replanned with the low-level search that uses the new set of constraints. If the low-level search founds a valid path for this agent, the node is added to the set of open nodes.

The *low-level search* performs a simple single-agent pathfinding search for a given agent while making sure the solution does not break any constraints concerning the agent. In this low-level of CBS, any optimal single-agent pathfinding algorithm can be used. The constraints that the algorithm needs to handle ensure that none of the previously-detected conflicts in the high-level search is repeated in the new path.

1.2.4.2 SAT-based solver

The principle of reduction-based solvers is different from other approaches to solving MAPF problems. The *SAT-based solver* [23, 24] transforms an instance of a MAPF problem into a *propositional formula.* This formula is satisfiable only if the MAPF problem is solvable and can be consulted with an already existing state-of-the-art SAT solver. If the formula can be satisfied, and the solver finds a *satisfying assignment*, the solution of the MAPF problem can be reconstructed from this assignment. Therefore, the main challenge is the encoding of the MAPF problem into a propositional formula. The benefit of this approach is that if there is any progress in solving SAT problems, it can increase the efficiency of solving MAPF problems.

Algorithm 2: SAT solving framework for MAPF problems

```
Input: MAPF problem instance $\Sigma$
1 solution ← set of shortest paths for all agents
2 $\mu$ ← makespan of solution
3 while true do
4     $\mathcal{F}(\mu) \leftarrow$ encode($\Sigma$, $\mu$)
5     assignment ← SAT-solver($\mathcal{F}(\mu)$)
6     if assignment $\neq$ UNSAT then
7         solution ← extract MAPF solution from assignment
8         return solution
9     $\mu \leftarrow \mu + 1$
```

The primary concept, allowing the encoding of a MAPF problem into a propositional formula, is a *time expansion graph* (TEG) of the original graph from the problem instance. The TEG is created by duplicating all vertices from the original graph for all time steps from 0 to a given bound μ[13]. This can be imagined as a layered graph where each layer of vertices represents an individual time step. Then all possible actions are represented by directed edges between consecutive layers. An edge between corresponding vertices in two layers represent a *wait action*, while an edge between neighboring vertices in the layers represent a *move action*. This TEG is created for each agent. In the encoding, a *propositional variable* is introduced for each of the vertices of these new graphs. The variable is *true* if the agent occupies the vertex at the time step that the variable represents. Similarly, each directed edge is encoded, and other constraints are added so that the found satisfying assignments correspond to a valid MAPF solution. However, because of the bound μ of the TEG, the encoded formula can only be satisfied if a solution that takes up to μ time steps exists. Note that this corresponds to the makespan of the solution.

The pseudocode of an optimal SAT-based solving framework is in Algorithm 2. The optimal solver operates in a way that repeatedly asks the SAT solver, if a solution of a certain makespan, which is incremented in a loop, exists. When it finally finds a solution, this solution has the lowest possible makespan and therefore is optimal. This process could start from makespan $\mu = 0$, but that would be inefficient. First, a lower bound makespan of the MAPF solution is determined, from which the incremental consulting begins. This can be done by finding the optimal single-agent pathfinding solutions for each agent, which can be done very efficiently. The makespan of the MAPF solution cannot be lower than the longest path of these agents. For each

[13]We have to set a bound because if we were to duplicate the set of vertices to infinity, we would run out of memory.

makespan, the problem is encoded into a propositional formula $\mathcal{F}(\mu)$. This formula is consulted with the SAT solver, and an assignment is returned. If the formula could not be satisfied, the makespan is incremented, and a new formula is created. If a satisfying assignment has been found, the MAPF solution is extracted from this assignment and returned. The framework, how it is described in Algorithm 2, is not complete because if a solution to given MAPF instance does not exist, the solver would never stop. The existence of a solution is usually checked with another algorithm, for example, Push-and-Rotate.

Chapter 2

Ozobots

This chapter analyzes the Ozobot and its capabilities to a considerable depth. The Ozobot, though it might seem like a simple line-following bot, is full of sensors and functionalities. Should a group of Ozobots be used for a task like a simulation of MAPF, all their capabilities need to be put into consideration.

Most of the information on how to use Ozobots can be found on the Ozobot website [48]. Even a better source of guides and manuals is the Ozobot Classroom [49], where a comprehensive *training for educators* can be accessed. Although this would give an Ozobot user a complete overview of the basics, the specific details are often hard to find, if at all possible. Therefore the best way to learn about Ozobots and their capabilities is through testing and playing with them.

2.1 What is Ozobot?

Ozobot is a small, award-winning robot developed by Evollve Inc., whose primary goal is to inspire children to be creative with technology. With tens of thousands of classrooms utilizing Ozobots, it is not only used for teaching

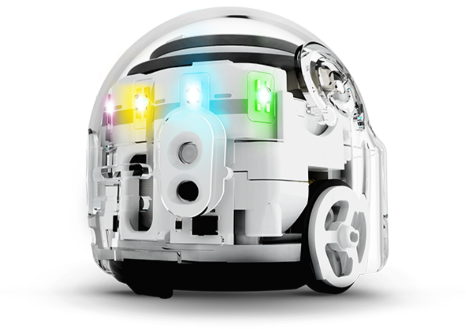


Figure 2.1: Ozobot Evo (photo from [50])

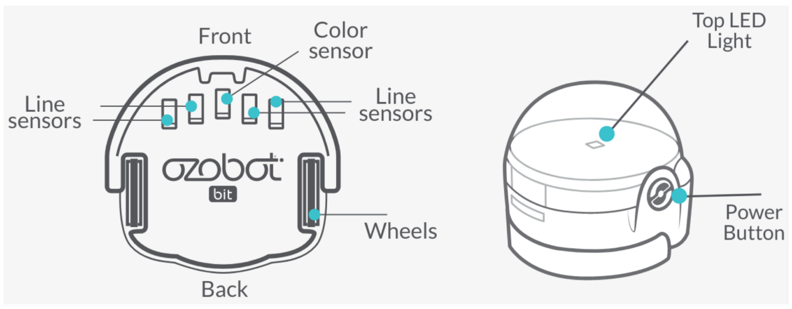


Figure 2.2: Sensor layout of Ozobot Bit (photo from [52])

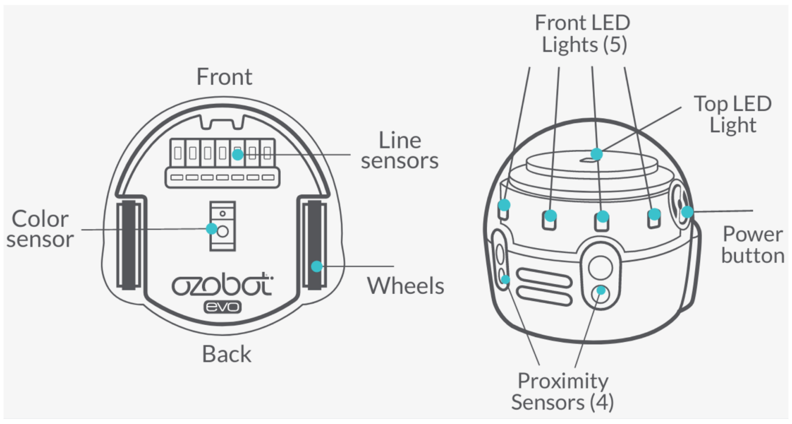


Figure 2.3: Sensor layout of Ozobot Evo (photo from [52])

computer science and coding, but also for demonstrating other theoretical concepts. Ozobot is also great for creating games and puzzles that are being shared through the Ozobot community.

At this moment, Ozobot can be bought in two versions: Ozobot Bit and Ozobot Evo. The *Ozobot Bit* was created in 2014, and it is the older and simpler version of the two bots. The creator of the Ozobot Bit has also written a detailed guide for parents and teachers [51], containing setup and maintenance instructions as well as short tutorials. Even though this guide is written specifically for Ozobot Bit, most of the information applies to both versions of Ozobot. *Ozobot Evo*, created in 2016, is very similar to its predecessor and can do everything that Bit can. The main difference is that Evo is packed with even more hardware, providing the user with more capabilities than the lighter version of Ozobot. A visual breakdown of both Ozobot Bit and Ozobot Evo can be found in Figures 2.2, 2.3, and 2.4.

Since the robots selected for this work are the Evo versions of Ozobot, the text will only refer to the Ozobot Evo model, from now on. An image of the Ozobot Evo can be found in Figure 2.1.

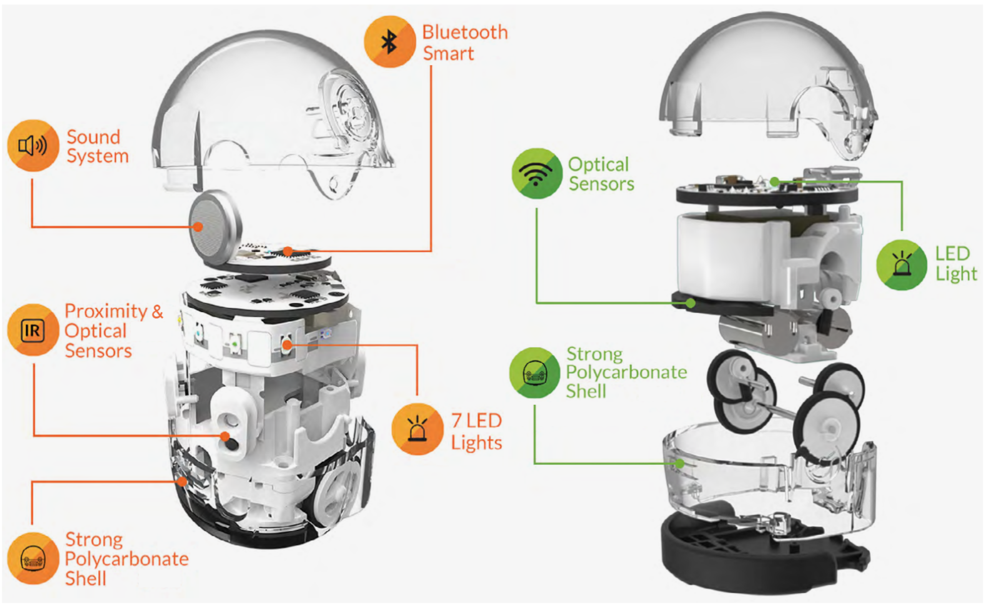

Bluetooth Smart
Sound System
Proximity & Optical Sensors
7 LED Lights
Strong Polycarbonate Shell
Optical Sensors
LED Light
Strong Polycarbonate Shell

Figure 2.4: Breakdown of Ozobot Evo (left) and Ozobot Bit (right) (photo from [52])

2.2 Capabilities

This section provides an overview of the Ozobot Evo and its hardware with a particular focus on the capabilities that it provides. The capabilities of the robot are further discussed in more detail later in the section. The default behavior of Ozobot can also be altered, and it can be done in two different ways. The first one is using *Color Codes* that are covered in Section 2.3. The second, more advanced, is coding in the visual programming editor *OzoBlockly* that is covered in Section 2.4.

2.2.1 Hardware

The body of Ozobot has almost a shape of a spherical dome with a flat base at the bottom. The diameter of the base and its height are around three centimeters, making it a tiny robot. Evo is equipped with a considerable amount of useful programmable sensors and other hardware inside its polycarbonate protective shell.

The most crucial part that is providing Ozobot with the movement is its *motor and wheels*. The wheels are shifted towards the back of the robot. That makes some of its mass to rest on a small protrusion located in the front of the base. The wheels have a rubber surface that makes it easier for the bot to move on smooth surfaces like a tablet display. Ozobot turns by moving the wheels separately at different speeds.

Under the base of the Ozobot are several line sensors arranged in an array and one optical color sensor. The purpose of the *line sensors* is to detect lines and intersections, so the robot knows what direction to follow or if it is time to choose a new one. On the other hand, the *color sensor* can detect the color of the surface on which the Ozobot is standing. The sensor can distinguish eight different colors: red, green, blue, cyan, magenta, yellow, white, and black. Ozobot classifies any color like one of these eight. For example, all shades of red are classified as red color. However, a color like orange would be classified either as yellow or red, depending on its shade. Ozobot is highly dependant on the color sensor because it allows the robot to read *Color Codes*[14], and it is also used to load programs into the Ozobot's memory[15]. Evo also has four *infrared proximity sensors*: two in the front and two in the back. These sensors can be used to detect an object as far as 10 centimeters from the robot.

The robot does not only have sensors for getting information from the environment, but it also has a few output devices. Namely, it is a *speaker* and seven *LED lights*: one on top, one in the back, and five in the front. Evo can use the speaker to play any note from four octaves[16], 128 MIDI notes, as well as some prerecorded words or emotions. The lights are standard RGB LEDs that can produce almost any color. The intensity spectrum of each color component is scaled down from 0–255 to 0–127. That technically means that only half of the colors can be reproduced, but a typical user will never notice.

Ozobot Evo is also equipped with *Bluetooth*, making it able to connect to other devices. However, connecting to other devices is only useful with a selection of applications that have been developed specifically for the bot. Evo holds a built-in *135 mAh LiPo battery* that can be charged with a micro USB cable and provides the bot with enough power to run for approximately 60 minutes. This time is variable and depends on the selected brightness of Ozobot's LED lights, the usage of the speaker, and the surface the bot is operating on. It is easier for the robot to move on a smooth display surface than on a paper. Last, the bot has a *power button* located on the side that is used not only for powering the robot on and off but also for running loaded programs and calibrating the sensors.

In comparison with Ozobot Bit, Evo is more sophisticated. The lighter version of Ozobot does not have Bluetooth, proximity sensors, nor a speaker, and has only one LED light on top compared to seven that Evo has.

2.2.2 Movement and line following

Ozobot is very good at following lines, and it is the default functionality the user gets every time the robot is turned on. However, the default behavior of Ozobot is not as simple as following black lines on white paper. When turned

[14]Explained in Section 2.3
[15]Explained in Section 2.4
[16]Octaves 5, 6, 7, and 8.

on, the bot does not move unless it founds a line to follow. When there is a line under Ozobot, it follows it at a default speed of 30 m/s. The motor and wheels provide the movement, and turning is executed by rotating both wheels separately at different speeds. Thus Ozobot can also follow curved lines, turn at intersections, or do a U-turn. When the bot finds itself at an intersection, it chooses one of the possible directions[17] at random, with all the possibilities having the same probability of being chosen. It is important to know that Ozobot does not register a 90-degree turn as an intersection. When the robot finds an end of the line, it stops its motor. While following lines, Ozobot will also read and execute *Color Codes* if found. Color Codes will be explained in Section 2.3.

Ozobot can follow lines either on a paper or on display, and it can also transfer between these two surfaces. If using a monitor, Evo behaves correctly on both matte and glossy types of display. Nevertheless, it is always recommended to calibrate Ozobot before every use to ensure the correct functionality of its optical sensors. This is especially true when there is a change in the display brightness or the surrounding light conditions. The calibration can be done in two ways. If a display is used for calibration, the user needs to press and hold the power button for two seconds. The top light will start flashing white, which signalizes that the bot needs to be put on a white background. The bot performs its calibration and signalizes a success by top light flashing green. Red light signalizes a failed calibration. If a white paper is used for the calibration instead, the user needs to draw a black circle slightly bigger than the Ozobot and use it for the process. The steps are the same, but this time the bot is placed on the black circle. The robot performs a full rotation and exits the circle in order to calibrate.

Even though Evo might seem reasonably tolerant when it comes to the quality of lines, there are a few recommendations to ensure the best accuracy. The color of the following line does not matter as long as it contrasts enough to the white background. The thickness and regularity of the line matter the most, and the recommended thickness for the following line is around 5 millimeters. The bot will also follow thinner or thicker lines, but the problems might come with turns and intersections, which might not be detected. The same goes for lines of irregular thickness. The angle of a turn might also affect the performance of the robot. Broad curves and 90-degree or wider turns work perfectly. However, when using very sharp angles, the bot might not detect the turn. The design of intersections is can also be crucial for Ozobot. The intersections work the best when lines meet at a right angle. This is not always necessary, but when more than four lines meet at a point, the chances are that Ozobot will be unable to detect one or more of the directions.

Both movement and line-following behavior can be modified via *OzoBlockly*. OzoBlockly will be explained in Section 2.4.

[17]Except for going back from where it came.

2.2.3 Lights and sounds

As mentioned in Section 2.2.1, Evo has one speaker and seven LED lights. The bot is using these output components to express its personality or to react to something. If the speaker is not muted, what can be done in the official application for Ozobot Evo, it will make a sound every time turned on, calibrated, or if a program was successfully loaded into its memory. Nevertheless, the capabilities of the speaker are beyond those few sounds. Via *OzoBlockly* programming, the speaker can be used to play prerecorded words or different notes. Evo can say any number between -127 and $+127$, each of the eight supported colors, four directions, and four different emotions. Other emotions and sounds are also available in the official application for Ozobot Evo.

Ozobot usually uses its lights to imitate the color of the surface on which it is standing. The lights are also used for signalization. When a calibration of optical sensors or program loading fails, the top light flashes red, and if successful, it flashes green. The lights flash and change colors according to the battery status when the robot is charging. Red color means low, green color means almost charged. When the robot is fully charged, the flashing stops and the color remains green. All lights except the back one are programmable and can be accessed via *OzoBlockly*. The only purpose of the back LED light is to signalize a low battery.

2.2.4 Object detection

Evo can use its proximity sensors to detect objects behind or in front of it. By default, this function is disabled. Should there be an obstacle in front of the robot, it will collide with the obstruction. However, all four proximity sensors are programmable via *OzoBlockly*, so the user can use them if wanted. Note that object detection works best on a flat, even surface. If the surface is uneven, false detection might be triggered, but this can be suppressed by configuring the sensitivity of the sensors.

2.2.5 Programmability

Due to the primary goal of Ozobot, programmability is an essential aspect of the tiny robot. Almost all of its sensors and functionalities are accessible to the user via a high-level abstraction. No knowledge in robotics is needed, and people are encouraged to develop original games or applications for the bot. There are mainly two ways to code with Ozobots: *Color Codes* and *OzoBlockly*. Both of these methods will be explained in Sections 2.3 and 2.4.

2.2.6 Applications

There are currently three applications for Ozobot available for both Android and iOS: *Ozobot Bit*, *Ozobot Bit Groove*, and *Evo by Ozobot*. The first two have been developed for Ozobot Bit and are not compatible with Ozobot Evo. However, the Ozobot Bit application can be useful for Ozobot Evo to some extent.

2.2.6.1 Application Ozobot Bit

This application is the main application for Ozobot Bit, and it is designed for tablets only. There are several minigames and challenges available for Bit. The most useful mode, which can also be useful for Evo, is *Freedraw*. This mode allows the user to draw the following lines on the screen and to use *Color Codes*. It is an excellent sandbox for exploring Color Codes and testing the behavior of the robot. Besides the different modes available in this application, there is also a feature called *Ozobot Tuneup*, where the user can perform a sensor calibration, and set some options for Ozobot Bit, like LED color or movement speed. From these features, only the calibration is compatible with Evo.

2.2.6.2 Application Ozobot Bit Groove

Ozobot Bit Groove can be used to design a dance for Ozobot Bit. The dance can be loaded into several Ozobots placed on the screen and executed simultaneously. The loading is done by *Flash Loading*, which is explained in Section 2.4. The application is designed for tablets only and is not compatible with Ozobot Evo at all. It also has the same *Ozobot Tuneup* feature as application Ozobot Bit.

2.2.6.3 Application Evo by Ozobot

The application Evo by Ozobot is more sophisticated than the other two applications, and it is incompatible with Bit. In this application, the user can create or sign in with an account that is also used on the *OzoBlockly website* [53]. Ozobots can be connected to the application via Bluetooth, and all of them can be managed from here. Ozobot ownership can be claimed via this account, and also Evo's firmware updates are done from the application. The user has the ability to rename the bot, mute its speaker and change the brightness of its LEDs. Other Ozobot users with this account can be added as friends.

The application can serve as a remote control for connected Ozobots. Most importantly, the OzoBlocky environment is included, and the code can be loaded and executed on multiple Ozobots at once, all done via Bluetooth connection. Unfortunately, Ozobots need to have over half of their battery charged in order to perform this loading, and the code cannot be loaded with *Flash Loading* like it would be done in the browser version of OzoBlockly.

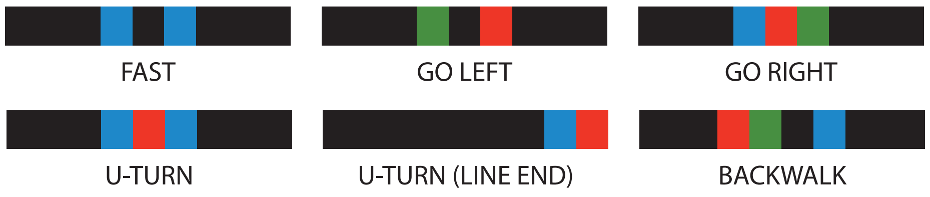


Figure 2.5: A selection of available Color Codes (adapted from [54])

2.3 Color Codes

The *Color Codes* are a unique way to tell Ozobot what to do, and it is referred to as a screen-free coding because the standard way of using Color Codes is with paper and markers. Since Ozobot has an optical sensor, it can recognize different colors, and the concept of Color Codes is based around this capability. A Color Code is a sequence of colors that the bot can read and react to during its movement on a line.

Color Codes use four colors: red, green, blue, and black. They can be drawn with classic markers, but are also available in a sticker format. The set of Color Codes and what they do is predefined. There are almost 30 Color Codes that Ozobot understands. They are mostly sequences of three or four colored rectangles in a row. Only three of the Color Codes are sequences of two colors and are only used at the end of a line. Some of the Color Codes are also symmetrical, and Ozobot can read them from both sides. However, other codes have different meanings depending on the side from which the Ozobot reads them. A selection of Color Codes can be found in Figure 2.5. Note that all Color Codes displayed in this work are read from left to right.

Just as for line quality, there are some recommendations for Color Codes as well, to make sure Evo can read them correctly. The essential requirement is to use white paper[18] and black lines. Color Codes should not be placed on colored lines or intersections. The different colors of a Color Code should not overlap, nor should there be a white space between them. It is also advised that all colored sections have the same size. If the colored sections are too long, Ozobot will think that it is only a change of line color and will not execute the command. If the sections are too short, some colors might not be detected at all. The recommended length of a color section is the same as the thickness of the following line.

2.3.1 Flashing Color Codes

A special kind of *Color Codes* can be found in the *Freedraw* mode of Ozobot Bit application, where the user can also use the classic Color Codes while drawing.

[18]Alternatively, white background if using a display.

When the Color Codes are placed on the screen, the user can tap on them, and they change into a circular point that is rapidly changing colors. This type of Color Codes has the same effect on Ozobot, but they work differently. First, a different set of colors is used: red, blue, black, cyan, magenta, and yellow. Each command is coded into a sequence of several colors, and the sequence is flashing in a loop very quickly. However, Evo cannot directly drive through this flashing point and read it. The robot needs to stop on the point for a split second to read the whole sequence, only then the command is executed.

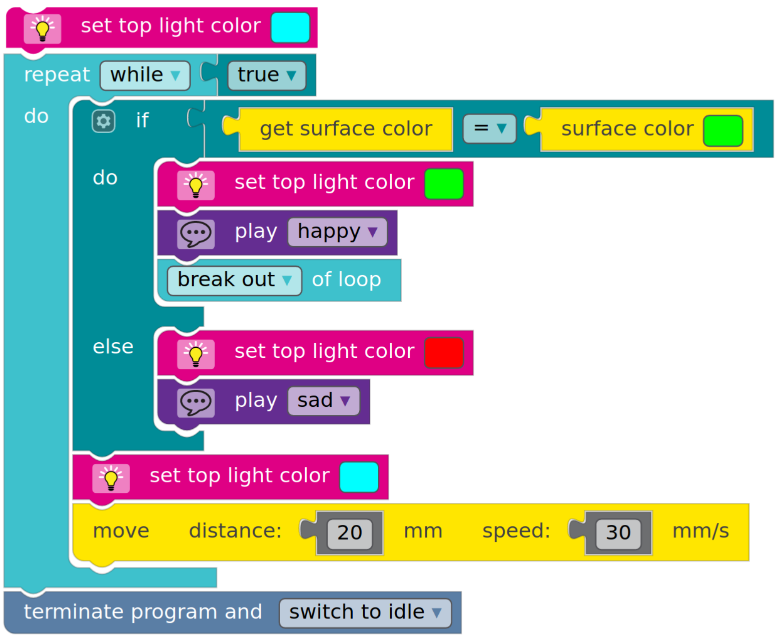
Unfortunately, this type of Color Codes has no documentation, and a human eye cannot register the color sequence. However, if the flashing is captured in slow-motion, the color sequence can be decoded. For example, the U-turn Color Code has a sequence of red, yellow, cyan, and yellow colors. When played in a loop at a rate of twenty flashes per second, Evo can easily decode the command.

2.4 OzoBlockly

OzoBlockly is a visual programming editor, powered by *Google's Blockly library*. The editor represents coding concepts as interlocking blocks, with which a program for an Ozobot can be built. The program can then be loaded into the bot's memory either via Bluetooth or *Flash Loading*. OzoBlockly can be accessed and used at the *OzoBlockly website* [53], but also in the official Ozobot Evo application and the *Ozobot Classroom*. The editor provides five skill levels, each adding more advanced coding concepts and block to use for a program. The first level only has a few basic blocks with pictographs of simple Ozobot moves, and the fifth level allows almost full control over the robot and its components. The OzoBlockly editor can be connected with the Ozobot account mentioned in Section 2.2.6, and up to twelve programs can be saved for the account. The programs can also be saved as a file in *XML format* and loaded back to OzoBlockly later on.

An example of a simple program in OzoBlockly editor can be found in Figure 2.6. When an Ozobot starts the execution of this program, it changes its top light color to cyan and starts its movement cycle. In each iteration, the bot reads the surface color from the optical color sensor. If the surface color is green, it changes the top light color to green, plays a happy emotion, breaks out of the loop, and terminates the program changing its state to idle. If the surface color is not green, the robot changes its top light color to red and plays a sad emotion. At the end of the loop, it turns the top light back to cyan and moves 20 millimeters forward. In other words, Ozobot moves forward until it finds a green surface.

In this work, all OzoBlockly programs will be rewritten into text for readability and consistency. The code for the program from Figure 2.6 is rewritten in Algorithm 3. Note that it is almost the same syntax as with the OzoBlockly



```
set top light color
repeat while true
do  if get surface color = surface color
    do   set top light color
         play happy
         break out of loop
    else set top light color
         play sad
    set top light color
    move distance: 20 mm speed: 30 mm/s
terminate program and switch to idle
```

Figure 2.6: An example of Ozobot program in OzoBlockly editor

Algorithm 3: Ozobot program example

```
1  top light color ← aqua
2  while true do
3      if get surface color = surface color green then
4          top light color ← green
5          play happy
6      else
7          top light color ← red
8          play sad
9      top light color ← aqua
10     move distance: 20 mm speed: 30mm/s
11 terminate program and switch to idle
```

code blocks, and it can be used as pseudocode.

2.4.1 Flash Loading

Flash Loading is a method used for loading programs from OzoBlockly into the Ozobot. Both Bit and Evo support this method, which is possible because of their optical color sensors. First, the OzoBlockly program is encoded as a color sequence using all eight colors that the bot can identify[19]. The bot is placed on a designated spot, where the sequence will be run. When the loading starts, the sequence is flashing at a rate of twenty colors per second, and Ozobot can read every color of the sequence, decode the program, and save it into its memory. The length of the sequence depends on the complexity of the program. Thus the loading can take anywhere between a few seconds to several minutes. The same concept is used for *Flashing Color Codes* that are explained in Section 2.3.

2.5 Limitations

Unfortunately, Ozobot cannot do everything, and some limitations come with it. The following limitations might even have a significant impact on the realizations of such tasks as MAPF simulation.

2.5.1 Ozobot communication

The Ozobot's proximity sensor is a pair of infrared emitter and receiver. However, there is currently no way to establish an infrared communication between two bots. Even though the proximity sensors are programmable, the only functionality available is to detect objects and alter their sensitivity through OzoBlockly. Evo also has Bluetooth to connect to other devices, but it is impossible to make a connection between the bots and transfer data.

This lack of ability to communication means that a group of Ozobots cannot share information about their status, nor the environment. Using Ozobots for simulation of MAPF that allows agents to exchange information about the environment, for example, is impossible.

2.5.2 Custom Color Codes

One of the unique features of Ozobot is the Color Codes. It provides an ability to tell Ozobot what to do via the environment. The analogy of Color Codes could be traffic signs, for example, and it could be beneficial for some applications in artificial intelligence. However, the significant limitation is that there are only a few predefined Color Codes, which are great for creating Ozobot racing tracks, but not as much for research. Unfortunately, there is currently no way of designing additional Color Codes.

2.5.3 OzoBlockly blocking operations

Most of the commands and operations in OzoBlockly are blocking, and Ozobot executes them one by one, but not in parallel. This creates a limitation for

[19]The colors are: red, green, blue, cyan, magenta, yellow, white, and black.

creating a modified line following behavior. While the bot is executing a command `follow line to next intersection or line end`, other commands like `read proximity sensor` or `get surface color` cannot be executed. So, for example, if a user wants the bot to use proximity sensors while moving, parallelism has to be simulated by iterating through small movements and proximity sensor readings. However, this is not always possible for every scenario, and some other workaround might be needed.

2.5.4 Text-based programming language

Although a visual editor like OzoBlockly is good for comprehending basic coding principles, it is very inconvenient for the implementation of complicated algorithms or the development of extensive programs. OzoBlockly editor can show a preview of the current program in JavaScript syntax and even highlight the code representation of the selected block. Unfortunately, there is no way to write the code for Ozobot in JavaScript or other text-based programming language and import it into OzoBlockly, or even straight to the Ozobot. Some projects have tried or aim to develop a text-based interface for Ozobot programming. However, they are usually intended for Ozobot Bit and therefore support only basic commands for the bot. One of the used approaches is to construct the programs in a specific XML format that OzoBlockly uses and can be loaded into the editor.

2.5.5 Online reprogramming

The process of loading programs into the memory of a group of Ozobots is relatively complicated. When using the Flash Loading, only a small number of robots can be programmed at once, and they have to be manually moved to loading spots on display. Even the execution of the program has to be activated with the power button located on Ozobot. Bluetooth loading and execution make this process easier. However, this method requires the Ozobots to be almost fully charged. Otherwise, it is impossible to transfer the program into their memory. This makes it impossible to reprogram Ozobots instantaneously while they are executing another program, or change instructions they are following.

Let us assume a MAPF simulation solution that is transforming all planned paths into a set of instructions for each Ozobot agent, is implemented. These paths would be loaded into each agent and executed simultaneously. It would be impossible for this solution to support replannings during the execution because the reprogramming cannot be performed in such a manner.

2.5.6 Other limitations

There are also some other minor limitations, which should not affect scientific research or algorithm development utilizing Ozobots. For example, the

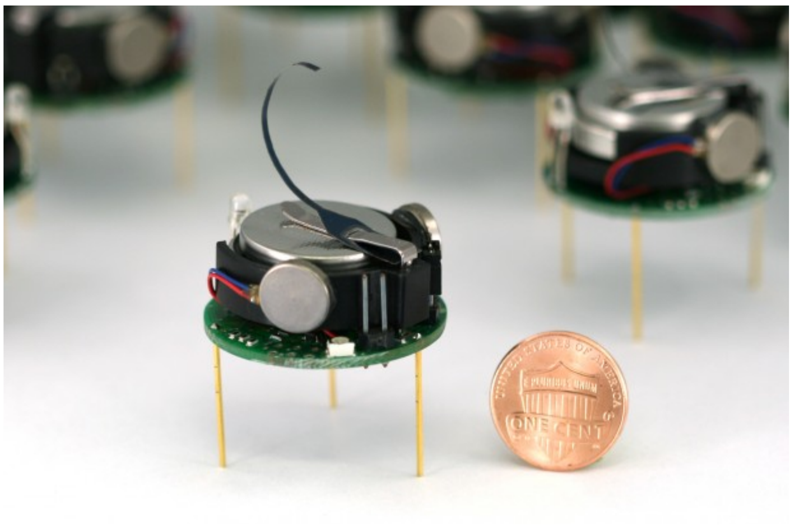


Figure 2.7: Kilobot (photo from [55])

inability to record custom sounds for the Ozobot to play, or that the back LED light is not programmable.

2.6 Alternative robots

Ozobots were chosen to be used in this work mainly because the faculty provided a group of these robots for the study. Another critical factor is their *affordability* compared to other similar robots, which makes it easier for other researchers to get access to a group of these tiny bots. For some, a positive fact might be that all capabilities of Ozobots are available without the need for any knowledge in robotics or low-level programming. There are several other robots available on the market and are also being utilized in research. This section explores a few of these robots.

2.6.1 Kilobot

Kilobot was developed at Harward University and is now being produced by K-Team. It is a small bot almost identical in size as Ozobot. Kilobot is being used in research, especially in swarm simulation, because of its tiny size, low price, and capabilities. The price of one Kilobot is about a seventh of the price of Ozobot Evo. However, a group of ten Kilobots can be bought at the same price as a group of twelve Evos. This is because a special *Kilobot Controller*, which is used for controlling a group of Kilobots simultaneously, and a *Kilobot Charger* are included in the package. The cost-efficiency of Kilobot shows with big groups of robots. A photo of Kilobot can be found in Figure 2.7.

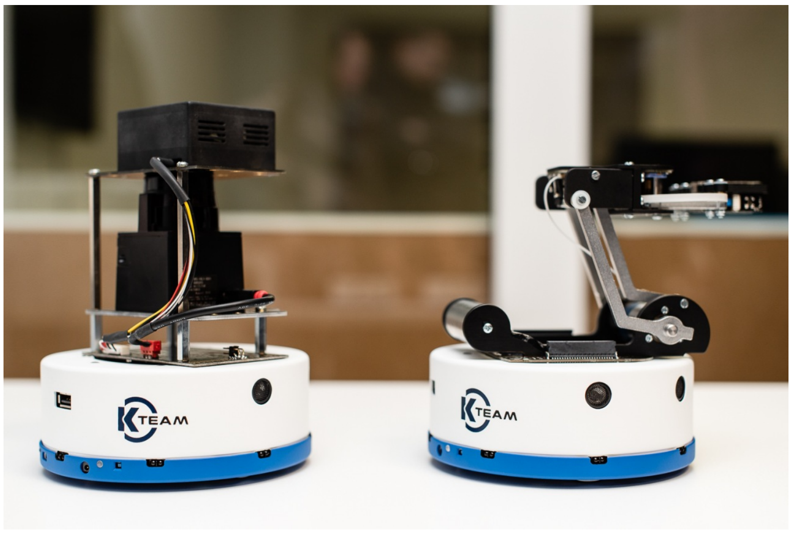


Figure 2.8: Two Khepera IV robots with different expansion modules (photo from [56])

The capabilities of Kilobot are more limited compared to Evo. The most crucial capabilities are the ability to communicate with different Kilobots in a seven-centimeter radius and a sense of distance from its neighbors. The robot can also sense ambient light and has one RGB light. These capabilities make Kilobot a perfect choice for simulating swarms, but MAPF would not be that easy to simulate with them.

2.6.2 Khepera IV

Khepera is another robot for education and research that is being produced by K-Team. This one, however, is a very robust and sophisticated robot. It has a shape of a puck with a diameter of 14 centimeters, a height of almost 6 centimeters, and weighs over half a kilogram. This robot has a lot of sensors and considerable computing power. The bot can use its sensors to follow lines and detect obstacles as far as 25 centimeters from the robot. The robot has speakers, microphones, and even an integrated color camera that can be used for advanced object detection and recognition. The most interesting feature of Khepera IV is its *modularity*. Different expansion modules can be added to the robot, as shown in Figure 2.8. It can even move a payload of two kilograms. The robot can run native on-board applications written in C/C++, which makes it very convenient for development.

With its capabilities, it should be no surprise that Khepera IV is suitable for experiments in navigation, artificial intelligence, real-time programming, and MAPF simulation. However, one Khepera IV robot costs over three thousand dollars, which makes any multi-agent system very expensive.

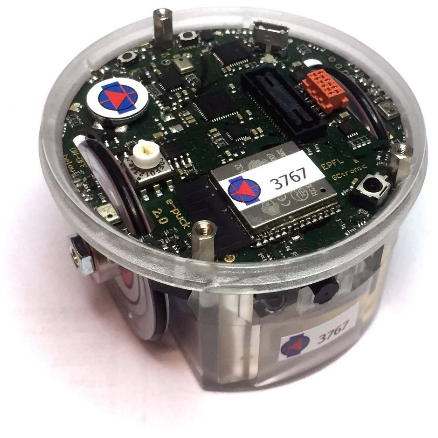


Figure 2.9: E-puck 2 (photo from [57])

2.6.3 E-puck

Another robot very similar to Khepera would be *E-puck*. It is less complex and smaller than Khepera, but still powerful enough to be used in the research of multi-agent systems. Although it is considered a low-cost robot, its price crosses a thousand dollars per unit. The E-puck robot is shown in Figure 2.9.

Chapter 3

Simulation of MAPF

Simulation or execution of planned paths is beneficial in MAPF because it helps the researchers to *visualize* the plans of their algorithms and solvers. An algorithm usually yields a solution to a given problem in the form of position or step sequences for all agents. For larger problem instances, it is often impossible to imagine the movements of agents in the environment to confirm the correctness of the found solution. Therefore, a simple animated simulation can be used to aid debugging processes or demonstrations.

This chapter explores the *simulation of MAPF*, especially one particular research that has been done with the use of Ozobots. Then a *novel approach* that this work chose to analyze and realize is presented.

3.1 Previous work

MAPF simulators are not usually published along with the main work of a research paper, but they can be used to create visualizations for a publication. These are often simple programs that take a generated plan as input and animate the agent execution on a computer screen. The simulator can be as simple as a grid that swaps agent icons between neighboring cells in each time step or as polished as a fully continuous animation of agent representations in an aesthetically pleasing environment. Either way, the simulator is developed on top of a similar abstraction as the solver, and the solution is rarely executed with physical robots. Therefore, the *practical usability* of a given abstract solution is usually unknown.

In [47], the authors used Ozobot Evo to simulate MAPF. That work aimed to study how suitable MAPF plans are for a real-robot execution. First, the authors had to figure out how to simulate abstract MAPF solutions with Ozobots, and then how the quality of these plans affect the execution of the bots. The study did not focus on solving MAPF problems, but rather on the simulation and practicality of the obtained solutions. Therefore, an optimal

SAT-based solver implemented in Picat programming language [58] was chosen for the study. This choice was partially made because of the ability of this solver to modify or extend the core model easily. The authors utilized this feature and tested three different models[20]: classical, robust, and split actions model.

The *classical MAPF model* is the same, as explained in Section 1.2. The solution using this model was directly translated into the movement primitives the robots can perform, such as rotation and movement. For the *robust model*, k-robustness [20], which is mentioned in Section 1.2.3, was incorporated in the model. The authors chose k to be 1, and the 1-robust plan was translated into agent actions in the same way as for the classical model. The *split actions model* that the authors presented in this study is mentioned as one of the solutions to real-world MAPF complications in Section 1.2.3. This model represents directions and rotations of the agents as additional vertices and edges in the environment representation. Because the study uses a grid graph as the representation of the environment, each *position vertex* in the original graph is split into four *directional vertices*. These new vertices represent the orientation of the agent in the current position. Edges between these new vertices represent a change of orientation of the agent, which is a result of rotation action of the physical robot. The edges between neighboring positions in the original graph remain and represent the movement of the robot. The *collision detection* of the model had to be also modified. A collision would occur if any two agents were to occupy any of the four directional vertices of a single position. The authors also presented two other modifications to this split model in order to minimize the potential desynchronizations. These models were tested on a single map with four Ozobot agents, and the experiments concluded that the classical MAPF plans are not practical for physical robots. Other proposed models proved to be more suitable for this type of robot, and the solutions resulted in shorter execution times and fewer collisions.

For this work, it is more relevant to analyze how the *simulation with Ozobots* was done. The approach that the authors chose is simple. They have utilized the *movement primitives* that Ozobot provides through the OzoBlockly editor. These primitives influenced the physical environment representation as well as the simulation process. In Figure 3.1, a map of the tested problem instance from the study can be found. The *physical environment* was modeled as a grid of following lines, where each intersection represented a position vertex in the abstract representation. The following lines between intersections represent edges in the original abstract graph. If a vertex is removed from the abstract representation, an intersection is removed from the map. However, the adjacent lines cannot be entirely removed, because Ozobot might not recognize the neighboring positions as intersections.

[20]Six models were tested because the three mentioned models were also modified in order to try keeping the agents synchronized.

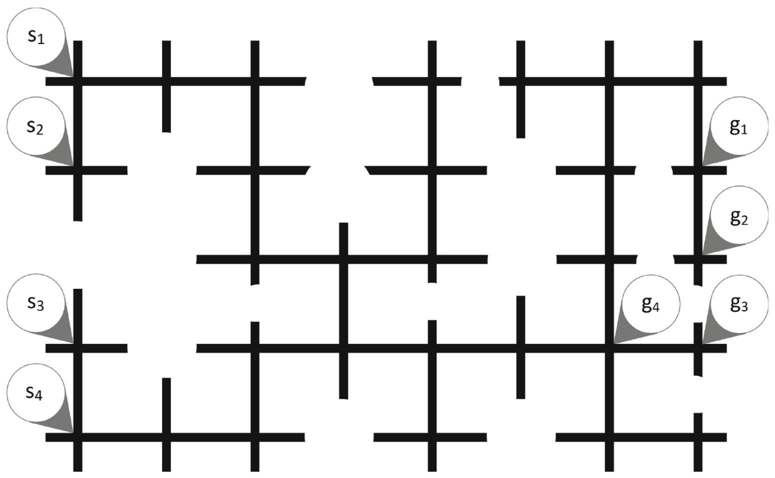


Figure 3.1: A map of the problem instance from [47] used to simulate MAPF on Ozobots

This map would then be correctly scaled and printed on a paper for the Ozobots. The thickness of the lines was chosen to be five millimeters, as it is recommended, and the length oh the line between two intersections was chosen to be 5.2 centimeters, so as the Ozobots could safely reside in neighboring intersections. With this map representation, authors were able to use Ozobot commands like `follow line to next intersection or line end`, `rotate left`, or `rotate right` to navigate the environment.

The abstract plan obtained from the solver can be directly translated into these primitives for Ozobots. The sequence of actions is then loaded into Ozobot as a program. So technically, each of the robotic agents has *memorized* its path for the execution. The solution can then be simulated by placing the bots on their starting intersections in the *correct orientation* and running their programs. The issue is that the plan has to start *synchronously*, meaning that all agents have to start their program execution at the same time. With Ozobots and their capabilities, there is no simple way of doing this in this approach. It is unreliable to start the programs simultaneously by hand, and the bots cannot communicate in order to synchronize their initial move. The authors resolved this by using the *proximity sensors* of the robots at the beginning of their programs. Before the simulation, obstacles are placed in front of the bots to prevent their plan execution. When all agents are at their position and ready to start the simulation, all obstacles are removed at once. This workaround, however, still requires *manual interference* and can result in minor desynchronization. Overall, this approach is minimal in terms of the utilization of Ozobot capabilities and works well for the study.

Later in [59], the authors presented *software* allowing users to utilize this simulation approach and to create their MAPF scenarios. The users can design their own MAPF instance, and the map can be printed or displayed on the screen for the Ozobot execution. The system provides several MAPF models and the ability to add custom models written in Picat programming language, which are then used for solving the MAPF problems. The solutions found by these models can be either visualized on the screen or exported as an Ozobot program in XML format with user-specified configurations.

3.2 Novel approach

The previous solution from [47] and [59] mentioned in Section 3.1 is simple and works fine for the comparison of MAPF models. However, because of how that approach utilizes Ozobots, there is not much that can be done to enhance the simulation or expand the usability to other MAPF variations. Moreover, it has several issues or drawbacks that could be solved by using a different simulation strategy. In this section, a *new approach* for MAPF simulation is proposed, compared with the previous solution, and the previous drawbacks are explained as well as how the new approach solves them. At the end of the section, some problems that might arise with this approach are anticipated.

3.2.1 Navigation by environment surface outputs (ESO-Nav)

The main idea of this novel MAPF simulation approach is to have agents that have *predetermined behavior*, and an *environment that can output information* for the agents, affecting the behavior. The environment is a physical representation of a given MAPF problem instance that can navigate the agents in itself by showing them the planned paths as well as additional information. The plans are obtained from a *centralized MAPF solver* and then processed for the simulation in the environment. On the other hand, the agents need to know how to read and translate the information outputted by the environment. Note that the agents do not need to know anything about their planned paths and do not participate in the planning process. Therefore, they can be simplistic and entirely *modular* to the multi-agent system. This means that agents can participate in any problem instance simulation without changing their programming.

In this work, the agents are *Ozobots* with their ability to *follow lines*, and the environment is a computer or television screen. On the screen, a map of a MAPF problem instance can be displayed, and the planned paths animated for the robots. The behavior of the agents can be determined with a simple *OzoBlockly program* loaded into each bot. This approach provides freedom of creating a more sophisticated and highly customizable MAPF simulation. In Chapter 4, a working *prototype* of a MAPF simulator using this approach is implemented, and all aspects of this solution are explained in greater detail. For the MAPF problem solving, a classical discrete centralized MAPF solver will be used to showcase that it can also be simulated with this approach.

ESO-Nav approach could also be used in a variety of *real-life applications* like intelligent evacuation systems or transportation around amusement parks or in buildings and warehouses. For example, let us have a shopping mall as an environment with a floor that could display the following lines. A monitoring system could determine the positions of individuals or groups of people in the building. With a map of the mall and the positions of people and entrances, a MAPF problem instance can be created. Such an instance can be solved during an emergency. A centralized solver could find an optimal or near-optimal evacuation plan that could be animated on the floor. In this scenario, people are agents with behavior affected by stress and fear, which can lead to disorientation and inability to solve problems. Therefore, they are limited to follow only simple instructions that are provided by the environment.

3.2.2 Improvements from the previous approach

Most of the drawbacks of the previous approach came from how the Ozobot capabilities were utilized. Another factor was that the agents had to memorize the paths before execution. The novel approach solves some of these drawbacks by using *different simulation strategy*. The previous issue with synchronization of the execution start can be easily solved. Before the start of the simulation, no paths are displayed on the map for the agents. They have *nothing to follow*, and their behavior can *initiate waiting* in that situation. In the previous solution, the map of the problem instance was constructed from the following lines, which leads Ozobots into movement. This environment structure is also hard to modify or extend to other variations of MAPF. In contrast, the *ESO-Nav* provides more *flexibility* in environment representation and map design, since the planned paths are only a piece of extra information on the map. The novel approach can be, for example, used for simulating environments that are not grid-based, and the paths could be continuous curves instead of straight lines between positions.

The fact that the paths need to be loaded into the Ozobots before each simulation makes the previous approach less efficient in transitions between different problem instances simulations. Robots need to be manually reprogrammed if the map changes, and that cannot be done instantaneously. In the *ESO-Nav* approach, the robots can be quickly used on different maps without changing their programming. The paths can even be replanned during the execution without the agents noticing. This allows simulation of sub-optimal MAPF solutions that are being optimized during the agent execution, or replanning can be performed if agents fail to follow their paths. Moreover, different behaviors of agents can be tested and compared with this simulation strategy.

3.2.3 Expected problems

Most of the *real-world complications* from Section 1.2.3 are still concerning this novel approach, but the complications can be mitigated by correct path processing and outputting. Nevertheless, only the prototype constructed in Chapter 4 will discover the potential issues and if they can be solved. The agent *collisions* can still be an issue, even if a MAPF solver finds a valid solution. The environment needs to be designed so that agents can move around each other without any contact. Incorrect path processing and outputting can also introduce conflicts that are unanticipated and would interrupt the simulation. The biggest challenge remains the fact that different robot movements take different time durations, creating *desynchronization* in the plan execution. However, the *ESO-Nav* approach with the use of Ozobots provides different ways of solving this desynchronization issue without modifying the MAPF solver nor the problem abstraction.

Chapter 4

Prototype realization

In this chapter, a working prototype of the *ESO-Nav* approach to MAPF simulation that utilizes Ozobots is presented. The prototype can take a MAPF problem instance in the form of a map and perform a simulation of the solution. In Section 4.1, the basic idea of the prototype functionality is described, and all modules of the prototype and interactions between them are explained in greater detail. In Section 4.2, several versions of the main simulation module are incrementally improved. These versions gradually discuss and solve different issues of the simulation process that arise from various Ozobot limitations. Lastly, Section 4.3 presents the final version of the prototype.

4.1 Overview

The prototype application is written in *Python* and could be divided into several modules that interact with each other during the simulation process. Most of these modules should be easy to modify or extend to simulate different MAPF models or situations. The program takes a map file that contains the MAPF problem instance to be simulated. The problem is passed to a MAPF solver, that yields a solution. This solution contains paths for all agents that should be outputted to the environment. However, before this solution is passed to the simulator, it might require some processing depending on the MAPF solver used. The main simulator module then takes the obtained solution and ensures a correct environment outputting for the agents.

4.1.1 Agents

This simulator prototype is specially built for *Ozobot Evo robots*, which are used as agents in the system. All capabilities and limitations of these bots are carefully examined and described in Chapter 2. As mentioned in Section 3.2, the agents in the *ESO-Nav* approach to MAPF simulation do not hold any information about their planned paths. The agents instead act based

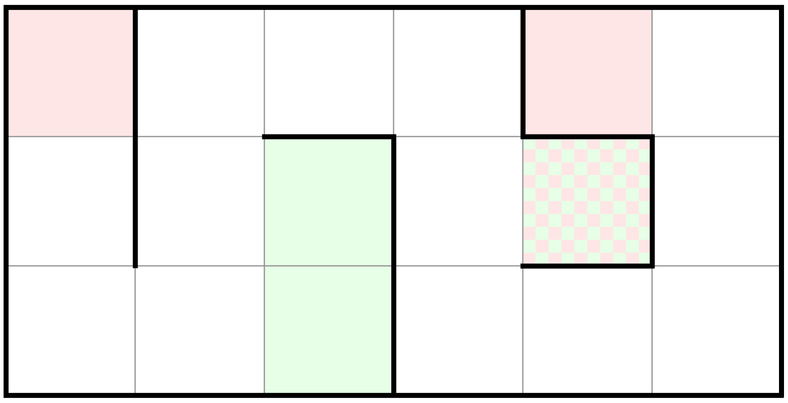


Figure 4.1: An example of a map displayed on the screen before the simulation

on their programmed behavior. The behavior of Ozobot can be changed with an OzoBlockly program loaded to the bot. However, in the first versions of the simulator, the *default behavior* of Ozobot is used without any modifications.

In later versions of the simulator, the default Ozobot behavior is not adequate for the task and needs to be changed. The behavior modifications and their justifications are presented in the version subsection that introduces them.

4.1.2 Environment

During the simulation, the *abstract environment* representation needs to be transformed into the *physical environment.* This is done by displaying the map on a computer screen on which the Ozobots can move. For this prototype, *grid-based maps* were chosen to be used, so as they can be represented as grid graphs for a MAPF solver, and graphically displayed as tiled maps. As the abstract representation of the environment is a graph, it needs to be converted into a more suitable representation for displaying, along with the solution found by the solver.

When the solution is ready, the map is stored as a tiled grid. Each *tile* of this grid can be accessed by the simulator and shown on the screen. An example of a map is shown in Figure 4.1. If there is not an edge between two neighboring vertices in the original graph, a *wall* is displayed on the map between the corresponding tiles. The walls are also displayed all around the map to indicate the perimeter. If a particular tile is a *start* or *goal* position of any agent, the tile is colored green or red color, respectively. If there is a start and also a goal position on a single tile, the tile is filled with both colors using a checker pattern. Note that the colors need to have very *low opacity*, so as the Ozobots do not register them as following lines. Before the simulation, Ozobots are placed on all green tiles, as shown in Figure 4.2, and at the end of the simulation, they should stand on the red tiles. When all bots are in

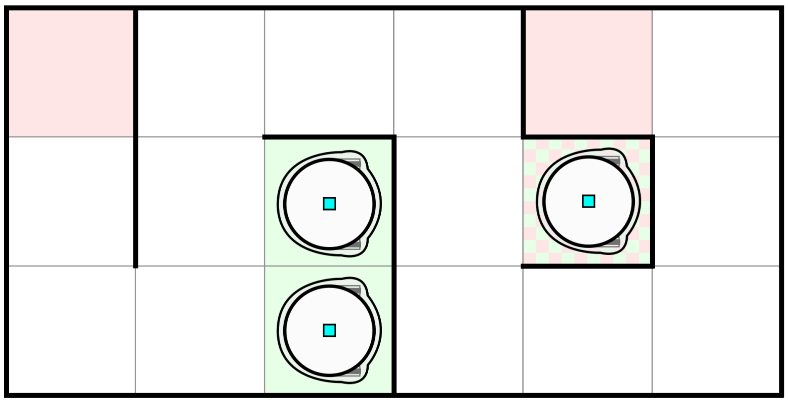


Figure 4.2: The map example with Ozobots ready to execute paths

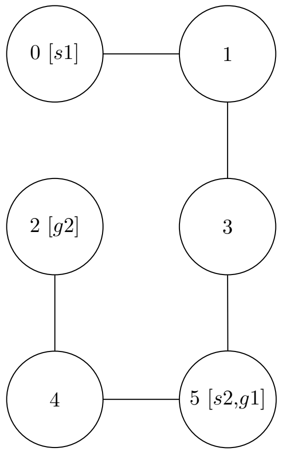
place, the *following lines* are displayed on the map as well by the simulation module.

The drawing on the screen is implemented in pyGame [60], around which a simple *graphical module* was created. The module provides a set of various objects and methods that the simulator can use to display all elements of the map. Since all screens have different sizes and resolutions, and the Ozobots depend on the output, it is crucial to ensure the *correct scaling* of the map elements. Therefore, the *screen parameters* need to be provided in a configuration file so as the application knows how to scale the map elements. These parameters also limit the maximum size of a map that can fit on the screen. Because of this configuration, the application can be easily used on different devices, including television screens, where extensive maps can be simulated.

In the configuration, the user can also set the width of different line types and the size of the tiles in millimeters. For this prototype, the tile size was set to five centimeters, the following line thickness to five millimeters, and the wall line thickness to two millimeters. The wall line is thick enough to be detected by an Ozobot, but the robot should never cross or come close to it. Note that all of these dimensions are converted to pixels and rounded, and therefore the actual displayed dimensions might slightly vary.

4.1.3 MAPF Solver

The *solver module* provides a simple interface to implement a custom MAPF solver, but in this work, no new MAPF solver is created. The prototype uses already existing program boOX [61] that implements several MAPF algorithms. Namely, algorithm SMT-CBS [62], which combines the SAT-based solving principle and the CBS algorithm, is used. All algorithms in this program are implemented under the *sum of costs* objective function and can also produce *1-robust* plans. The role of this module in the prototype is to run the boOX program as a *subprocess* and obtain the found solution for simulation.



(a) Underlying graph of the map example

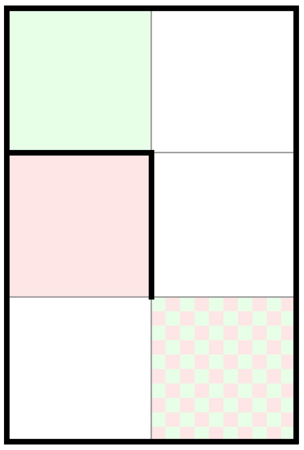

(b) The example map displayed by simulator

Figure 4.3: An example map in graph representation (left) and how it is displayed by the simulator (right)

4.1.4 Maps

The *maps* are stored as graphs in text files. In Figure 4.3, there is a simple map example in graph representation as well as how the simulator displays it. The $s1$ and $s2$ in the graph indicate the start positions of agent one and agent two. Analogically, $g1$ and $g2$ are their goal positions. In Figure 4.4, the *text representation* of this problem instance that would be stored in a map file can be found. On lines 2–7 are listed all *vertices* of the graph in format (v, s, g), where v is the index of the vertex. The s and g are indices of agents that have their start and goal positions in this vertex. If the agent index is 0, there is no agent's start nor goal position. On lines 9–13, all *edges* are listed in format $\{v_i, v_j\}$, where both v_i and v_j are vertex indices. Line 1 and line 8 state the beginning of the vertex list and edge list, respectively.

From both the graph in Figure 4.3 and the text representation from Figure 4.4, it is visible, which start and goal positions correspond to which agents. However, in the graphical representation displayed by the simulator, this information is missing. All Ozobots moving on the screen are considered to be *anonymous and indistinguishable*, and the information would have no added value for the simulation.

The boOX program also supports this map format. Therefore the simulator can directly pass the text file with the problem instance into the boOX subprocess for solving.

```
V =
(0,1,0)
(1,0,0)
(2,0,2)
(3,0,0)
(4,0,0)
(5,2,1)
E =
{0,1}
{1,3}
{2,4}
{3,5}
{4,5}
```

Figure 4.4: Text representation of a MAPF problem instance

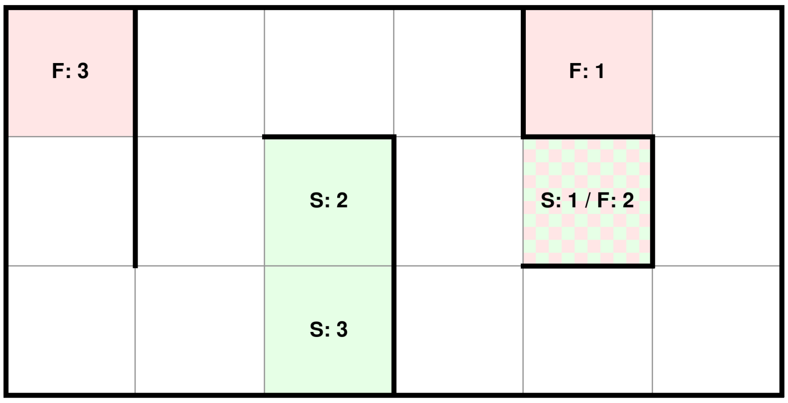


Figure 4.5: An example of a map displayed in the map editor

4.1.5 Simulator

The main and most sophisticated module of the application is the *simulator*. It takes the loaded map of the problem and an obtained solution from the solver module and is responsible for simulating the paths for Ozobots. The configuration is also accessible to the simulator because it is in charge of outputting the map elements on the screen. This module is described and iteratively modified in Chapter 4.2, where the problems of simulating MAPF on Ozobots are encountered and solved.

4.1.6 Map editor

Creating MAPF problem instances in a text editor directly in the text representation would be very inconvenient. Therefore a *map editor* is incorporated in the prototype. When the map editor starts, the user can define the new map's width and height, as well as the number of agents that will be simulated. After that, an empty tile grid is displayed with the map borders, that can be modified. In Figure 4.5, the example map from Figure 4.1 is displayed in the map editor. The editor operates in three modes: wall placement, start placement, and finish placement. These modes can be activated by pressing the `w`, `s`, and `f` keys, respectively.

The *wall placement* mode is activated from the start of the map editor. In this mode, the user can toggle walls on and off by clicking the mouse on the tile borders. By doing this, the whole map can be designed. The *start placement* and *finish placement* modes can be used to place or remove the agent's start and goal positions. This is done by mouse-clicking on the tiles while having the correct mode activated. The positions of agents are being placed in ascending order until all agents have their position. Every time an agent position is placed, it is assigned to an agent with the lowest index that has not been placed yet.

4.2 Simulator versions

The core of this solution is the simulator module. It is responsible for transforming the discrete solution obtained from the solver module into continuous following lines for Ozobots, and displaying them for the bots into their physical environment. This section goes over several versions of this module, each using a different plan outputting method, and each encountering and solving various simulation issues.

The process of performing a simulation is the same for every version of the module, only the path outputting differ. First, when the simulator obtains a solution from the solver, a *map preview* is displayed on the screen, as shown in Figure 4.1. The robots are then placed on their initial positions. When all agents are ready, the user can start the path outputting that should make the bots execute their planned paths without any manual interference. As mentioned in Section 4.1.1, the first versions of the simulator are used with the default Ozobot behavior.

4.2.1 ESO-OzoNav 0: Full paths

The simplest solution for the path outputting would be to display the whole plan at once. This version of the simulator takes the solution, which is a set of positions in each time step of the execution for every agent and draws the following lines between the corresponding tiles in the map. This way, the full path is displayed from the beginning of the execution, and the Ozobots can freely move on it. Figure 4.6 shows how the paths are displayed on the map. However, this simple outputting method creates several problems, which make it unusable for Ozobots.

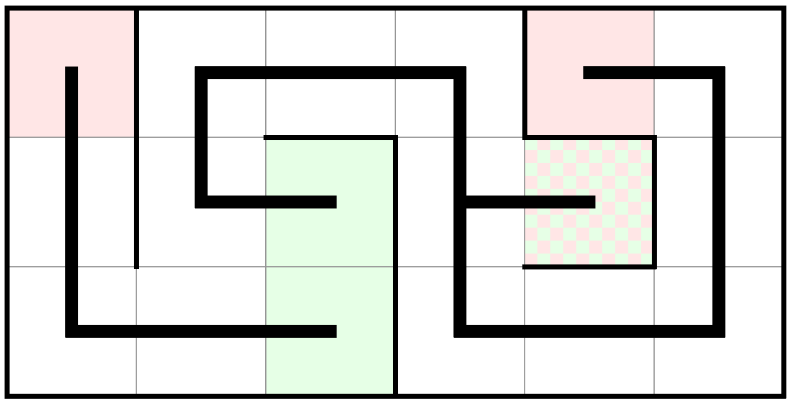


Figure 4.6: Full agent paths displayed in the map

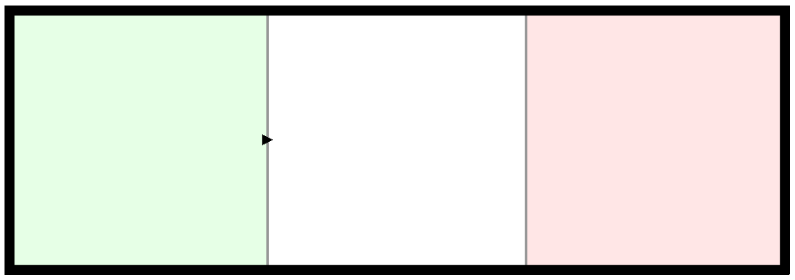


Figure 4.7: Initial agent orientation indicator displayed in the map preview

4.2.1.1 Initial orientation problem

The first problem that has to be solved in this prototype is the correct initial orientation of the robots. The problem is that the solver has no information about the initial agent orientations, nor does the user know how the plan looks before it is displayed. If the Ozobots face the wrong direction, they will be unable to detect the following line or move in the wrong direction, which will lead to an immediate loss of the line.

Because there is no monitoring system that could detect the initial orientations of the agents and feed that information into the solver, a correct orientation has to be ensured according to their individual planned paths. These initial orientation can be determined from the plan since it is found in advance and not during the simulation. These orientations are displayed in the map preview before the execution in the form of arrow indicators, as shown in Figure 4.7. Therefore, the user can place the Ozobots on their initial positions with the correct orientation based on these indicators. In Figure 4.8, the example map's preview is shown with the orientation arrow indicators. These indicators are incorporated in the following simulator versions as well.

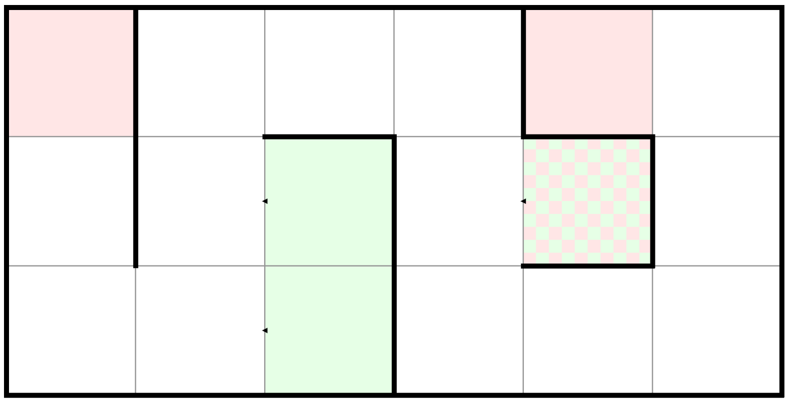


Figure 4.8: The example map preview with orientation indicators

4.2.1.2 Random direction problem

When paths of at least two agents cross, an intersection is created in the displayed following lines. An example of such a situation can be seen in Figure 4.6, where the tile $(3, 1)^{21}$ contains an intersection. As mentioned in Section 2.2.2, when an Ozobot comes to an intersection, it chooses its direction randomly. Nevertheless, in the plan execution, only one direction leads to the position, where the agent should be in the next time step. Failing to choose the correct direction would most likely lead to a collision or other execution failure. Generally, a plan where agents visit the same subset of positions in different time steps cannot be simulated this way.

One way to solve this is to program the correct direction choice into the robot's memory for all intersections that are present on its path. However, in the *ESO-Nav* approach to MAPF simulation, this is unwanted. This issue indicates that the paths cannot be displayed fully.

4.2.1.3 No wait problem

The ability to wait at a specific position is one of the two main actions an agent can make. Unfortunately, when the paths are fully displayed, Ozobot will stop only when it finds a line end. The wait action cannot be indicated, which will undoubtedly lead to large desynchronization and collisions.

When the Ozobot agent is required to stop and wait, no following path can be displayed under the robot. This is also a reason why the full path outputting is not suitable for the simulation with Ozobots.

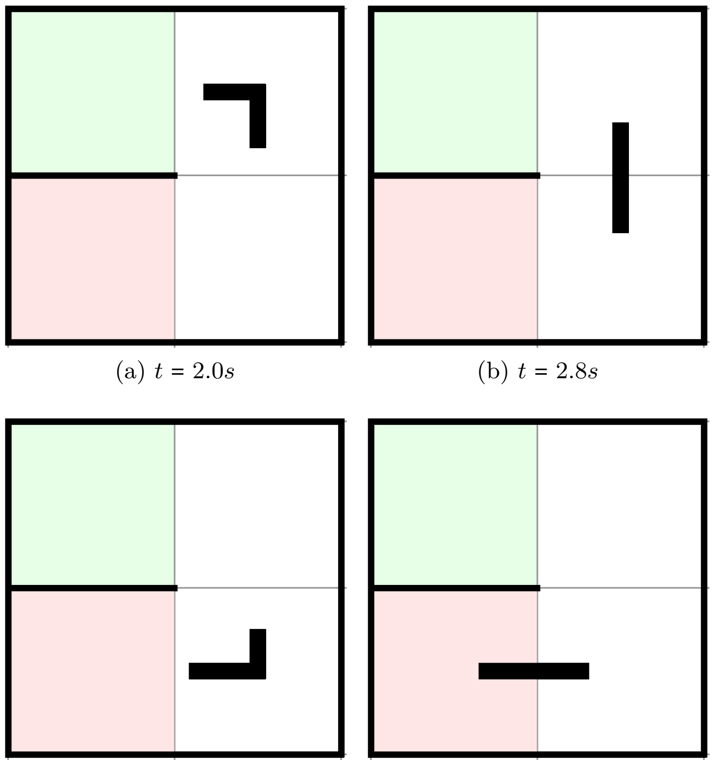

(a) $t = 2.0s$ (b) $t = 2.8s$ (c) $t = 3.7s$ (d) $t = 4.5s$

Figure 4.9: Path segments displayed in the map after t seconds from the start of the execution

4.2.2 ESO-OzoNav 1: Simple path animation

Both the *random direction problem* and *no wait problem* can be solved by displaying the agent paths partially. This version of the simulator outputs only a short segment of the path, that is continuously animated under the Ozobot as it moves across the map. The paths use sharp turns, and line ends are located in the middle of the map tiles, as in the previous version of the simulator. Though this time, during each screen update, a different segment of the line is displayed. The segment is selected based on the time that has passed from the beginning of the execution. This style of path drawing is shown in Figure 4.9.

The path segment can be defined by two components: head and tail. The *head* of the segment is the first point on the displayed line in the direction of movement. It is the theoretical position, where the agent should be located at a given time of execution if it were perfectly synchronized with the path. The *tail* of the segment is the line displayed from the head back in the direction

[21]Recall the position referencing in a grid from Section 1.1.2.2.

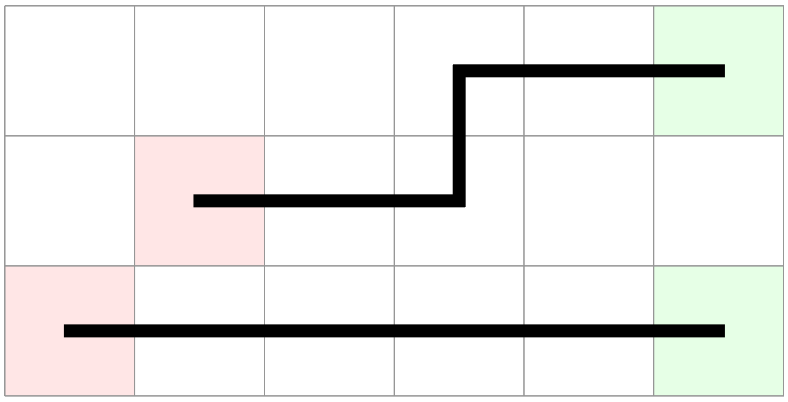


Figure 4.10: Two paths of the same length but one has turns

through where it was moving. The tail provides a limited length of following line for the Ozobot, should it fall behind the head of the path segment due to desynchronization. Ideally, the Ozobot should move in the middle of the displayed path segment. If the bot were located at the head of the segment, it would not be able to detect turns as they are not yet displayed at the head position.

Each time the screen is updated, both head and tail are updated, and thus the path segment slightly moves. The *wait* action can be simulated by not displaying the path segment. The head is not updated, and the tail starts to shrink towards it. When the agent should start to move again, the head is being moved forward, and the tail gradually grows to its full size.

The speed of the animation is given by a configurable time duration that takes the agent to transfer between two neighboring tiles. The segment length can also be configured. It is crucial to ensure that the line speed matches the speed of Ozobot. If the path segment moves too fast, the agent will lose it. On the other hand, if the line moves slowly, the robot would not detect turns and lose the line.

4.2.2.1 Desynchronization problem

Let there be two following lines of the same length, but one of them has turns, and the other is straight, as depicted in Figure 4.10. If Ozobots were to follow these lines simultaneously, one on each line, the one on the straight line would be at the end slightly sooner than the other bot. This is because the second robot has to perform extra rotations on the turns, and desynchronization is introduced into the execution. The desynchronization creates complications for this simulation approach as well.

If the path of an Ozobot is too complicated, it will fall behind the path segment animated under it and thus lose the line and fail the execution. Using curved turns instead of sharp ones would minimize the desynchronization and likely solve this problem.

4.2.2.2 Kiss problem

When an Ozobot loses the following line, usually because it is at the line end, it does not stop immediately. The robot travels a short distance before stopping completely. Therefore, when the path segment ends in the middle of a tile, the robot stops close to the border.

If two paths end in neighboring tiles and the robots finish on those tiles facing each other, they collide with each other while stopping. Even though this collision does not fail the whole path execution, it is a collision none the less. This could easily be solved by stopping the path segment before the middle of the tile. When the Ozobot encounters the end of the following line, it will stop nearly in the middle.

4.2.2.3 Wait on a turn problem

The issue with Ozobot's stopping described in the *kiss problem* also creates a different complication in the simulation. When an agent is required to stop and wait at a particular tile, the path segment also stops in the middle of it. As already mentioned, this way, the Ozobot is unable to stop in the middle of that tile. If the tile also contains a turn, it is a problem. When the waiting is over, the path segment will start its movement from the middle of the tile, but the Ozobot is not there to detect the line and follow it.

Unfortunately, this problem cannot be solved as the *kiss problem*. Even if the Ozobot would stop with its optical sensors precisely above the middle of the tile, it is missing the information that it needs to turn as the path shows up. Therefore, it will only move slightly forwards, losing the line immediately. Hypothetically turns displayed as curved lines could solve this problem. When the Ozobot stops on such a line, it should be sufficiently angled to continue when the line reappears under it.

4.2.2.4 No U-turn problem

Sometimes during the plan execution, agents need to turn around to continue in the direction from which they have come. An example of such a situation is a map in Figure 4.11, where agents need to swap their positions in a narrow corridor. The only way the agents can perform this swap is to have one of them to park in tile $(2, 0)$ and wait for the other agent to go through. Then the parked agent can get back into the corridor and continue to its goal destination. Sometimes during the waiting period, the bot has to turn around in order to continue its movement. However, in this version of the simulator, there is no way to tell the Ozobot to turn around, and therefore, it will not detect the line animating the rest of its path.

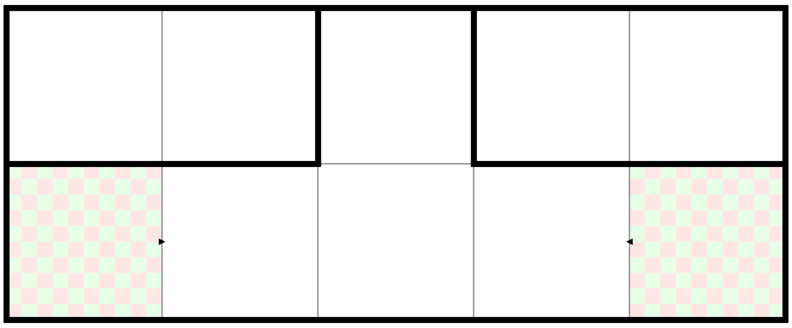
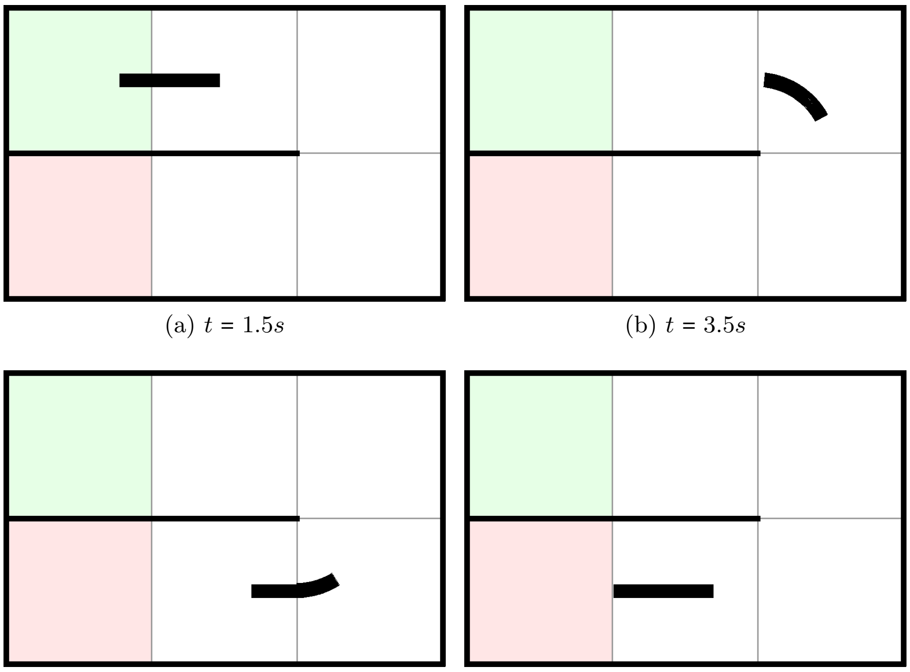


Figure 4.11: Narrow corridor map, where agents need to swap their positions



(a) $t = 1.5s$ (b) $t = 3.5s$ (c) $t = 6.0s$ (d) $t = 7.0s$

Figure 4.12: Path segments with curves displayed in the map after t seconds from the start of the execution

Fortunately, Ozobots can use Color Codes, which were explained in Section 2.3, and could be utilized in the simulation. Namely, the U-turn Color Code can be displayed for the Ozobot to solve this problem.

4.2.3 ESO-OzoNav 2: Path animation

This version of the simulator aims to solve the problems from the previous one. Turns in the paths are displayed as curves, as shown in Figure 4.12. If

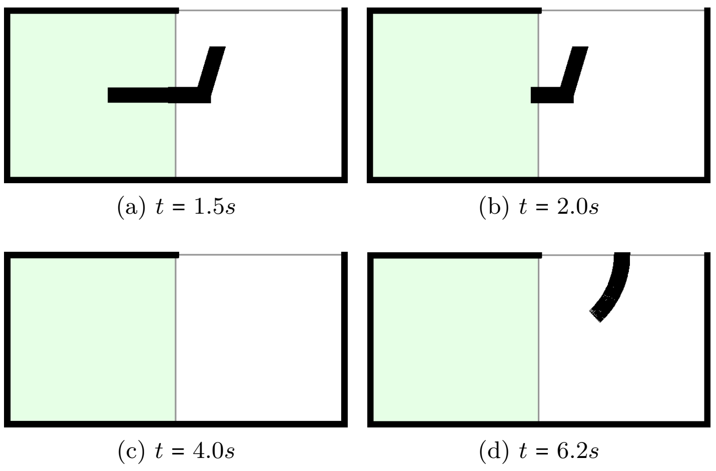

(a) $t = 1.5s$ (b) $t = 2.0s$ (c) $t = 4.0s$ (d) $t = 6.2s$

Figure 4.13: Path segments displayed in the map during waiting on curve after t seconds from the start of the execution

the robot needs to stop and wait on a tile, the path segment stops before the middle of that tile. This way, the bot stops its movement in the middle of the tile, which is more aesthetically pleasing but also solves the *kiss problem. Flashing Color Codes* can also be displayed in this simulator version. Namely, the U-turn Color Code is implemented, so the Ozobots can be turned around if needed.

Even though the *no U-turn problem* and the *kiss problem* were solved by the changes made in this simulator version, other problems remain. Moreover, the curved turns introduced new complication that needs to be solved.

4.2.3.1 Wait on a turn problem

The curved turns did not solve the *wait on a turn problem* as it was expected. The reason for this is that when Ozobot loses the curved line, it does not continue in its circular motion before stopping, but rather moves forwards in the facing direction as with the straight line. Therefore, the Ozobot moves away from the curve, and when it reappears on the tile again, the robot cannot detect it.

This problem, therefore, requires a different workaround. When the robot stops, its optical sensors have to be above the position, where the following line will reappear. It also has to be sufficiently rotated towards the next movement direction. To help Ozobots achieve this waiting position on a turn, the simulator displays a following guiding line for the bot, as shown in Figure 4.13. This guiding line forces the bot to stop in the correct position with the correct orientation for its next move. This workaround is also applied in the next simulator version.

(a) Ideal gradient colored path (b) Path segment divided into colored parts

Figure 4.14: Two versions of colored path segments

4.2.3.2 Curve following problem

Each discrete time step of the plan is assigned an execution time at which an agent should stand in the middle of the corresponding position tile. The path segment animation between two tiles assures the continuous transfer of the agent between two positions. The duration of this transfer is given by the configured speed of the line movement, and it is the same for every move. The curved turns have a shorter line length than the sharp turns, which means they have a slower animation speed. The speed of the Ozobot, on the other hand, does not change. Therefore, the Ozobot overtakes the path segment during the turn and loses the line. As it was explained in the *wait on a turn problem*, the bot will continue in the facing direction and lose the line completely. To solve this problem, the Ozobot has to follow the curve at a slower speed.

4.2.4 ESO-OzoNav 3: Colored paths

The previous version of the simulator module failed to mitigate the desynchronization of the path execution since the curved turns introduced a new issue. Therefore, this version of the module needs to solve the *curve following problem* and also the *desynchronization*.

These two problems can be solved with a variable speed of the robots. Generally, the robot should stay in the middle of the path segment displayed under it to follow the path correctly. Currently, the Ozobot is unable to do so due to the desynchronization and introduction of curved turns. When the bot falls behind the path segment, it needs to increase its following speed to keep up with the following line. On the other hand, when Ozobot gets close to the head of the path segment, speed needs to be decreased. It also has to follow the curved lines at a slower speed, as explained in the *curve following problem*. A modification of the Ozobot's behavior is required to change its following speed during the path execution. Because the bot needs to know when and how to change its movement speed, the environment outputs need to convey this information.

This could be achieved by displaying colored path segments and having the Ozobots change their movement speed based on the colors. Ideally, the color of the path segment would gradually change along the tail's length. For example, from blue color at the head position to red color at the end of the tail, as shown in Figure 4.14a. The Ozobot, moving on the following line,

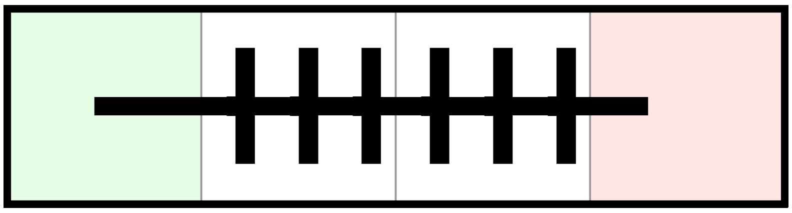


Figure 4.15: Full straight path with artificial intersections

would continually read the line color underneath its optical color sensor, and gradually change its movement speed. The speed would be increased towards the red color and decreased towards the blue color, forcing the robot to stay in the middle of the path segment.

However, this ideal path segment representation and agent behavior are not achievable due to the Ozobot's limitations. As mentioned in Section 2.2.1, Ozobot cannot read the exact values of color channels, but instead recognizes eight different colors. Therefore, a color gradient could not be fully exploited, and the path segment had to be divided into a few colored parts, as shown in Figure 4.14b. Another drawback is the limitation of Ozobot programs, explained in Section 2.5. The bot cannot read from its optical sensors while executing a different command. This limitation means that the robot needs to follow the line in small movements to perform the color reading between them. Unfortunately, the only line-following command that the OzoBlockly editor provides is `follow line to next intersection or line end`. This command blocks the program execution until the Ozobot finds an intersection or an end of the line, and so no color readings can be done. To solve this problem, artificial intersections have been added to the paths to interrupt the line-following command in order to perform color readings. A path with these artificial intersections is depicted in Figure 4.15.

The final version of the environment outputs and agent behavior is explained in more detail in the Section 4.3, collectively with the fundamental features included from previous versions.

4.3 ESO-OzoNav

This section describes the *final version* of the simulation prototype for Ozobot Evo robots. The *environment outputs* and *agent behavior* are explained in detail, including the changes from Section 4.2.4.

4.3.1 Environment outputs

The environment outputs are colored path segments displayed on the map, on which the Ozobots are moving. These colored path segments are depicted in Figure 4.16. Turns in paths are drawn as curves to make the turning of

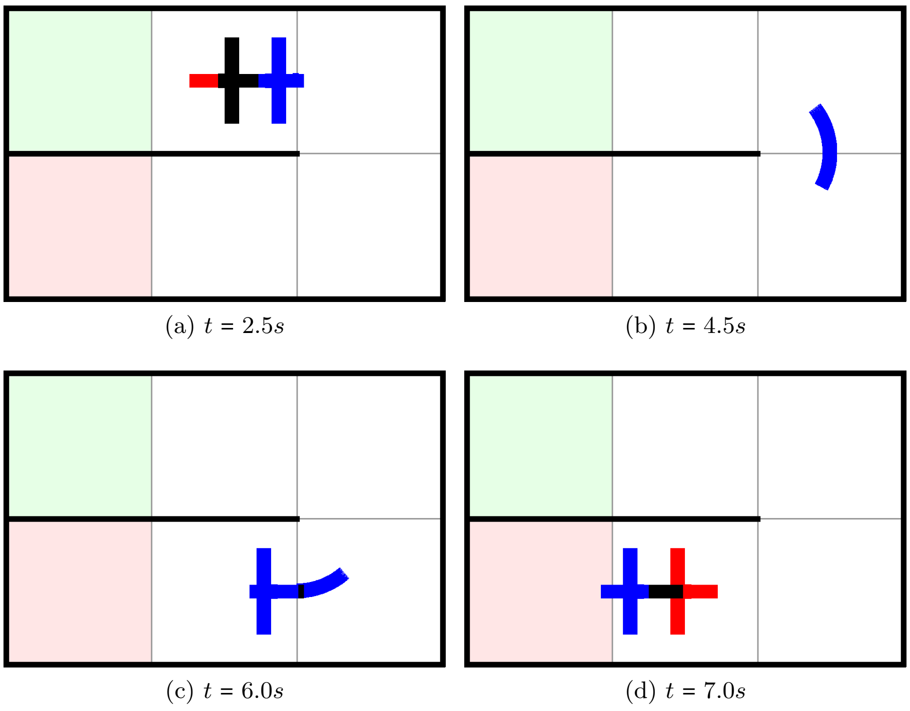

(a) $t = 2.5s$ (b) $t = 4.5s$ (c) $t = 6.0s$ (d) $t = 7.0s$

Figure 4.16: Colored path segments displayed in the map after t seconds from the start of the execution

the robots smooth. The path segments are divided into three colored parts representing three different movement speeds: slow, normal, and fast. The *slow speed* is represented by the *blue color* and corresponds to the speed of curve animation on a turn. The blue part of the line is located at the head of the path segment. Also, all curves are colored blue to indicate the slow speed of the agent movement. The *normal speed* is represented by the *black color*. The robot's speed on a black line matches the speed of the straight line animation. The black part of the line is located in the middle of the tail. Lastly, the *fast speed* is indicated by the *red color*. The red part of the line is located at the end of the tail and informs the agent to speed up in order to keep up with the path segment.

Artificial intersections are indicated on the paths, where the Ozobots perform color readings and change their movement speed accordingly. These intersection indicators have to have a sufficient length, cannot be displayed on the curved lines, and there has to be adequate separation between them. Otherwise, the Ozobot would fail to detect them during the movement. Keeping this in mind, only three intersection indicators are placed in a tile, unless the agent should wait at the position, or there is a turn. The indicators are shifted towards the direction of the agent's movement. The reason for

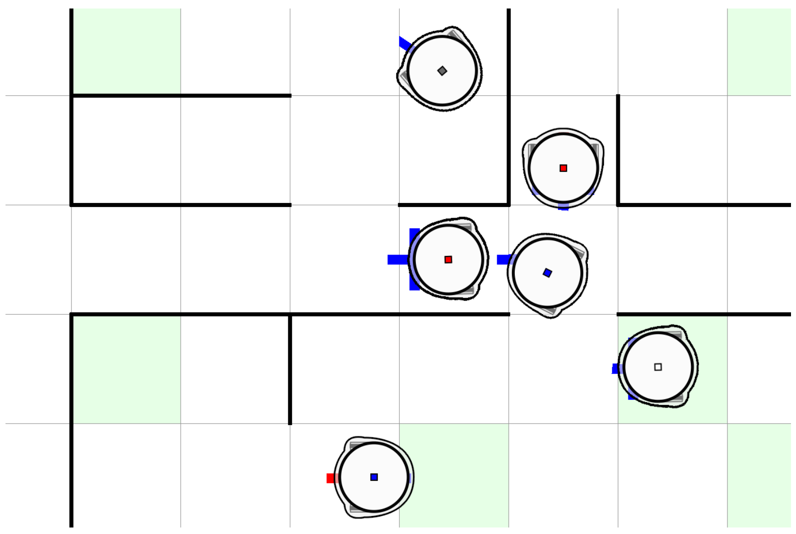


Figure 4.17: Illustration of Ozobots following colored path segments on a map

this is, that if Ozobot enters the tile from a curve, it needs to straighten its movement before encountering an intersection to avoid unwanted behavior. Another benefit is that if a curve is following after the current tile, Ozobot's speed is updated closer to the turn. The color of the intersection indicator[22] changes according to the path segment color at its location. The only exception is the intersection before a turn, which will always remain blue to slow the Ozobot down before the curve.

If the agent has to stop at a specific position, the path segment stops before the middle of the corresponding tile. The Ozobot then stops in the middle of the tile. The exception is when the robot needs to wait on a turn. In that situation, the following guiding line is displayed to bring the Ozobot into a correct orientation. This ensures that the bot can continue on its path when the path segment reappears underneath it.

During the Ozobot's movement on the colored path segment, it can still read the *Flashing Color Codes*. If the robot should wait at a tile, and the next movement direction is the same as the direction from where it entered, the *U-turn Color Code* is displayed upon the robot's arrival to the tile.

An illustration of Ozobots navigating on the path segments is shown in Figure 4.17.

4.3.2 Agent behavior

Ozobots need to follow the lines outputted by the environment, but also change their movement speed according to the surface color of the lines. This modified

[22]And also a small neighborhood around the intersection.

Algorithm 4: Ozobot behavior program

```
1  while true do
2      set line-following speed: getSpeedFromLineColor() mm/s
3      if there is way straight then
4          pick direction: straight
5      else
6          stop motors
7      follow line to next intersection or line end
```

Algorithm 5: Function reading line color and returning speed

```
1  Function getSpeedFromLineColor()
2      color ← get surface color
3      if color = surface color red then
4          speed ← 37
5      else if color = surface color black then
6          speed ← 30
7      else if color = surface color blue then
8          speed ← 23
9      else
10         speed ← 21
11     return speed
```

behavior can be written as an Ozobot program in the OzoBlockly editor and loaded to all robots simultaneously.

The program can be found in Algorithm 4. The main loop runs until the program is manually terminated. As can be seen on line 7 of the code, the robot is moving on the path segments between the artificial intersections or until it loses the following line. When an intersection or a line end is encountered, the line-following speed is updated on line 2. Because the Ozobot naturally wants to choose a random direction at any intersection, line 4 makes it always go straight. However, if the movement was interrupted and there is no line under the robot, line 6 stops it from moving.

The function `getSpeedFromLineColor` from line 2 is shown in Algorithm 5. First, the function reads the surface color from the Ozobot's optical color sensor on line 2. On lines 3–10, the speed is chosen according to the color, and on line 11, it is returned. The speed on the black colored path corresponds to the straight path animation speed, and the blue line's speed was chosen to match the curve animation speed. Line 10 is executed, when the surface color is white, meaning there is no line under the robot. This situation usually

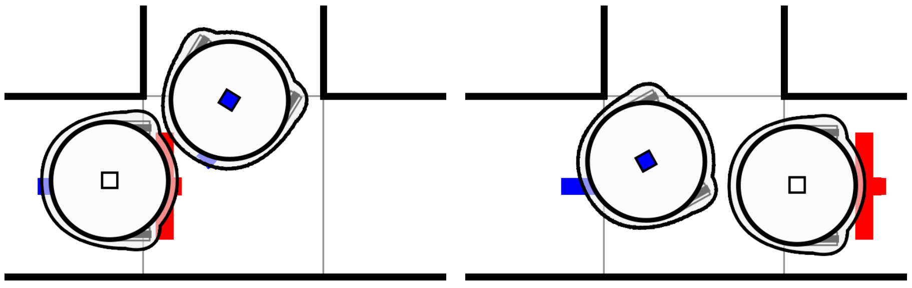

(a) Agent getting behind another agent (b) Agent getting in front of another agent

Figure 4.18: Two situations where the agents can touch during the path execution

happens when the Ozobot is waiting on a path segment to appear under it. The speed is therefore set to a lower value so the robot would not lose the line immediately and the path segment could take the lead.

4.3.3 Limitations

During the development of the final prototype version, a few limitations of the system were discovered. The most impactful is the occurrence of *agent oscillation between two positions* in a plan. The oscillation is a situation when in one time step, an agent enters a position, and in the following time step, it returns to the previous one. Even though this maneuver is valid in the MAPF solution, for robots like Ozobot, it is impossible to perform a U-turn in such a short time. In the plans from the boOX program, these oscillations occur because of the *sum of costs* objective function. In the sum of costs, the cost of a path is increased even during the *wait* actions. Therefore agents can freely move back and forth between two positions without any cost penalization. Utilization of a *fuel* objective function[23] would solve this problem. For the use in the simulator prototype, these agent oscillations had to be manually removed from the plans before simulation.

Another drawback of the prototype is minor agent collisions. These collisions can occur when an agent moving on a curve is trying to get right in front of or behind another agent. This situation is depicted in Figure 4.18. The agents usually only touch or scratch against each other, and the collision does not impact the plan execution. This problem could be solved by using a bigger tile size and configuring the animation and Ozobot speeds accordingly. This issue does not concern the *1-robust* plans, because of the extra spacing between the agents.

The optical color sensors of Ozobot Evo can be very dependant on the surrounding *light conditions*. With even a slight difference in the light conditions

[23]Where the cost is increased only when an agent moves to a different position.

that can occur during the day, the accuracy of Ozobot's sensors can change considerably. The sensors need to be calibrated occasionally, or else the bots may fail to read the colors or follow curves correctly. This inaccuracy often leads to loss of the following line. It seems that this problem cannot be fully removed, but it can be mitigated with the sensor calibrations. The robots also seem to have a problem with *shadows* and lighting coming from only one direction. When the Ozobots are calibrated with a direct light falling at them, introducing shadows makes them believe that they are detecting a following line. A *directional light* source that is not directly above the map can cause that some parts of the map are darker than others. Ozobot will, therefore, behave differently on these parts of the map.

CHAPTER 5

Experiments

In this chapter, the final version of the prototype is tested and evaluated. In the first section, the maps created for the testing are shown and discussed. The critical parts for each plan are also addressed. In the second section, the results of the experiments are summarized.

Each of the problem instances, represented by a map, is solved with both the *classical* and *1-robust* MAPF solver[24]. If there are any *agent oscillation*[25] present in any of the plans, they are manually removed, as they would result in almost certain failure of the execution. Each of the plans is executed on the prototype 32 times.

5.1 Maps

For the experiments, six maps were created. Some of the maps aim to test a specific feature or a critical maneuver of the system, others provide a balanced scenario for the execution. All of the maps are listed in Table 5.1 and the editor previews of these maps are shown in this section. For each map, its width, height, and the number of agents are provided in the table.

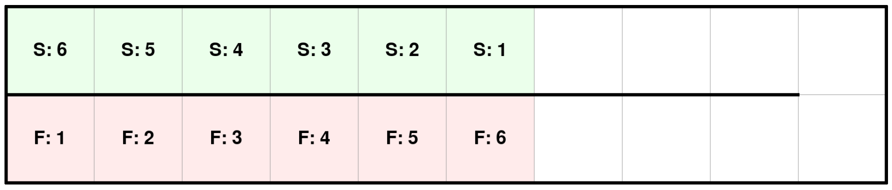


Figure 5.1: Experiment map: The snake

[24]For some maps, these two plans are identical, and only one of them is experimented on.
[25]Agent oscillations are described in Section 4.3.3.

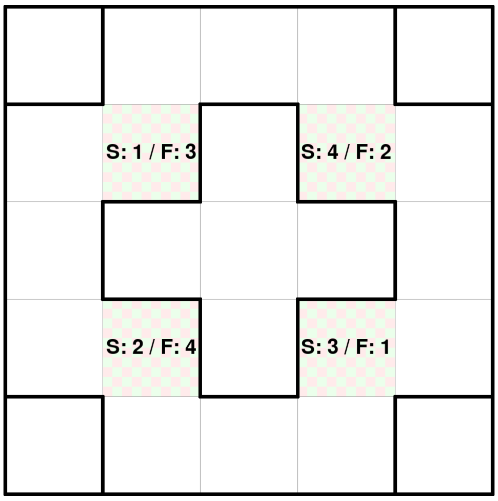


Figure 5.2: Experiment map: The rotation

In Table 5.2, all plans for these maps are listed. Both *classic* and the *1-robust* plan[26] for each map were constructed, with the exception of the *rotation* map, where both plans would be identical. For each plan, some maneuvers that could be problematic for the robots are counted. Namely, the number of turns without waiting, the number of *Color Codes* displayed (CC), and the number of *wait on a turn* positions (WoT).

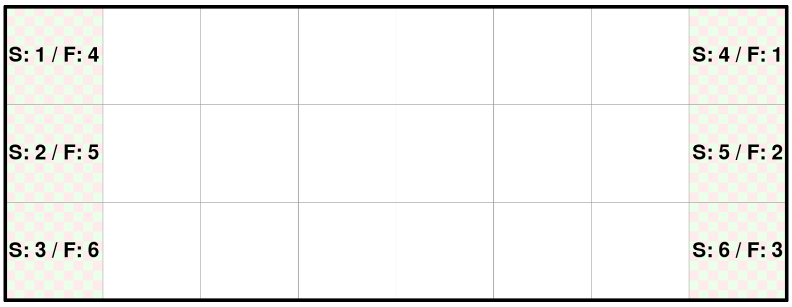


Figure 5.3: Experiment map: The swap

5.2 Evaluation

Every plan was executed 32 times with the implemented prototype. The execution is marked as *successful* if all Ozobots reach their goal positions. If at least one loses the following line and does not reach the goal, the execution is marked as a *failure*. The results of the experiments are summarized in

[26]The name of the 1-robust plan has a suffix "_robust".

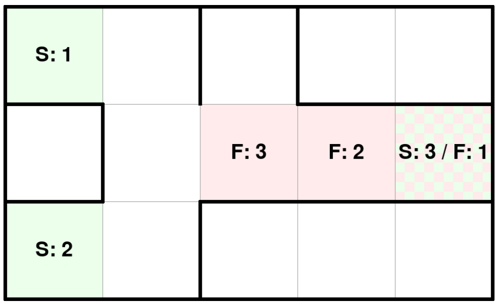


Figure 5.4: Experiment map: The ordering

Table 5.1: Maps created for experiments

Map name	Width	Height	Agents	Image of the map
The snake	10	2	6	In Figure 5.1
The rotation	5	5	4	In Figure 5.2
The swap	8	3	6	In Figure 5.3
The ordering	5	3	3	In Figure 5.4
The evacuation	10	5	6	In Figure 5.5
The roundabout	9	5	6	In Figure 5.6

Table 5.3. During the testing, five different reasons for execution failure were recorded. The occurrence of these failures was counted and is presented in the table of results.

As mentioned in Section 4.3.3, Ozobots can sometimes touch during the executions. Sometimes, however, a *severe collision* (SC) can occur, where the robots push against each other or lift their wheels from the surface. This

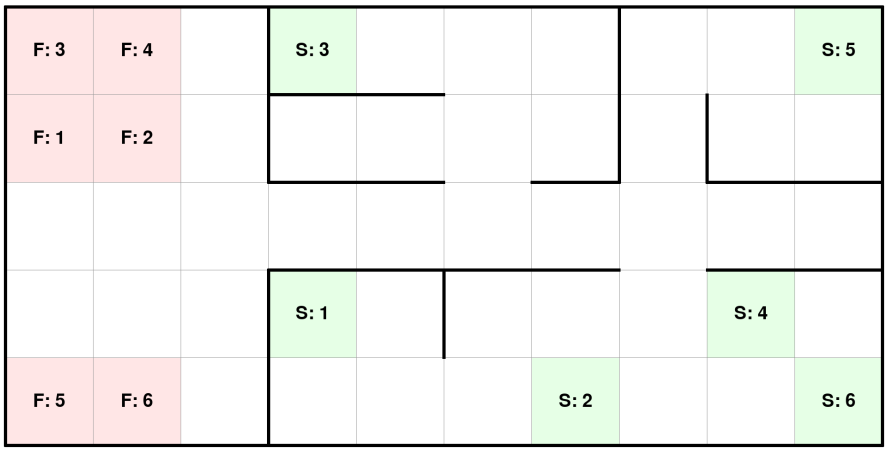


Figure 5.5: Experiment map: The evacuation

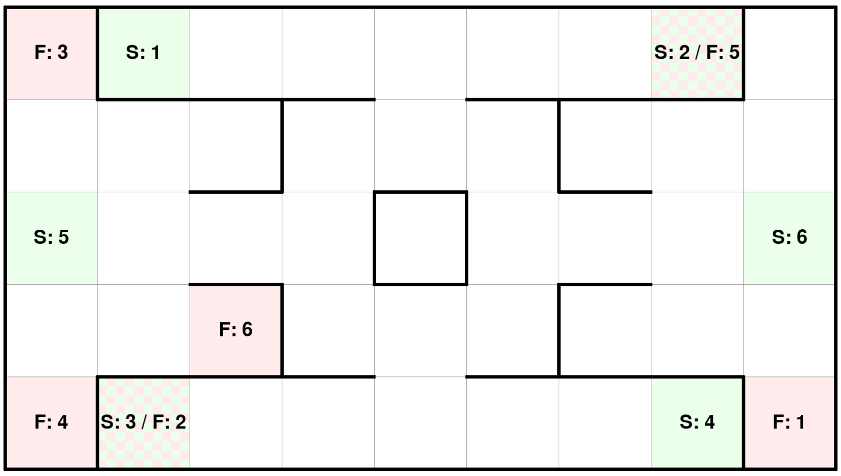


Figure 5.6: Experiment map: The roundabout

Table 5.2: Plans executed with the prototype

Plan name	Map	Turns	CC	WoT
snake	The snake	12	0	0
snake_robust	The snake	12	0	0
rotation	The rotation	20	0	0
swap	The swap	10	0	1
swap_robust	The swap	9	0	3
ordering	The ordering	5	1	0
ordering_robust	The ordering	3	1	2
evacuation	The evacuation	26	0	1
evacuation_robust	The evacuation	25	0	1
roundabout	The roundabout	33	0	0
roundabout_robust	The roundabout	30	0	1

collision results in an execution failure because the robots lose their following lines and are unable to continue. This problem does not occur very often and is non-existent in the 1-robust plans.

On the other hand, more frequent was the failure due to *missed intersection* (MI). Sometimes, an Ozobot failed to update its speed at an intersection because it was not detected by its sensors. This almost always resulted in an execution failure if the agent did not slow down before a turn. The first experiments showed that the success rate of an Ozobot to detect an intersection correctly fluctuates with changing light conditions and performing calibrations. However, results from the plan executions on the *roundabout* map suggest that the complexity of the paths might also affect the frequency of these mistakes.

Table 5.3: Results of the experiments

Plan name	Success	Fail	SC	MI	CC	WoT	EC
snake	32	0	0	0	0	0	0
snake_robust	32	0	0	0	0	0	0
rotation	29	3	0	3	0	0	0
swap	30	2	0	1	0	1	0
swap_robust	30	2	1	0	1	0	0
ordering	30	2	1	0	1	0	0
ordering_robust	24	8	0	0	4	3	1
evacuation	28	4	2	0	0	2	0
evacuation_robust	30	2	0	2	0	0	0
roundabout	23	9	0	9	0	0	0
roundabout_robust	19	13	0	8	0	3	2

The maneuvers as *Color Code* execution (CC) and *wait on a turn* (WoT) also caused a few execution failures. To fail to execute the *U-turn*, the bot can either arrive at the tile too soon or too late. If it arrives too soon, it fails to read the whole code and does not rotate at all. If it arrives too late, it reads the code twice and makes two rotations ending up in the original orientation. Both of these scenarios ensure the failure of plan execution. Performing the *wait on a turn* maneuver, the robot sometimes makes a U-turn at the end of the following guiding line. This behavior is most likely triggered when Ozobot loses the following line without detecting a line end.

The last problem noticed during the experiments was Ozobot failing to *exit a curve* (EC). Even though this was a rare occurrence, sometimes, when the bot passed through a curved turn, it was unable to detect the following line correctly and lost the path.

5.3 Summary of results

Overall, these experiments demonstrated the ability of the *ESO-Nav* to be used for simulation of MAPF solutions that are even constructed on top of the classical abstract model. The prototype implemented with Ozobot Evo robots also shown, that this ability to perform the simulation correctly is highly dependant on the physical agents and their ability to read and respond to the environment outputs. As for the Ozobots, their main weakness is the variable accuracy of optical sensors with different light conditions. To improve this prototype, the *wait on a turn* maneuver and *Color Code* displaying should be refined, and adequate and stable light conditions would need to be ensured.

Conclusion

This work has introduced a novel approach to simulation of centralized MAPF algorithms called *ESO-Nav* that was tested on a group of Ozobot Evo robots. The simulation prototype created in this work showed that even centralized MAPF algorithms that use classical discrete model could be simulated on physical robots.

In Chapter 1, the reader was familiarized with the theoretical background and previous MAPF research that has been done. Chapter 2 provided an extensive overview of Ozobot Evo, including its hardware, capabilities, and limitations. Chapter 3 discussed previous work concerning MAPF simulation on real robots that used Ozobots and proposed the novel approach. In Chapter 4, a prototype of a MAPF simulator for Ozobot Evo was presented. This chapter also explained various issues that were discovered and solved during the prototype implementation.

The prototype showed that the *ESO-Nav* approach is capable of simulation of various MAPF scenarios with physical agents. It could be used for visualization in research or academics, as well as in real-world applications like intelligent evacuation system or indoor transporter navigation. The prototype itself can be used for educational MAPF demonstrations, and it could be extended for use in other research.

Review of the thesis aims

All of the goals that were set in the introduction of this work were also met. These goals were the following:

- Explore Ozobots and their capabilities.
- Explain the selected approach and compare it to the already existing work.
- Create a simulation prototype.

The first goal was completed in Chapter 2, where the Ozobot Evo was analyzed. The findings showed that Ozobots are very simple and limited bots. Although they are programmable to a large extent, the programmability provided with a visual editor like OzoBlockly is very impractical. At the end of the chapter, alternative robots were also reviewed. Chapter 3 completed the second goal by defining the *ESO-Nav* approach. This approach also got around the impracticality of Ozobot programming by utilizing their reactive behavior. The novel approach was then compared to the previous approach of MAPF simulation on Ozobots. In Chapter 4, the completion of the last goal was described. The simulator prototype faced the challenge of combining a discrete solver with robots that have continuous movement. This combination posed several problems that needed to be solved during the implementation process. The prototype was then tested on experiments from Chapter 5.

Future work

There are several ways how the prototype from Chapter 4 can be improved or extended. Also, *ESO-Nav* provides new opportunities for research. Here are some examples:

- A video camera could be added to the system for detecting conflicts or incorrect path execution. With such monitoring, agents could support imperfect behavior that would lead to mistakes, and the pats would require replanning. Collision prediction would also be possible.
- Other maps than grid-based maps could be used, and agents could move on continuously curved paths. These features would require the use of a continuous solver like CCBS.
- A different line-following robot could be used instead of the Ozobot.
- The impact of different agent behaviors on path execution could be studied.

Bibliography

1. DIESTEL, Reinhard. *Graph theory.* 5th ed. Berlin: Springer, 2017. Graduate texts in mathematics. ISBN 978-3-662-53621-6.
2. BUROSCH, Gustav; LABORDE, Jean-Marie. Characterization of grid graphs. *Discrete mathematics.* 1991, vol. 87, no. 1, pp. 85–88. Available from DOI: `10.1016/0012-365X(91)90074-C`.
3. CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009. ISBN 978-0-262-03384-8.
4. HART, Peter E.; NILSSON, Nils J.; RAPHAEL, Bertram. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics.* 1968, vol. 4, no. 2, pp. 100–107. Available from DOI: `10.1109/TSSC.1968.300136`.
5. POHL, Ira. Heuristic Search Viewed as Path Finding in a Graph. *Artificial Intelligence.* 1970, vol. 1, no. 3, pp. 193–204. Available from DOI: `10.1016/0004-3702(70)90007-X`.
6. BOTEA, Adi; MÜLLER, Martin; SCHAEFFER, Jonathan. Near Optimal Hierarchical Path-Finding. *Journal of Game Development.* 2004, vol. 1, no. 1, pp. 1–30.
7. YAP, Peter. Grid-Based Path-Finding. In: *Conference of the Canadian Society for Computational Studies of Intelligence.* Springer, 2002, vol. 2338, pp. 44–55. Available from DOI: `10.1007/3-540-47922-8_4`.
8. ARIKAN, Okan; CHENNEY, Stephen; FORSYTH, David A. Efficient multi-agent path planning. In: *Computer Animation and Simulation 2001.* Springer, 2001, pp. 151–162. Available from DOI: `10.1007/978-3-7091-6240-8_14`.

9. SILVER, David. Cooperative Pathfinding. In: *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference*. Marina del Rey, California: AAAI Press, 2005, vol. 1, pp. 117–122. ISBN 978-1-57735-235-8.

10. WANG, Ko-Hsin Cindy; BOTEA, Adi. Fast and Memory-Efficient Multi-Agent Pathfinding. In: *Proceedings of the 18th International Conference on Automated Planning and Scheduling*. AAAI, 2008, pp. 380–387. ISBN 978-1-57735-386-7.

11. SURYNEK, Pavel. *Abstract path planning for multiple robots: A theoretical study* [online]. 2010 [visited on 2020-05-16]. Available from: `https://iti.mff.cuni.cz/series/2010/503.pdf`. Technical report. Faculty of Mathematics and Physics, Charles University.

12. SURYNEK, Pavel. Solving Abstract Cooperative Path-Finding in Densely Populated Environments. *Computational Intelligence*. 2014, vol. 30, no. 2, pp. 402–450. Available from DOI: `10.1111/coin.12002`.

13. KORNHAUSER, Daniel; MILLER, Gary L.; SPIRAKIS, Paul G. Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications. In: *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984*. IEEE Computer Society, 1984, pp. 241–250. Available from DOI: `10.1109/SFCS.1984.715921`.

14. WILSON, Richard M. Graph puzzles, homotopy, and the alternating group. *Journal of Combinatorial Theory, Series B*. 1974, vol. 16, no. 1, pp. 86–96. Available from DOI: `10.1016/0095-8956(74)90098-7`.

15. RATNER, Daniel; WARMUTH, Manfred K. Finding a Shortest Solution for the N × N Extension of the 15-PUZZLE Is Intractable. In: *Proceedings of the 5th National Conference on Artificial Intelligence*. Morgan Kaufmann, 1986, pp. 168–172. ISBN 978-0-262-51054-7.

16. RYAN, Malcolm R. K. Graph Decomposition for Efficient Multi-Robot Path Planning. In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence* [online]. 2007, pp. 2003–2008 [visited on 2020-05-16]. Available from: `http://ijcai.org/Proceedings/07/Papers/323.pdf`.

17. RYAN, Malcolm R. K. Exploiting Subgraph Structure in Multi-Robot Path Planning. *Journal of Artificial Intelligence Research*. 2008, vol. 31, pp. 497–542. Available from DOI: `10.1613/jair.2408`.

18. DRESNER, Kurt M.; STONE, Peter. A Multiagent Approach to Autonomous Intersection Management. *Journal of Artificial Intelligence Research*. 2008, vol. 31, pp. 591–656. Available from DOI: `10.1613/jair.2502`.

19. STANDLEY, Trevor Scott. Finding Optimal Solutions to Cooperative Pathfinding Problems. In: *Proceedings of the 24th AAAI Conference on Artificial Intelligence*. AAAI Press, 2010. ISBN 978-1-57735-463-5.
20. ATZMON, Dor; STERN, Roni; FELNER, Ariel; WAGNER, Glenn; BARTÁK, Roman; ZHOU, Neng-Fa. Robust Multi-Agent Path Finding. In: *Proceedings of the 11th International Symposium on Combinatorial Search*. AAAI Press, 2018, pp. 2–9. ISBN 978-1-57735-802-2.
21. SHARON, Guni; STERN, Roni; GOLDENBERG, Meir; FELNER, Ariel. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*. 2013, vol. 195, pp. 470–495. Available from DOI: `10.1016/j.artint.2012.11.006`.
22. SHARON, Guni; STERN, Roni; FELNER, Ariel; STURTEVANT, Nathan R. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*. 2015, vol. 219, pp. 40–66. Available from DOI: `10.1016/j.artint.2014.11.006`.
23. SURYNEK, Pavel. Towards Optimal Cooperative Path Planning in Hard Setups through Satisfiability Solving. In: *Proceedings of the 12th Pacific Rim International Conference on Artificial Intelligence*. Springer, 2012, vol. 7458, pp. 564–576. Lecture Notes in Computer Science. Available from DOI: `10.1007/978-3-642-32695-0_50`.
24. SURYNEK, Pavel; FELNER, Ariel; STERN, Roni; BOYARSKI, Eli. Efficient SAT Approach to Multi-Agent Path Finding Under the Sum of Costs Objective. In: *22nd European Conference on Artificial Intelligence*. IOS Press, 2016, vol. 285, pp. 810–818. Frontiers in Artificial Intelligence and Applications. Available from DOI: `10.3233/978-1-61499-672-9-810`.
25. MA, Hang; TOVEY, Craig A.; SHARON, Guni; KUMAR, T. K. Satish; KOENIG, Sven. Multi-Agent Path Finding with Payload Transfers and the Package-Exchange Robot-Routing Problem. In: *Proceedings of the 30th AAAI Conference on Artificial Intelligence*. AAAI Press, 2016, pp. 3166–3173. ISBN 978-1-57735-760-5.
26. YU, Jingjin; LAVALLE, Steven M. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. In: *Proceedings of the 27th AAAI Conference on Artificial Intelligence*. AAAI Press, 2013. ISBN 978-1-57735-615-8.
27. BARER, Max; SHARON, Guni; STERN, Roni; FELNER, Ariel. Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem. In: *21st European Conference on Artificial Intelligence*. IOS Press, 2014, vol. 263, pp. 961–962. Frontiers in Artificial Intelligence and Applications. Available from DOI: `10.3233/978-1-61499-419-0-961`.

28. SURYNEK, Pavel; FELNER, Ariel; STERN, Roni; BOYARSKI, Eli. Sub-Optimal SAT-Based Approach to Multi-Agent Path-Finding Problem. In: *Proceedings of the 11th International Symposium on Combinatorial Search*. AAAI Press, 2018, pp. 90–105. ISBN 978-1-57735-802-2.

29. BOYARSKI, Eli; FELNER, Ariel; STERN, Roni; SHARON, Guni; BETZALEL, Oded; TOLPIN, David; SHIMONY, Solomon Eyal. ICBS: The Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding. In: *Proceedings of the 8th Annual Symposium on Combinatorial Search*. AAAI Press, 2015, pp. 223–225. ISBN 978-1-57735-732-2.

30. ERDEM, Esra; KISA, Doga Gizem; ÖZTOK, Umut; SCHÜLLER, Peter. A General Formal Framework for Pathfinding Problems with Multiple Agents. In: *Proceedings of the 27th AAAI Conference on Artificial Intelligence*. AAAI Press, 2013. ISBN 978-1-57735-615-8.

31. YU, Jingjin; LAVALLE, Steven M. Planning optimal paths for multiple robots on graphs. In: *IEEE International Conference on Robotics and Automation*. IEEE, 2013, pp. 3612–3617. Available from DOI: `10.1109/ICRA.2013.6631084`.

32. LUNA, Ryan; BEKRIS, Kostas E. Push and Swap: Fast Cooperative Path-Finding with Completeness Guarantees. In: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*. IJCAI/AAAI, 2011, pp. 294–300. Available from DOI: `10.5591/978-1-57735-516-8/IJCAI11-059`.

33. SURYNEK, Pavel. A novel approach to path planning for multiple robots in bi-connected graphs. In: *IEEE International Conference on Robotics and Automation*. IEEE, 2009, pp. 3613–3619. Available from DOI: `10.1109/ROBOT.2009.5152326`.

34. WILDE, Boris de; MORS, Adriaan ter; WITTEVEEN, Cees. Push and Rotate: a Complete Multi-agent Pathfinding Algorithm. *Journal of Artificial Intelligence Research*. 2014, vol. 51, pp. 443–492. Available from DOI: `10.1613/jair.4447`.

35. WANG, Ko-Hsin Cindy; BOTEA, Adi. MAPP: a Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees. *Journal of Artificial Intelligence Research*. 2011, vol. 42, pp. 55–90. Available from DOI: `10.1613/jair.3370`.

36. ANDREYCHUK, Anton; YAKOVLEV, Konstantin S.; ATZMON, Dor; STERN, Roni. Multi-Agent Pathfinding with Continuous Time. In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. ijcai.org, 2019, pp. 39–45. Available from DOI: `10.24963/ijcai.2019/6`.

37. JANSEN, M. Renee; STURTEVANT, Nathan R. A new approach to cooperative pathfinding. In: *7th International Joint Conference on Autonomous Agents and Multiagent Systems*. IFAAMAS, 2008, pp. 1401–1404. ISBN 978-0-9817381-2-3.

38. IVANOVÁ, Marika; SURYNEK, Pavel. Adversarial Cooperative Path-Finding: A First View. In: *Late-Breaking Developments in the Field of Artificial Intelligence*. AAAI, 2013, vol. WS-13-17. AAAI Workshops. ISBN 978-1-57735-628-8.

39. IVANOVÁ, Marika; SURYNEK, Pavel. Area Protection in Adversarial Path-Finding Scenarios with Multiple Mobile Agents on Graphs: a theoretical and experimental study of target-allocation strategies for defense coordination. *Computing Research Repository* [online]. 2017 [visited on 2020-05-16]. Available from arXiv: `1708.07285`.

40. SELVEK, Róbert; SURYNEK, Pavel. Engineering Smart Behavior in Evacuation Planning using Local Cooperative Path Finding Algorithms and Agent-based Simulations. In: *Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*. ScitePress, 2019, pp. 137–143. Available from DOI: `10.5220/0008071501370143`.

41. VELOSO, Manuela M.; BISWAS, Joydeep; COLTIN, Brian; ROSENTHAL, Stephanie. CoBots: Robust Symbiotic Autonomous Mobile Service Robots. In: *Proceedings of the 24th International Joint Conference on Artificial Intelligence*. AAAI Press, 2015, p. 4423. ISBN 978-1-57735-738-4.

42. WURMAN, Peter R.; D'ANDREA, Raffaello; MOUNTZ, Mick. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI Magazine* [online]. 2008, vol. 29, no. 1, pp. 9–20 [visited on 2020-05-16]. Available from: `http://www.aaai.org/ojs/index.php/aimagazine/article/view/2082`.

43. MORRIS, Robert; PASAREANU, Corina S.; LUCKOW, Kasper Søe; MALIK, Waqar; MA, Hang; KUMAR, T. K. Satish; KOENIG, Sven. Planning, Scheduling and Monitoring for Airport Surface Operations. In: *Planning for Hybrid Systems, Papers from the 2016 AAAI Workshop*. AAAI Press, 2016, vol. WS-16-12. AAAI Workshops. ISBN 978-1-57735-759-9.

44. BOTEA, Adi; BOUZY, Bruno; BURO, Michael; BAUCKHAGE, Christian; NAU, Dana S. Pathfinding in Games. In: *Artificial and Computational Intelligence in Games*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013, vol. 6, pp. 21–31. Dagstuhl Follow-Ups. Available from DOI: `10.4230/DFU.Vol6.12191.21`.

45. MA, Hang; YANG, Jingxing; COHEN, Liron; KUMAR, T. K. Satish; KOENIG, Sven. Feasibility Study: Moving Non-Homogeneous Teams in Congested Video Game Environments. *Computing Research Repository* [online]. 2017, vol. abs/1710.01447 [visited on 2020-05-16]. Available from arXiv: `1710.01447`.

46. LI, Jiaoyang; SURYNEK, Pavel; FELNER, Ariel; MA, Hang; KUMAR, T. K. Satish; KOENIG, Sven. Multi-Agent Path Finding for Large Agents. In: *The 33rd AAAI Conference on Artificial Intelligence, The 31st Innovative Applications of Artificial Intelligence Conference, The 9th AAAI Symposium on Educational Advances in Artificial Intelligence*. AAAI Press, 2019, pp. 7627–7634. Available from DOI: `10.1609/aaai.v33i01.33017627`.

47. BARTÁK, Roman; SVANCARA, Jirí; SKOPKOVÁ, Vera; NOHEJL, David. Multi-agent Path Finding on Real Robots: First Experience with Ozobots. In: *Proceedings of the 16th Ibero-American Conference on AI*. Springer, 2018, vol. 11238, pp. 290–301. Lecture Notes in Computer Science. Available from DOI: `10.1007/978-3-030-03928-8_24`.

48. EVOLLVE, Inc. *Ozobot* [online]. 2020 [visited on 2020-03-23]. Available from: `https://ozobot.com/`.

49. EVOLLVE, Inc. *Ozobot Classroom* [online]. 2020 [visited on 2020-03-23]. Available from: `https://classroom.ozobot.com/`.

50. EVOLLVE, Inc. *Image of Ozobot Evo* [online] [visited on 2020-03-25]. Available from: `https://ozobot.com/`.

51. HUNSAKER, Enoch. Ozobot Bit: A Guide for Parents and Educators [Online Document]. 2018 [visited on 2020-03-23]. Available from: `https://scholarsarchive.byu.edu/cgi/viewcontent.cgi?filename=6&article=1007&context=ipt_projects&type=additional`.

52. EVOLLVE, Inc. *Ozobot sensor layout images* [online] [visited on 2020-03-26]. Available from: `https://files.ozobot.com/classroom/2019-Educator-Guide.pdf`.

53. EVOLLVE, Inc. *OzoBlockly* [online]. 2018 [visited on 2020-03-23]. Available from: `https://ozoblockly.com/`.

54. EVOLLVE, Inc. *Ozobot Color Codes* [online] [visited on 2020-04-06]. Available from: `https://files.ozobot.com/stem-education/ozobot-color-codes.pdf`.

55. MONE, Gregory. *MIT Technology Review: Robo Swarm* [online]. 2016 [visited on 2020-04-10]. Available from: `https://www.technologyreview.com/2016/08/23/158011/robo-swarm/`.

56. TRANSPORT AND TELECOMMUNICATION INSTITUTE. *Great news from Computer Science and Telecommunication Faculty: Robots are coming, they are here* [online] [visited on 2020-04-10]. Available from: `http://www.tsi.lv/en/content/great-news-computer-science-and-telecommunication-faculty-robots-are-coming-they-are-here`.

57. RAHAL TECHNOLOGY LIMITED. *e-puck2* [online] [visited on 2020-04-10]. Available from: `https://rahalco.co.uk/portfolio/e-puck-2/`.

58. BARTÁK, Roman; ZHOU, Neng-Fa; STERN, Roni; BOYARSKI, Eli; SURYNEK, Pavel. Modeling and Solving the Multi-agent Pathfinding Problem in Picat. In: *29th IEEE International Conference on Tools with Artificial Intelligence*. IEEE Computer Society, 2017, pp. 959–966. Available from DOI: `10.1109/ICTAI.2017.00147`.

59. BARTÁK, Roman; SVANCARA, Jirí; SKOPKOVÁ, Vera; NOHEJL, David; KRASICENKO, Ivan. Multi-agent path finding on real robots. *AI Communications*. 2019, vol. 32, no. 3, pp. 175–189. Available from DOI: `10.3233/AIC-190621`.

60. SHINNERS, Pete; PYGAME COMMUNITY. *PyGame: Python Game Development* [online]. 2020 [visited on 2020-05-16]. Available from: `http://pygame.org/`.

61. SURYNEK, Pavel. Lazy Modeling of Variants of Token Swapping Problem and Multi-agent Path Finding through Combination of Satisfiability Modulo Theories and Conflict-based Search. *Computing Research Repository* [online]. 2018, vol. abs/1809.05959 [visited on 2020-05-16]. Available from arXiv: `1809.05959`.

62. SURYNEK, Pavel. Unifying Search-based and Compilation-based Approaches to Multi-agent Path Finding through Satisfiability Modulo Theories. In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. ijcai.org, 2019, pp. 1177–1183. Available from DOI: `10.24963/ijcai.2019/164`.

Appendix A

Acronyms

ACPF	Adversarial cooperative pathfinding
BFS	Breadth-first search
CBS	Conflict-based search
CCBS	Continuous-time conflict-based search
CPF	Cooperative pathfinding
DFS	Depth-first search
ICTS	Increasing cost tree search
LED	Light-emitting diode
MAE	Multi-agent evacuation
MAPF	Multi-agent pathfinding
RGB	Red, green, blue (color model)
SAT	Boolean satisfiability problem
TAG	Time expansion graph
XML	Extensible markup language

Appendix B

Contents of enclosed SD card

```
readme.txt ........................ the file with SD card contents description
src ....................................... the directory of source codes
  ozonav ............................... implementation of the prototype
    README.md ............. the file with information about the prototype
  boox ........................................ boOX program repository
  thesis ............... the directory of LaTeX source codes of the thesis
visdoc .............................. the visual documentation directory
text ............................................ the thesis text directory
  thesis.pdf ........................... the thesis text in PDF format
```