Multi-Agent Systems and Planning with MiR

Oliver Fleckenstein

s144472@student.dtu.dk

Technical University of Denmark

September 23, 2019

Abstract

This project studies how methods and theory from Multi-Agent System (MAS) can be applied at Mobile Industrial Robots to improve collaboration. First, a tool to simulate the robots and their environment is created, which allows for various layouts and scenarios. A design for a MAS is discussed, focusing on communication between agents, how plans can be constructed, and an algorithm for solving the Multi-Agent Path-Finding problem. A variation adding weights to edges in the search graph is proposed, which helps minimize conflicts. The MAS is evaluated based on the simulation tool, using warehouse environments as a case study. This shows that the MAS is able to handle dynamic objects in an environment and scale up to many robots.

Preface

This thesis was conducted as the conclusion for a M.Sc in Computer Science and Engineering in the period between February 1st and June 28th 2019, with Thomas Bolander as supervisor.

The project studies robots made by Mobile Industrial Robots (MiR) and how theories of Multi-Agent Systems can be applied to make their robots more collaborative. A special thanks goes to Niels Jul Jacobsen, who has been the primary contact at MiR and has helped guide the project.

Contents

Abstract **I**

Preface **II**

1. Introduction **1**
1.1. Aim and scope . . . 1
1.2. Outline . . . 2

2. Analysis and Related Work **3**
2.1. Mobile Industrial Robots . . . 3
2.2. Existing MAS solutions within robotics . . . 4
2.2.1. Kiva Systems (now Amazon Robotics) . . . 4
2.2.2. Australian Artificial Intelligence Institute . . . 8
2.2.3. CoBots . . . 8
2.3. Summary . . . 9

3. Methods and theory **10**
3.1. Multi-agent systems . . . 10
3.1.1. Multi-agent path-finding . . . 11
3.1.2. Artifacts . . . 12
3.1.3. Workspaces . . . 13
3.2. Planning . . . 13
3.2.1. Hierarchical planning . . . 14
3.3. Multi-agent execution and plan monitoring . . . 15
3.3.1. Plan recovery . . . 16
3.3.2. Continuous planning . . . 16
3.4. Summary . . . 17

4. Simulation **18**
4.1. Artifacts in the environment . . . 19
4.2. Simulation of robots . . . 20
4.3. Simulation of tasks . . . 21
4.4. Summary . . . 21

5. Multi-Agent Systems for MiR **22**
5.1. Agent architecture . . . 22
5.1.1. Reactive layer . . . 23
5.1.2. Modelling layer . . . 23
5.1.3. Planning layer . . . 25
5.2. Communication . . . 25
5.2.1. Messages from robots . . . 26
5.2.2. Inter-agent communication . . . 28
5.2.3. Workspaces - Limiting communication to local agents . . . 29
5.3. Distribution of tasks . . . 30
5.3.1. Auction based task distribution . . . 31
5.4. Planning and coordination . . . 31
5.4.1. Search strategies . . . 31
5.4.2. Tasks and plans . . . 33
5.4.3. Merging tasks . . . 37
5.4.4. Aborting tasks . . . 37
5.5. Path finding . . . 38
5.5.1. Weighed graph search . . . 40
5.6. Summary . . . 41

6. Evaluation and discussion **42**
6.1. Experiments and results . . . 42
6.1.1. Confined space . . . 42
6.1.2. Weighed graph . . . 45
6.1.3. Dynamic elements . . . 46
6.1.4. Scaling . . . 47
6.2. Further work . . . 48
6.3. Summary . . . 50

7. Conclusion **51**

References **52**

Appendices **52**

A. Implementation notes **53**
A.1. Simulation vs. real-world . . . 53
A.1.1. Monitoring and statistics . . . 54
A.1.2. Defining environments . . . 54
A.2. Centralized message delivery . . . 54

B. Simulation data **56**

List of Figures

2.1. MiR200 Robot 3
2.2. MiRHook200 Robot 4
2.3. Illustration of a typical layout of a Kiva warehouse 5

4.1. Example of a simulated environment with robots 19
4.2. Illustration of the area of an agent's sensor information. 20

5.1. The layered structure of a robot agent 23
5.2. Sending a message from one agent to another 26
5.3. Illustration of how announcements are send to all agents in the system . . 27
5.4. Overview of a DeliverAndStoreShelf 36
5.5. Illustrations of path-finding. 39

6.1. Comparison of the duration for completing task between SR and NSR. . . 43
6.2. Comparison of robot related statistics from SR and NSR simulations. . . 44
6.3. Comparision between weighed graph searches with different penalty values 45
6.4. Comparison of MAS in environment with dynamic objects. 46
6.5. Heat map conflicts in simulation with five agents and five unpredictable elements. 47
6.6. Comparison of the MAS with different amount of robots. 48

List of Tables

6.1. Mean difference in distance and conflicts for weighed graph search 45

A.1. Definition of the simulation's language to create environments 54

B.1. Task related data for the restricted space simulations on a small map . . . 58
B.2. Robot and agent related data for the restricted space simulations on a small map 59
B.3. Task related data for the non-restricted space simulations on a small map 60
B.4. Robot and agent related data for the restricted space simulations on a small map 61
B.5. Task related data using the WeighedGraph search with $penalty = 3$ . . 61
B.6. Robot and agent related data using the WeighedGraph search with $penalty = 3$ 62
B.7. Task related data using the WeighedGraph search with $penalty = 5$ . . 62
B.8. Robot and agent related data using the WeighedGraph search with $penalty = 5$ 63
B.9. Task related data using the WeighedGraph search with $penalty = 10$. 63
B.10. Robot and agent related data using the WeighedGraph search with $penalty = 10$ 64
B.11. Task related data with five robots and unpredictable elements 64
B.12. Robot and agent related data with five robots and unpredictable elements 64
B.13. Task related data with ten robots and unpredictable elements 65
B.14. Robot and agent related data with ten robots and unpredictable elements 65
B.15. Task related data for simulations with many robots 65
B.16. Robot and agent related data with many robots 65
B.17. Task related data for scaling robots and $penalty = 3$ 66
B.18. Robot and agent related data for scaling robots and $penalty = 3$ 66
B.19. Task related data for simulation with many robots and $penalty = 5$ 66
B.20. Robot and agent related data with many robots and $penalty = 5$ 67

CHAPTER 1

Introduction

Multi-agent systems (MAS) are composed of several intelligent agents, capable of interacting with each other to achieve both common and individual goals in a shared environment. Intelligent agents are in general defined to be *autonomous*, *proactive*, *reactive*, and in the case of MAS also *social* [**?**, p. 8]. Agents in MAS can be applied for both physical robots or a computational entity and are able to autonomously plan and act to achieve their desired objectives. With the use of MAS, problems that are difficult or impossible to achieve by a single-agent can become possible through coordination and collaboration between multiple agents. However, this also introduces new challenges: how does agents interact with each other, share resources and distribute tasks, make plans where other agents' goals are considered, and resolve conflicts when they appear. This project studies these challenges, focusing on multiple robots collaborating in a shared physical space. It is based on robots developed at Mobile Industrial Robots (MiR) with the environments and problems they are trying to solve.

Included in the challenges mentioned is one of the central problems within MAS: multi-agent path-finding (MAPF), which is concerned with find paths within a shared environment without having the agents collide. Several algorithms exist that tries to solve this challenging problem, but it still remains an open research problem. MiR's robots are working in highly dynamic environments, where new obstacles can appear and new task are created throughout the lifetime of the robots, introducing another layer of complexity. The robots must be able to adjust and make new plans in real-time, taking into account existing plans.

1.1. Aim and scope

This project analyzes the robots developed at MiR and how they operate today, and how the aspects of MAS can be applied to improve their robots' ability to collaborate. Their robots are built to automate transportation of goods around work environment. The cases being studied in this project will focus on transportation within warehouses and distribution centers. The robots have to work in dynamic environments alongside humans, where they only receive a limited information through their sensors. The goals

for the system are changing often, requiring the robots to adapt their plans to handling new tasks.

The focus will first be on building a tool to simulate the robots and their environment, along with the tasks they are being applied to solve. A new simulation tool is needed which is capable of simulating the MiR robots accurately and can scale to host hundreds of robots in one instance. This tool is necessary to evaluate different strategies applied by the MAS, and create comparable scenarios. A required feature is to easily be able to define the environment, allowing for users to visualize and test different layout configurations or the number of robots. The robots should only be provided with limited information equivalent to the real robot, thereby simulating the partial observability of the environment. The tool should include humans that dynamically changes the environment.

Secondly, this project will look into designing and implementing a MAS solution for this problem, using the simulator to verify that correct plans can be created. The main criteria for the system is for agents to be able to distribute and solve tasks between them, without occupying the same space at any time. Agents must therefore be able to make agreements on who has access to which resources at any time. As the system should be able to scale to many robots, a look into how planning can be divided into subproblems will also be done, limiting coordination to a subset of the agents. The MAS will be designed with the constraint in mind that it should be possible to apply to the real working robots.

1.2. Outline

The outline of this report is as follows:

- Chapter 2 describes MiR and their robots, along with a look at MAS applied elsewhere.
- Chapter 3 describes the relevant methods and theory of multi-agent systems.
- Chapter 4 describes how the environment of the MiR robots are simulated, along with the specific cases that are being focused on.
- Chapter 5 describes the design of a MAS, attempting to solve the challenges described in the introduction and created in the simulation.
- In Chapter 6 the MAS described in the previous chapter is analyzed and evaluated, discussing how the given system can be applied to the MiR robots.
- Lastly, Chapter 7 will conclude the project.

CHAPTER 2

Analysis and Related Work

2.1. Mobile Industrial Robots

Mobile Industrial Robots (MiR) is a robotics company based in Odense, specializing in building autonomous robots for transportation within industry, logistics or health care. Their the robots are in use today at hospitals, where they used to transport medicine to patients and doctors, and on factory floors to transport materials between different workstations. The company is building several different models of robots, such as the MiR200 seen in Figure 2.1, where several different sizes of robots exists. Their smallest robots have the dimensions of 0.89 m long, 0.58 m wide and 0.352 m tall, capable of lifting 200 kg or pulling up to 500 kg [**?**], with their larger robots being even more powerful. Wheels at the bottom of robot allow for fast movement around a floor at up to 1.1 $\mathrm{m\,s^{-1}}$. With large batteries, the robots can work up to ten hours on a single charge, traveling up to 15 km, thereby making them able to work a full human workday. Recharging batteries can be done in three hours [**?**], and is built into the robots routines which make them automatic seek charging stations when needed.

Figure 2.1.: MiR200 Robot (from [**?**])

Different attachments exists which the robot can equip. These make a robot able to interact with its environment and be used by humans in multiple ways. The attachments that are primarily in use consists of a shelf, allowing humans to place and retrieve items from a robot, and a hook making the robot capable of pulling heavy load (Figure 2.2). Other tools are build for more specialized use, such as a robot arm placed on top of the robot, allowing it to interact with complex object.

Figure 2.2.: MiRHook200 Robot (from [**?**])

Their robots are build for collaboration, both between humans and other robots, where multiple robots and humans are able to share the same physical workspace. Currently, they have the MiRFleet software, which allows users to manage multiple robots, allowing them to setup tasks (or missions as they are called) for the individual robots. However, the software is limited to defining tasks and giving them directly to the individual robots by a user, missing features such as automated task delegation and having multiple robots working together on the same tasks. As such, robots only have individual goals, no shared goals exists for the fleet, limiting their ability to collaborate on tasks. Requiring a human to assign tasks makes it difficult to scale to hundreds of robots. Having one hundred robots completing tasks that takes on average ten minutes would mean that a robot is completing a task and free to take on a new every 6 s, which might still be feasible, but difficult for a human to handle. Scaling to one thousand robots would make it every 0.6 s, making it impossible for a human to assign tasks fast enough.

2.2. Existing MAS solutions within robotics

MAS applications can be found in several places today, applied to both autonomous robots and pure software agents built to scrape the internet or solve optimization problems. All of these tries to solve different problems, where control over and information about the environment where the agents are working varies. The following sections will highlight some successful applications of MAS, both applied to robotics and virtual agents.

2.2.1. Kiva Systems (now Amazon Robotics)

Kiva Systems created an innovative new approach to automating warehouses and distribution centers using robotics and MAS to optimize storing and packaging wares. Before being acquired by Amazon in 2012 and transformed into Amazon Robotics, Kiva System published several papers describing their system, however limited information exists on how their system have been developed since. The summary given here is based on [**?**, **?**, **?**]. Existing warehouse solutions consist of static conveyor belts, which is fast but inflexible if the demand changes. Solutions with static shelves of items, where humans walk around with wagons to pick up items are very flexible, as new human workers can be hired when demand increases, but cost-ineffective and slow. Kiva's MAS approach combines a flexible and cost-efficient solution.

The system consisted of many small robots (called *drive units*) capable of driving around

and picking up shelving units (called *inventory pods*), which contains stored items. Each of the robots are mechanically relatively simple, as they are using two DC motors to move back, forth and rotate around their own axis, along with a lifting mechanism to hook into and move inventory pods. The robots can locate themselves using two cameras to read fixed barcodes on the floor and on inventory pods. The inventory pods consist of several trays with bins of different sizes which creates storage locations for many types of items.

The inventory pods are typically stored in a grid layout as illustrated in Figure 2.3, arranging the pods in rectangle storage zones with connecting paths in between them. Around the perimeter of the grid are *inventory stations*, where human workers either refill the inventory pods or pack new orders. The robots are tasked with locating the correct bins, drive it to the inventory station, where a human worker can interact with the bins, either retrieving the desired items or refilling empty bins. The inventory station will use lights to inform the worker of which items should pick up from the given pod. When finished, the robot can drive off to either store the pod again, or deliver items to other stations if needed.

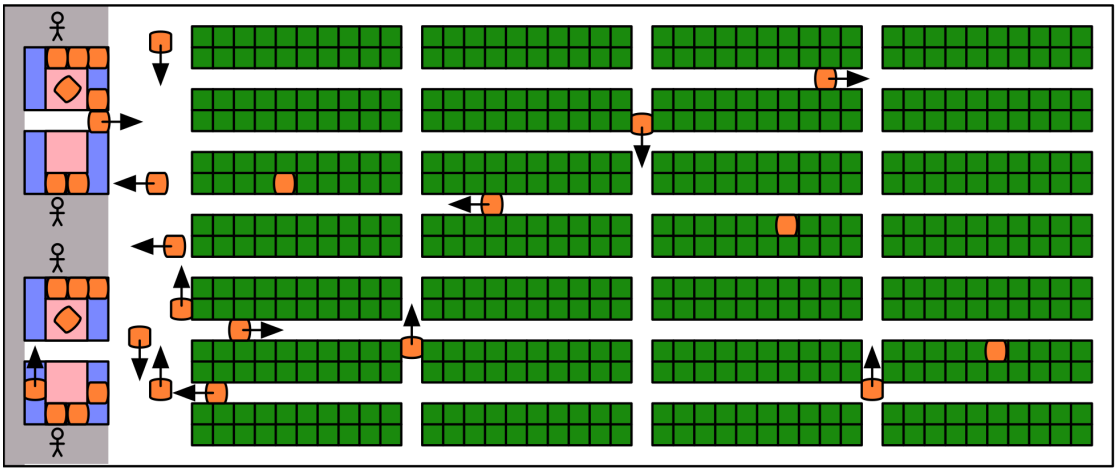


Figure 2.3.: Illustration of a typical layout of a Kiva warehouse. Green cells represent inventory pods, orange drive units, and to the left is inventory stations (from [**?**]).

Kiva's solution has been implemented in many places with up to 500 of robots working in the same warehouse [**?**]. The improvements from the system comes from the fact that human workers do not need to walk around to retrieve items and can focus on the more complex tasks of packing items. The robots allow the workers to have random access to all of the items in the warehouse, and by having enough robots to keep a queue at each inventory station, the system can ensure that the workers are always busy. This means the system can deliver a new inventory pod every six seconds, allowing 600 items to be picked per hour [**?**]. Even more is possible, if multiple items can be picked off from the same inventory pod.

Design of MAS in Kiva

The Kiva System represents each robot by a *drive unit agent* (DUA) and each inventory station by a *inventory station agent* (ISA), while the inventory pods are global resources. Each robot works as an independent unit capable of functioning completely by itself, however, the complete system requires the robots to be cooperative to fulfill customer orders. A centralized *Job Manager* (JM) handles the incoming customer orders and assigns robots, pods, and stations to complete the job. Assignment is based on different heuristics, which for ISA units is based on which orders they currently have assigned, as orders with the same items have a higher change of picking multiple items from one pod. For DUA, the JM primarily is base assignments on physical distance to inventory pods. When a DUA have been assigned a task, it will handle its own path and motion planning to retrieve the relevant inventory pods.

The assignment is divided into multiple phases, where a order is first assigned to a ISA, then afterwards DUAs and pods are found. Through communication between the DUA and ISA, scheduling of when the robot should arrive at the inventory station is agreed upon, ensuring that the workflow of the worker is optimized. Each order is handled completely independent of other orders, which encapsulate information and ownership to the agents. The JM only gives information that is strictly necessary for an agent to know.

As the environment is designed in a simple grid and the robots can only move in two dimensions[1], a weighed graph over the warehouse is constructed, where weights can be given at design time. The DUA can then use a textbook implementation of the A* algorithm to do path planning to find inventory pods and stations. However, no information is given about Kiva System's solution to the multi-agent path-finding problem.

The other important part of the planning is handling resource allocation, where the goal is to minimize the number of robots and items in storage, while completing orders fast and keeping the human workers busy [**?**]. This consists of several interesting challenges. Everything have to be handled in real time, as new orders are received throughout the day, where new orders can have higher priorities than current orders. The priority of an order can also changed based on the time of day, as the last truck of the day can have left, meaning it is impossible to further process the order until the next day. Warehouses can contain hundreds of robots and millions of different products, which make planning and finding the closest items more difficult.

The location of inventory pods, and thereby items, is dynamic. Each time a pod is retrieved and needs to be stored again, it might not be possible to store the pod in the same location as before, nor is it necessarily the optimal solution. Instead, a new storage location must be found, where the agent need to consider how often this resource is accessed and how accessible it is, i.e. popular items should be placed close to the inventory station for faster retrieval. The goal is to minimize the distance that the robot

[1] An elevator for the robots has been constructed to allow movement in three dimensions but no recorded use has been found.

needs to travel to store the pod, plus the distance for the next robot to retrieve it. Along with this, each pod also have multiple items on them. Items that are packed together often should therefore be placed on the same inventory pod, allowing multiple items to be retrieved at once. One item might be needed at multiple stations at the same time, which would mean a delay if there is only one bin with given item. All of these parameters must be considered when choosing which bins to store items in and where to store the inventory pods.

Lastly, the robots are delivering items to humans, which are by their nature unpredictable. The interaction time at each station will vary, and the agents must be able to adjust their plans accordingly. All of this makes it difficult to create an optimal plan for the global problem. Instead, Kiva applies several utility-based heuristics to make decisions online, which is tweaked by measuring how well the system is performing.

Suggested Improvements

There are already different studies looking into improvements for the Kiva system [**?**, **?**], from improving how the robots perceive their environment to improving system efficient and robot collaboration. [**?**] suggests several improvements and their benefits to the structure of the KIVA system, including transitioning the MAS to a more decentralized approach. This would make the system more robust by avoiding any single point of failure. As explained earlier, the KIVA system relies on a central job manager for task planning and distributing, which currently will damage the entire system if it failed. If it stopped working, all of the robots would be unable to complete any orders. However, it also means giving up more control to the individual robots, making it more challenging to optimize the global problem.

Optimization of the slotting of storage pods are also an interesting problem introduced by Kiva robots that still remains an unsolved challenge [**?**]. In general, the Kiva System tries to keep the most used items closest to the picking stations, however the system will currently only move a shelf if it is necessary for completing an order. Because of this, a shelf that is never used can take up space close to the picking stations forever. One suggestion is to use robots not currently working on a task to reorganize shelves for faster access in the future. This will require more energy for the agents, which would otherwise be idle.

Another suggested improvement is to add roles to the robots, to limit the overall traffic in the warehouse. Instead of having every robot be in the same retrieve, deliver, store loop, some subset of the robots could be tasked with retrieving shelves and deliver them to a pick-up area, where other robots could pick up the shelves and drive them to the picking stations. This would help avoid every robot having to drive into the congested areas between storage units.

When the robots are recharging their batteries is another optimization problem where improvements are possible. The Kiva robots will automatically go to a charging station when needed [**?**], but this can sometimes result in queues at the charging stations. This

can be acceptable, as long as the system is not busy and the robots waiting in line would be idling otherwise. However, it can prove problematic if the packing center is busy and could use more active robots. One solution could be to let the robots know how busy the entire system is, allowing them to consider if they should go recharge even though their battery are only partly depleted. Even though the robots could be used at a given time, using predictions about how busy the system will be in the future can be used to ensure more robots are charged and ready at that point in time. As the Kiva System already relies on a centralized job manager, this could keep track of when charging stations are going unused, and then able to distribute special recharge jobs to idling agents.

2.2.2. Australian Artificial Intelligence Institute

One of the early applications of MAS in action is the HORIZON project created by the Australian Artificial Intelligence Intelligence (AAII) to manage air traffic around Sydney airport [**?**]. The goal of the project was to reduce air congestion and optimize the use of the runways. This project was build using dMARS (distributed multi-agent reasoning system), an environment created by the AAII to build and test software agents. Each of the agents in this system was designed from a belief-desire-intention model, contained a set of beliefs about its environment, including the state of themselves and other agents. Along with these, each had a set of desires or goals they wanted to achieve, and a set of plans describing how these desires could be achieved. An intention structure kept track of all the plans that had been chosen for executing.

The system consisted of two classes of agents: *global agents* responsible for coordination and reasoning between aircrafts, and *aircraft agents* which preformed computations and reasoning for the individual aircraft. Several different global agents existed, each with their own responsibilities. One agent was responsible for coordinating activities between all other agents, and another for computing the sequence of aircrafts takeoffs and landings in order to minimize delay. A *trajectory* agent analyzed the global plan to verify that no conflicts existed, effectively ensuring that aircrafts did not crash into each other. An interesting part of the system was its ability to share important knowledge between aircrafts, such as gathering and combining wind observations from all aircrafts, thereby improving every agent's model of the environment.

2.2.3. CoBots

CoBots shows an interesting use of collaborative robots, working in a dynamic environment and interacting with both other robots and humans, described by [**?**, **?**]. The CoBots system consists of four robots, each building on its predecessor, capable of moving around office buildings, including taking elevators between floors. Their tasks include delivering items to other people and taking guests on guide tours though out the building. They have the ability to speak out load, explaining what guest see or to deliver voice messages in person. This also allow the robots to ask for help from humans. Their localization capabilities are limited, making the robots lost sometimes, or they might find obstacles that make it impossible for them to complete their task. In both cases,

they can ask a human for help, which can than tell the robot where it is or move the obstacle blocking its path.

While the robots in this system primarily focused on interaction with humans and not each other, they where still able to inform each other about jobs that had to be completed. This was done through a scheduler, which would receive commands from both robots and humans. The scheduler was responsible for computing the assignments of jobs to robots, satisfying the constraints about time-windows, location, and transportation capacity.

2.3. Summary

As seen in this chapter, MAS has already seen applications within robotics. With the advancements within robotics, new opportunities are created that require interaction with not just with humans but also other robots. MAS has helped modelling how agents can interact, along with the inter-agent relationships and how information and knowledge is shared. Both AAII and Kiva has been examples of how using MAS has helped dividing complex problems into smaller subproblems, that can be compartmentalized and distributed to several agents. This allows MAS to solve problems that are difficult, if not impossible, to solve for a single-agent. Information given to the individual agents are limited to a minimum, encapsulating data to where it is necessary. In Kiva, multiple types of agents have been defined, each capable of solving different kinds of problems, which they can be divided into different organizations with their own responsibilities.

CHAPTER 3

Methods and theory

This chapter will introduce the general theory multi-agent systems that is used throughout this project, while Chapter 5 will describe the details of the design and implement of a MAS applied to the MiR robots. The focus will be on how the system models the agent and the environment using artifacts, while introducing the multi-agent path-finding problem. Methods for agents to plan are discussed, with a look at plan execution and monitoring.

3.1. Multi-agent systems

Multi-agent systems (MAS) can be defined as a set or organization composed of more than one agent, who communicate and coordinate their actions to achieve their goals. Goals in MAS can both be individual and shared, the latter allowing several agents to work towards the same goal. An example of MAS is a network of sensors monitoring traffic in a busy intersection attempting to regulate and optimize traffic, or fleet of self-driving cars that all want to carry their owners to their desired destinations. Each of the agents in such a system is seen as being an autonomous entity (e.g. a software program or robot), which are *reactive* to the changes in its environment, *proactively* plans towards goals further out in the future, and are *social*, thereby making the agents capable of coordinating their actions [**?**, p. 8]. Each of these entities might have several limitations, including processing power, memory, and power supply.

MAS are characterized as a system ([**?**, **?**]) where:

- No global control exists to inform agents of their actions. This means that an agent should be able to plan and carry out actions, even if it is not connected to the reset of the system.
- The data in the system is decentralized.
- The agents can plan and carry out tasks asynchronously.

A subset of the problems studied within MAS are considering partial observable environments, where agents only perceive limited information of its surroundings. Each

individual agent therefore have incomplete information about the global state. This makes it difficult for a single agent to consider solutions for global problems, although a single agent might be able to find sub-optimal solutions.

One of the central elements in MAS is coordination. Agents make plans towards achieving their own goals, and must coordinate with other agents in the same environment to ensure conflicts between individual plans does not occur. A goal can be impossible to solve for a single agent, and it must therefore be able to cooperate towards their common goal. Some agents might be selfish, only deliberating about how to solve its own goals, being unwilling to help others. They can be directly hostile, e.g. in multi-player games, where agents work towards conflicting goals or have a direct interest in preventing opponents' goals. In this last case, agents might communicate false information, which introduces how trustworthy different information sources are. However, the focus in this project will be on cooperative agents.

3.1.1. Multi-agent path-finding

The multi-agent path-finding (MAPF) problem is central to MAS where agents share the same physical space. The generalized version of MAPF based on [**?**] is given here:

Definition 3.1.1 (Multi-Agent Path-Finding (MAPF))**. Input**: A graph $G(V, E)$, a set of k agents $(a_1, a_2, \ldots, a_k)$, a set of initial locations $s_1, s_2, \ldots, s_k \in V$, and a set of goal locations $g_1, g_2, \ldots, g_k \in V$. **Output**: A set of plans, where a plan is a sequence of MOVE, traversing an edge to a neighbouring vertex, and WAIT actions. The plans must take each agent a_i from s_i to g_i.

The solution must satisfy that no agents share the same vertex at any time, i.e. let $\pi_i(t)$ denote the vertex of agent a_i at time t, then a solution to the MAPF problem has to satisfy $\pi_i(t) \neq \pi_j(t)$ for all time points t and all pairs of agents $(i, j) \in \{(i, j) \mid i \in [0, k] \wedge j \in [0, k] \wedge i \neq j\}$. Note that two agents cannot traverse the same edge at the same time either, meaning that for two adjacent vertices a and b, two agents cannot swap positions in one time step (from a to b and from b to a), as they would pass through each other. They are however allowed to follow each other, i.e. a_i moves from location b to c while a_j moves from a to b.

The general form of the MAPF problem is NP-complete, as it can be seen as a generalization as the sliding tile puzzle, which has been shown to be NP-complete by [**?**]. It is therefore likely that no algorithm exists that can find an optimal solution in polynomial time. Different approaches to solving MAPF exists, such as using a standard A* algorithm to search the state space of all possible permutations of the k agents' locations in the graph [**?**]. If b_{single} is the branching factor for a single agent, then $b_{multi} = (b_{single})^k$ is the branching factor for global A* considering k agents, i.e. $(b_{single})^k$ states has to be explored at each depth.

Several other algorithms are discussed in [**?**] which focuses more on dividing the problem into subproblems for the individual agents. Local Repair A* (LRA*) is an algorithm

where each individual agent creates in own plan without considering the other agents. The agents begin execution of their plan, and continue until a collision would happen in the next step. When this happens, the agent will recalculate the remainder of its route to its goal. LRA* has several issues, including cycles causing it to be unable to find a solution. Crowded regions have many conflicts and will force agents to recalculate the full A* algorithm often.

Cooperative A* slices the global planning problem into a series of single-agent searches. Each single-agent search is done in a three dimensional time-space state space, where the planned paths of other agents are taken into account. This is done by marking the position of other agents as unusable at the time step when they will be there. When an agent has calculated its path, the next agent start their search, taking the previous agents plans into account. It is noted that because of the greedy approach of this algorithm, some problems might be unsolvable. Agents planning early in the sequence might plan paths that block other agents.

3.1.2. Artifacts

A newer addition to the field of agents and especially multi-agent systems is the concepts of *artifacts*. The agents and artifacts (A&A) meta-model introduced in [**?**] explains how artifacts can help model reactive entities in environments, that does not themselves have goals. This is where the main difference lay in agents and artifacts; an artifacts is something that can be used by an agent, but does not by themselves do anything unless instructed. Artifacts provide a service for other agents to achieve their goals.

An artifact can model many different things in a MAS, but are in general used to model services or functions in the agents' environment, where they define how agents can interact with the environment and each other. This spans from a computational entity that supplies information or can provide some computational resources, to a physical object in space. The artifact model are build with several concerns in mind, described in [**?**]. An artifact should be an *abstraction* of the entities the agents interact with in the environment. This in general means that the artifact is perceived through an interface or protocol which agents can understand and use. Artifacts should be *modular* by providing an explicit way of modularizing components of the environment into units of functionality, each *encapsulating* data, which is part of the global state, and a set of operations that can be applied to the artifact (the interface). Agents can retrieve this data through either an event based approach, notifying agents that have subscribed to a given artifact, or by manually pulling the data from the artifact. They should be *extensible* and *adaptable*, allowing for more complex artifacts to be build from simpler once, and for agents to dynamically construct, change, and destroy artifacts during execution. Common protocols for how these actions are done also allows agents to define their own types of artifacts as needed, by extending from some base artifact protocols. Lastly, *reusability* is an important feature of artifacts, allowing multiple instances of the same artifact type to be in use at the same time. The CArtAgO framework [**?**] is implemented to allow for creating of artifact-based environments, allowing each integration with the

Java programming language.

3.1.3. Workspaces

To organize agents and artifacts together in logical groups, the concepts of a *workspace* is introduced in [**?**]. The concept of agent organization is not new, with a full chapter dedicated for just that by [**?**], but is in a workspace joined closer together with the environment and its artifacts. Using workspaces, the environment is divided into smaller parts, where an artifact can only exist within one at time. For an artifact to be accessible for an agent to use, it has to belong to the same workspace. However, nothing limits agents to only belong to one workspace, allowing them to access artifacts within different workspaces concurrently. Agents are capable of dynamically joining and leaving workspace, thereby directly controlling which resources (artifacts) they have available. With the workspace model, coordination between agents are confined to the agents in the same workspace(s), as it is ensured that nobody outside the workspace can access the artifacts that are involved. The global planning and coordination problem can therefore by simplified to a subset of agents.

Examples of workspaces could be defined by physical limitations, such as an office where an agent would have the artifacts within the room available only while inside. A workspace could also be virtual entity, e.g. a file directory containing shared files that multiple agents can access. As mentioned in Section 3.1.2, artifacts within a workspace can be used to coordinate tasks between agents. Take a classroom (the workspace) with students and a teacher (agents). The students want to know when they can go and ask the teacher a question. How do they coordinate who should go when? If everyone goes when they want, they might waste time idling when others are asking their question, or might be skipped if they choose to sit and wait for an opening. Instead, the blackboard (an artifact) can be used for coordinating which student is currently communicating with the teacher. Students can write their name on the blackboard, with the names of students acting as a queue. The teacher can then, when available, consult the blackboard to learn the name of the next student, cross out the name, and call the student up. Nobody outside the classroom is able to use the blackboard and have no knowledge (nor need for it) about which student is currently talking with the teacher.

3.2. Planning

There exists several different approaches to planning, both for single-agent (SA) and multi-agent (MA) scenarios, where many approaches to the MA scenarios are carried over from SA cases but with an added solution to solve coordination. To solve the coordination problem, there exists solutions to both do local planning prior to coordination, as well as doing coordination before doing any local planning.

Solutions of MAS where coordination is done before local planning is often done using predefined plan templates for how each agent that are involved in the problem can access resources, and on a high level, which order the agents should carry out their actions. The

concrete details of how each agent then solves its sub-problem can then be done with local planning, without concern for conflicts. However, this requires that the system has plans and templates to resolve all possible problems, which can be difficult and costly, for example if actions planned and taken locally by an agent after coordination affects other agents.

The opposite approach, where agents first do local planning then coordination, is appealing when many of the agents' goals are largely independent and can be carried out with limited help from other agents. This "divide and conquer" approach allows each agent to solve their planning problem locally in parallel, and then afterwards resolve any possible conflicts with other agents. This will therefore mean that coordination is only done when there are actual conflicts, and it is unnecessary to consider every scenario. The general algorithm for resolving these conflicts can be described as:

1. **Plan**: Each agent finds a plan to fulfill their own goal locally.
2. **Identify conflicts**: Agents find conflicts that would occur if their plans are executed together. This can either be done through a centralized coordinator, or directly between agents.
3. **Repair plan**: The agents change or inject new steps into their plans to avoid the found conflicts. In many cases, this could be by waiting for another robot to move out of the way or take a slighly altered route.
4. **Execute**: If all problems are resolved, the agents are done and the execution of their plans can start. Otherwise, if there are some conflicts that the agents cannot resolve, an agent involved in the conflict is chosen and will restart the algorithm from step 1, generating an alternative plan and attempt coordination again.

3.2.1. Hierarchical planning

One approach to doing planning is to construct a library of structured hierarchic plans, that is building from abstract plans to concrete actions that an agent can take. A complete solution to a problem can then be constructed by first finding some abstract plan that promises to solve the problem, and then deconstruct it into smaller, more actionable steps.

First, a look at how this is applied for a single agent, who, for example, would want to go on a vacation. A library to solve this problem could contain an abstract plan saying: the agent should find a way to get to its desired vacation location, have the vacation, and then come back from vacation. The agent has thereby converted its one abstract plan into three less abstract but still not actionable steps. Multiple plans in the agent's library could now be applicable. Assuming the destination is not too far away, it could consider both driving or flying to get there. If it chooses to fly, it would have to first find tickets, buy them, then actually go to the airport and get on the plane. These steps might be implemented directly as concrete actions for the agent, and the planning for this branch can therefore be concluded. The agent would continue to plan what it

should actually do on the vacation and so on, until a complete plan for its vacation has been made.

In multi-agent systems the challenge of coordination is introduced into the problem. The approach for an single-agent can expanded to the multi-agent case, and is generally defined as *hierarchical task network* (HTN) planning. As in the single-agent case, plans are formed by starting from an abstract plan that promise to solve the problem, and then its steps are filled in by less abstract steps, until a concrete plan is found. At each level, the agents consider if there are any conflicts or coordination needed, resolves these issues, and continues to the next level of planning. This means that coordination is done early and can help avoid expensive computations of lower-level planning. However, a common issue in MAS planning is timing of actions, which in many cases still have to done on the concrete level, especially when working in continuous environments.

This approach also have the benefit of generally leading to more flexibly plans, as agents are free to change their plans at lower levels than where coordination is done. This does however also lead to less efficient joint plans, as plan optimizations are done locally for the single agent.

3.3. Multi-agent execution and plan monitoring

While planning, agents will often assume that its model of the environment remains static or at least predictable, or that planning is instantaneous. However, these assumptions does not hold in dynamic environments, where unpredictable objects might cause conflicts that are not detected because of the limited model. The agent will have incorrect assumptions about the environment or what actions in can perform, which can cause the agent's plan fail in the future. It is therefore necessary for the agents to monitor their plans to verify that everything is proceeding correctly and make adjustments if necessary.

In MAS this can cause cascading problems, if one agent is unable to execute its plan, other agents dependent on the first might be blocked as well. Even if the first agent can recover its plan, the altered steps might be done at a later time than other agents anticipate, causing their plans to fail. Agents must therefore coordinate their actions again.

During the execution of the agents' plans, something can cause the plan to fail at any time. In such cases, agents need to consider if they are currently accessing any resources in the environment which must be released to be available for other agents. An agent could for example be carrying an object, and if its goal suddenly becomes unachievable, it must first make a plan to place the object somewhere, preferably in a place where it will not block other agents and is still reachable in the future.

3.3.1. Plan recovery

In both single-agent and multi-agent problems, agents must consider if part of their plans can be reused or if they have to start over with a completely new plan. An example of this in a single-agent setting is when the agent move along some pre-planned path and some unpredicted object appears on the agent's path. The agent can new choose to start the path planning process over from its current position s and find a entire new path to the goal location g. Alternatively, the agent can attempt to only redo part of the path planning. The agent can assume some location l further ahead on its original is still reachable, and can instead search for a path from its current position s to some location l on the original path. The new path from s to l is then joined with the sub-path l to g from the original path, potentially saving time exploring known states. Worst case is when $l = g$, where the agent has redone the full path planning.

Choosing to start the planning process over from the beginning can be expensive, as it requires planning and coordination again which might be unnecessary. In a system with many agents, even through a few fail, most will likely be able to continue their original plan without any changes. Computation resources will therefore be wasted if old plans are not reused.

On the other hand, repairing old plans can be potentially very cost efficient but also more complicated. As mentioned above, one agent having to replan might force others to change their plans and do similar plan repairs. The agents might also rethink the allocation of tasks between them, which means that the goals for some agents are completely changed. Finding which agents that it is necessary to coordinate with again can prove difficult.

When using the hierarchical planning approach, one way to do plan recovery is to backtrack, using the updated model of the environment. Starting from the concrete steps that is not possible for an agent to execute, travel backwards through the planning tree until some abstract plan is found where its preconditions are still satisfied. Planning can then start from a lower point in the planning tree, and the necessary coordination steps can be identified.

3.3.2. Continuous planning

On top of the problems described in the section above, in many real-world applications of agents every goal or task might not be known from the beginning of their process. New objectives might be created by a human or some other external entity, which will affect what the agents should be doing. The agents in such a MAS must be able to handle these new objectives. This first includes agreeing on who should be working on the task, which will then have to be incorporated into the agent's current plan.

All of this means that every agent has to continuously repair or replan how they will achieve their goals. One approach to solving this problem is called *partial global planning* (PGP) [**?**]. This utilizes several different mechanisms to trade of the cost of planning and

coordination to create a new plan. Abstraction, for example with hierarchical planning, helps agents to coordination on a higher level, allowing them to frequently change the details of their plans, without other agents perceiving any difference. By sharing abstract plans between agents, each agent can do partial global reasoning to search for how its plan will fit into others plans, giving the best possible joint plan. This might even be by creating a plan that will change some of the other agents' abstract plans. Coordination of this can be solved by two different approaches. Either the system can assume that every agent will eventually formulate the same plan given the same information, then other agents will agree to the new joint plan when they revise their own local plan. Otherwise the agents have to do explicit coordination to agree on the new joint plan.

The PGP approach also allows the agents to act asynchronously. Instead of waiting for everyone to agree on the plan, the agent can start carrying out its own plan as soon as it is formulated. If others determine that there are any conflicts or suggest changes to the plan, the agent must consider these and replan. This might mean that the steps it have take so far is wasted, but as the agent would have waiting idling for the other agents to finish, its time would have been wasted anyway.

3.4. Summary

This chapter has introduced the theories of MAS applied to design and implement a system for the MiR robots. The problem of MAPF is introduced, along with some algorithms that attempts to solve it, each with different trade-offs. The A&A model provides an approach using artifacts for modelling the environment and helping coordination, along with workspaces for limiting access and coordination to nearby agents. A high-level algorithm for how agents can produce individual plans and coordinate them is given in Section 3.2, along with how plans and coordination can be done at different levels of abstraction. Lastly, a discussion of how plans can be executed and monitored, considering how MAS works in a continuous environment.

This of course does not cover the full domain of MAS. The focus has been on collaborative agents, leaving out subjects like agents working to achieve incompatible or opposite goals and whether information is trustworthy. Concepts within multi-agent learning has been left out but will be mentioned briefly in Chapter 6. Different techniques for coordination and reaching agreements have also been limited, however the details of how coordination is actually achieved using a formal protocol between agents and auctions to reach agreements will be given when applied in Chapter 5.

CHAPTER 4

Simulation

To develop and evaluate a MAS for the MiR robots, a tool to simulate the robots and their environment is required. The simulation tool enables repeatable scenarios, allowing different strategies to be evaluated under the same conditions. Secondly, it allows a MAS to be tested in different environments, e.g. testing different layouts for factory floor to optimize the MAS's performance. The simulation should provide the agents of the MAS the same information as the real MiR robots would.

As factory floors and warehouses are some of the primary use cases for the robots, it was chosen to focus on these types of environment. These environments also provide cases with many robots. One example of the types of environments included is based on Kiva's warehouses [**?**], which is illustrated in Figure 2.3. The objectives for the robots are focused on moving objects around in their environments to complete certain tasks. The environment from Kiva is extended to include *feeders* that can introduce new products into the environment, such that shelves can be restocked, along with *charging stations* allowing the robots to recharge.

The environment is represented as a two-dimensional grid, in which the robots can move around. An example of the simulation is illustrated in Figure 4.1. Each cell on the grid can either contain an object or not, along with possibly a robot. As robots in the real world cannot drive through each other, each cell is also constrained to containing at most one robot. The environment also contains cells that cannot contain any robots. There are two types of these: static cells that does not move during the simulation (walls), and dynamic cells. Dynamic cells marked as unreachable at anytime during the simulation, forcing robots to alter their paths and find ways around. Dynamic objects also exist, representing a human or other external entities that can move around the environment. These dynamic objects are capable of moving to any adjacent cell and are also capable of picking up and placing objects like robots, thereby introducing changes into the environment that cannot be controlled by the MAS. These changes will only be known to agents when they have observed them through their own sensors.

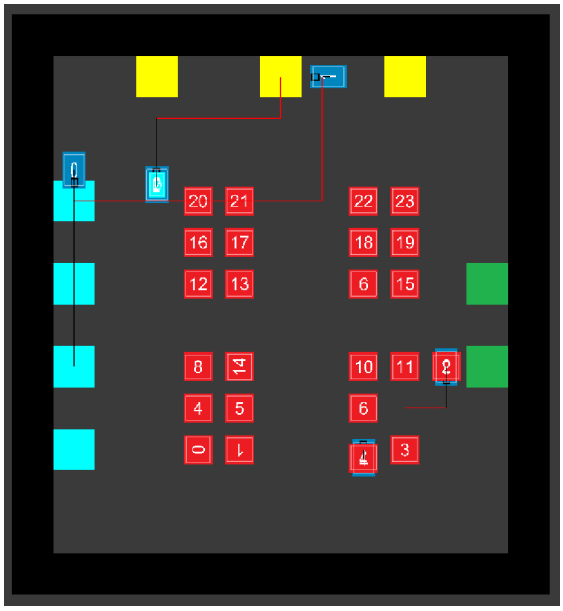


Figure 4.1.: Example of a simulated environment with robots (blue). Shelves (red) are objects that can be picked up by robots, feeders (cyan) places objects on robots, workstations (yellow) interacts with or removes objects from robots. Charging stations (green) allows robots to recharge.

4.1. Artifacts in the environment

Different types of objects are included in the simulated environment for the robots to interact with, which are all modelled as *artifacts* in the MAS. All of the described artifacts are illustrated in Figure 4.1. The two main artifacts that the robots can interact with are a *shelf* and a *feeder*. The shelf is representing multiple items or products stored together, which a robot can pick up and move around, similar to inventory pods in Kiva's system [**?**]. Like in their case, other workers, robot or human, can retrieve or place items on the shelf for packaging or storage respectively. Feeders represents workers that supply materials to the system. For a warehouse, this could be trucks arriving with new wares being unloaded on robots, for then to be stored. This is simulated by either at regular or irregular intervals producing new objects that robots can pick up, allowing for simulations that behave both deterministic and stochastic.

Certain locations in the environment are specified as *goal* locations, representing places where objects from shelves or feeders are required. Using the warehouse example again, these represent workstations where objects are being picked up and packaged. The last type of object included in the simulation is *charging stations*, which are the only cells where robots can recharge their battery.

4.2. Simulation of robots

A MiR robot can have different tools attached to it, making it capable of carrying out various tasks. For this project's purposes, the simulation is built to emulate robots that can pick up objects or having objects placed on top of them. Adding other types of tools is possible by extending the set of actions available to the robots. As such, the two main actions that robots have for interacting with artifacts in the environment is PICKUP and DROP.

Robots are limited to being able to move FORWARD and BACKWARD, which allows them to move from one cell on the grid to the one directly ahead or behind. The TURNCW and TURNCCW action rotates the robot 90 degrees around their own axis, clockwise and counter-clockwise respectively. These correspond to the four actions that robots are able to execute for movement. Only one action is allowed at any time. This limits the robot's movement compared to the real world, where robots are able to turn while moving forwards or backwards. However, robots are still able to reach the same locations in the world, while allowing a simpler model of how the robots move. To simulate the robots more accurately, an option to simulate the physics of movement is included, meaning the velocity will not be constant. Each movement action will be done with an acceleration and deceleration This makes it slightly more difficult to accurately model and predict how the robot moves when planning. When doing multiple FORWARD or BACKWARD actions in a row, the robot keeps its velocity between each action. Higher average speeds are therefore achieved when following straight paths compared to curved.

Running a robot in the real-world requires energy. Without energy, the robot is unable to execute actions. A simulated robot therefore also contains a *battery*, describing how much energy the robot has left. Each time an action is executed, the battery level will drop. As the robots are also running in real-time and have to use power just to stay awake, a small, constant drain on the battery is also included, penalizing spending time idling. A battery can be recharged at *charging stations* where the CHARGE action is possible.

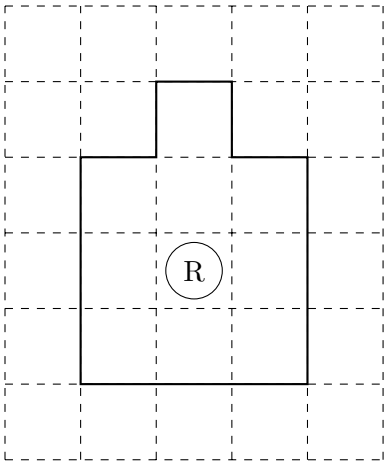


Figure 4.2.: Illustration of the area of an agent's sensor information.

Each robot also contains sensors to observe its surrounding environment. Sensor information is being published to agents as a, possibly empty, set of tuples containing the location and object captured by the sensors. For simplicity, it is assumed that the robots sensors are perfectly accurate in determining both the exact position of the observed object, and which object is being

observed. The sensors' range is limited to the two cells directly in front of the robot and the other seven cells surrounding the robot, illustrated in Figure 4.2. This corresponds to less than what the real sensors on MiR's robots can capture but represents a worst case scenario and serves as a simpler model of the robots.

4.3. Simulation of tasks

The tasks which the MAS has to carry out consists of bringing a shelf or objects from a feeder to predetermined goal locations. Simple tasks require one agent to bring one item to the given location, while more complex tasks can require multiple items. The order of which these items arrive can be given importance but are generally not included for the tests done in this project. Two different approaches to generate tasks that robots should carry out has been implemented. Both works by publishing messages that the MAS can capture and use to distribute and build plans for completing.

A task *creator* was made to simulate a real-world example where new tasks are generated dynamically throughout the run-time of the system. The creator will generate random tasks based on the current state of the environment. It does so by building its own model of the world, while observing robots to determine how busy they are. As such, this approach can balance how often tasks should be published, create both situations where many robots are idle and situations where there are more tasks to be completed than robots. Note that all tasks that are being generated are ensured to be solvable.

The *scheduler* implements a predefined list with pairs of tasks and times to publish them. By doing so, the same tasks can be used for multiple simulation, create repeatably scenarios for testing and evaluation of the MAS.

4.4. Summary

This chapter describes the simulation tool and a distribution of how the robots and their environment are modeled. The environments are focused around factory floors and warehouses, with shelves and feeders to provide objects that can be moved around. Unpredictable objects, such as humans, moving around creates a dynamic environment. Each robot can perform actions for movement, picking up and dropping objects, and receive limited precepts about its surroundings, making the environment partial observable for their agents. Lastly, two tools have been made to generate random and predictable tasks respectively.

CHAPTER 5

Multi-Agent Systems for MiR

This chapter describes how a MAS is designed and applied to the MiR robots, making them work in a shared environment to solve tasks. First, the overall architecture of how each robot agent is designed will be described, followed by a definition of how the communication is used by agents to coordinate their actions, along with how sharing these messages correctly is achieved. How tasks are being distributed between agents is explored, defining the messages necessary. The library of plans which each agent have will be described, showing how agents are able to find solutions to their goals. Lastly, an algorithm to solve the multi-agent path-finding problem is suggested.

The system designed in this chapter will be evaluated using the simulation tool. The results are presented in Chapter 6.

5.1. Agent architecture

The primary agent of the MAS is the *robot agent*, which exists in a one-to-one relationship with the robots. Next to this exists an *task manager* agent detailed in Section 5.3, which handles the distribution of tasks and keeps track of what agents are doing at a high level. Each agent is regarded as an independent process, each running isolated from each other. Agents will only receive information about each other through communication or perceiving them through their robot's sensors.

The robot agents are designed after a layered architecture, where each layer is responsible for part of the agent's processes. These consist of three layers: the *reactive* layer, *model* layer, and *planning* layer, illustrated in Figure 5.1. Tasks are sent to the agent from the *task manager*, which defines the goals for the agent. The reactive and model layers will receive percept information from the environment through the robot's sensors. The model layer will update the agent's model of the world, while the reactive can execute emergency actions (breaking). The planning layer uses the information from the other layers to build plans to achieve long term goals.

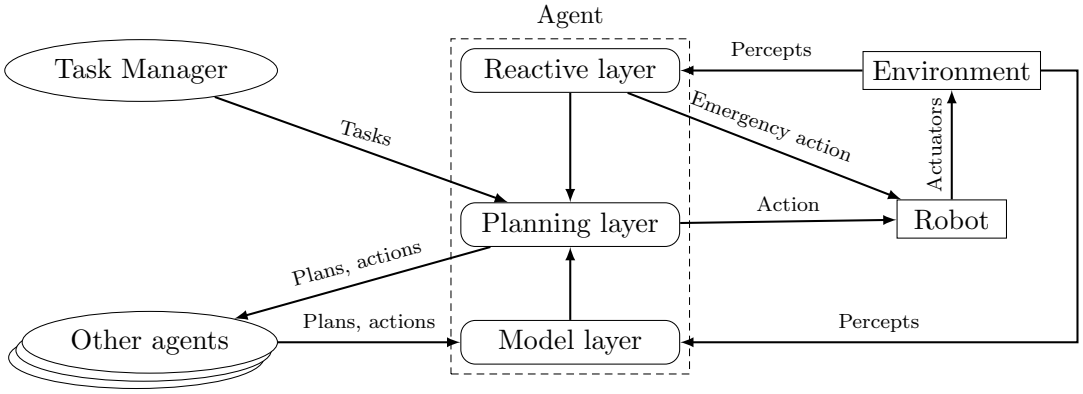


Figure 5.1.: The layered structure of a *robot agent*.

5.1.1. Reactive layer

The reactive layer is primarily tasked with ensuring the robot does not crash into other agents or objects in the environment and is the layer that handles sudden changes. It must respond quickly to the changes it receives from the robot's sensors, and does therefore not perform any complex planning. This layer will analyze where objects around the robot are located and predict where they will be within the short future. The primary concern is if there currently is anything in front of the robot, or if there will be in the near future. If anything is detected, it will immediately cause the robot to break, taking precedence over the remaining layers.

The reactive layer is also enforcing the *priority of the right* rule between agents, making the robot wait until the other has passed in-front of it from the right. This helps mitigate conflicts by attempting to wait a short while before starting the whole replanning procedure. While the planning layer should plan paths such that agents does not drive into each other, unpredictability in the movement might cause the robot to be out of synchronization from the current plan. This means that two agents might want to move into the same cell at the same time, which is where the right-of-way rule is enforced. Precisely, if this layer perceives that another robot is in front and to the right of itself and is currently on a trajectory that will cross its own path, the layer will break and to wait until the other robot have passed by.

The last task for this layer is to verify that nothing within the received precepts blocks the current path which the agent is following. If this is the case, it will pass the information on to the planning layer.

5.1.2. Modelling layer

The modelling layer is responsible for representing the agent's belief about the state of the world, as well as the robot it is controlling. The model is represented as a tuple $\mathcal{M} = (\mathcal{G}, \mathcal{W}, \mathcal{A}, \varrho, \omega, \mathcal{D}, \varphi, \mathcal{S}, \mathcal{R})$:

- $\mathcal{G}$ is a graph containing the static elements of the environment, including the floor, walls, charging and work stations. Each vertex in the graph represent a coordinate (x, y) in two-dimensional space. Edges represent possible actions from a vertex to every adjacent neighbour. The edges are directed and weighed with the cost of executing the action (the default cost of all actions is 1).
- $\mathcal{W}$ is the set of workspaces in the MAS, described in Section 5.2.3.
- $\mathcal{A}$ is the set of agents in the MAS.
- ϱ is a function mapping an agent to the current path it is believed to follow. This can be described as $\varrho : \mathcal{A} \mapsto \langle p_0, p_1, \ldots, p_k \rangle$, where $p_i = (x, y) \in \mathcal{G}$. The function ϱ is updated based on new announcements from agents.
- $\omega : \mathcal{A} \mapsto \mathcal{W}$ is a function mapping each agent to a workspace. Workspaces in MAS generally allow for agents to be in multiple workspaces but are here limited to be in only one at a time. As workspaces are defined by physical constraints, detailed in Section 5.2.3, it is also physically impossible for a robot be at two locations at the same time.
- $\mathcal{D}$ describes the set of artifacts in the environment, used by the agent to coordinate their intentions. Their types are introduced in Section 4.1.
- φ maps each location in $\mathcal{G}$ to a set of artifacts ($\varphi : \mathcal{G} \mapsto \mathcal{D}' \subseteq \mathcal{D}$). Note that while this allows multiple artifacts to be present at the same location, some types of artifacts are not allowed to share location due to physical constraints. Each artifact contains a $locked(A, X)$ and a $queue(A, L)$ property, where X is the current agent that is able to use the given artifact A, and $L = \langle a_1, a_2, \ldots, a_j \rangle$ is the sequence of agents, ordered by when the agent can access the artifact.
- $\mathcal{S}$ is a set of (x, y)-coordinates describing objects from the robots sensor which the agent cannot classify (humans or other dynamic objects). In each step, the agent receives a new set of sensor information, where if it is detected to be an artifact, updates φ accordingly and does not add the object to $\mathcal{S}$.
- Lastly, $\mathcal{R}$ describes the state of the robot, which itself is an tuple (p, b, s, e), where:
  - p is the robot's position
  - $b \in [0, 1]$ is the robot's battery level.
  - $s \in \{\textsc{Free}, \textsc{HoldingArtifact}, \textsc{ActuatorsWorking}, \textsc{Charging}\}$ represents the robots internal state.
  - e is an error signal, informing the agent of whether the robot can carryout the actions specified.

As the agents are also able to communicate with each other, agents can send information about the environment to each other. These are received by the model layer and

will update the agent's belief about the world accordingly. The language for this is described in Section 5.2. As all of the agents in this scenario are collaborative, the agents have complete trust in the messages they receive. However, if conflicting information is received from precepts and messages, the agent will choose its own precepts data over what other agents are informing them of.

5.1.3. Planning layer

The planning layer is tasked with planning which goal agent should currently be work towards, and how it should achieve this goal. This layer is the core of the agent and is where the complex deliberation process happens. It's *update* function is only called after both the reactive layer responded, and the agent's model has been updated. All the possible tasks that agents can handle are defined in Section 5.4.2, along with their plan library for how they solve the tasks.

If the agent does not have a goal or task, it will not do anything. If it does, the planning layer will consult its plan library and build plans to complete the task. When new precepts or messages from other agents arrive, an *update* function is executed to analyze the new information and verify that the plan is still valid. If adjustments are necessary, replanning will be done. Tasks can either be provided by other agents or the task manager, as discussed in Section 5.3.

This layer will also reflect on the robot's battery level. If it at any time determine that it does not have enough battery left to complete its current task, it will abort the task, and instead work towards getting its battery recharged by assigning itself a CHARGE task. This introduces the problem of having the ability of aborting tasks. While some tasks can just be aborted without any issues, the robot can be in a state that requires some cleanup, hence a need for the agent to build a plan to get back to a free state. This deliberation is detailed in Section 5.4.4.

5.2. Communication

When an agent moves an artifact in the environment, other agents have no way of knowing it has been moved without perceiving the location. This creates *false beliefs* in the agent's model of the world. Agents can minimize these false beliefs by informing each other of the changes they make and observe in the environment. To do so, the agents must have the ability to communicate with each other. This is done through a set of predefined messages, that allows the agents to *announce* when they are moving artifacts or need to access a resource in the future, or *request* for other agents to move or help finding certain artifacts. At the same time, this system is also used to decouple the simulation engine from the agents, by having each of the robots publish their state through a channel. An agent can then subscribe to a robot's channel, thereby receiving updates about the given robot's state.

Each of the agents are running as an independent process, and in the real world likely

on their own machine, which means they must be able to communicate across machine through the internet. One of the challenges with such as system is to ensure reliable communication. A naive solution would be to just send the message and assume it is received. Without any special handling, the message might be successfully sent from one agent, but never received by the recipient agent. Hence, the message system should ensure that messages are always successfully delivered. Another desired property for the message system to have, is that messages send to multiple agents should arrive in the same order to every agent. Note that it does not require that messages send from two different agents are received in the same order they are send, only that they are received in the same order by every agent.

To solve these problems, a queue-based message system is used. Each agent have their own queue of messages, which everyone else can put messages into. When an agent A want to send a message to agent B, it will transfer the message over the internet, a controller at B will receive the message, put in into its queue, and send back an acknowledgement of receiving the message. This protocol is illustrated in Figure 5.2.

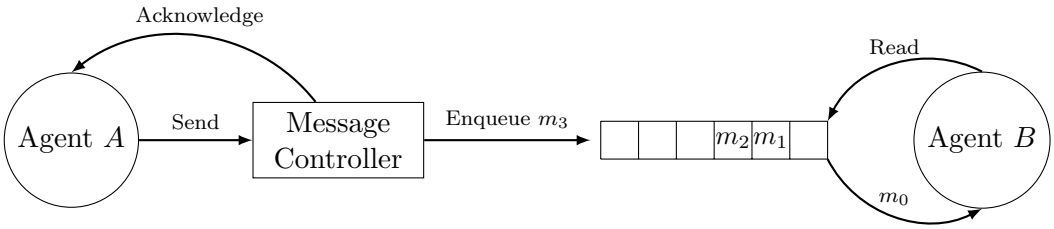


Figure 5.2.: Sending a message from one agent to another. Agent A will transmit its message over the internet, which will be received by a controller at B. The controller enqueue the message and send an acknowledgement back to agent A. Agent B reads messages from the other end of the queue.

This solves the problem of communication between agents, but not public announcements to every agent. The MAS need to ensure that messages being broadcasted all are received in the same total order as defined by [**?**], such that every agent agree on which order they can access artifacts. One solution to this is to use a central *exchange*, where agents can send messages to, which will then handle the distribution to every agent in the system. The messages are thereby only sent to one receiver, and whichever order the exchange receives the messages in will be the order in which the messages is redistributed to every agent. An example of this is illustrated in Figure 5.3, where three agents sends an announcement to the exchange at the roughly the same time. The exchange receives the messages the order m_1, m_3, m_2, and enqueue each of the messages to its own queue. It will then transmit the messages in the same order to every agent in the system.

5.2.1. Messages from robots

A subset of the messages define the data that are transmitted between the robot and its agent. This consists of the RobotState, Path, and RobotAction messages. All of

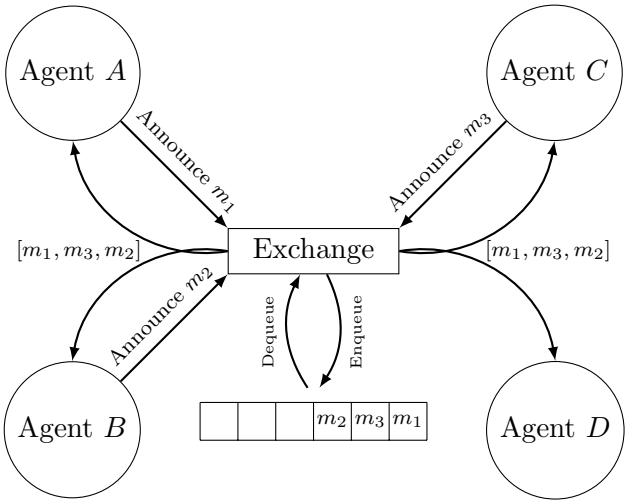


Figure 5.3.: Illustration of how announcements are sent to all agents in the system. Any agent can choose to send an announcement to the exchange at any given time. The order in which the messages arrives at the exchange in, is the order that the messages are send to each agent as well. Note that as before, each agent does have their own controller and queue to ensure messages are delivered.

these messages are received by the robot's message controller, which can interpret them and ensure they are handled correctly.

RobotState message This message contains the robots current state and sensor information corresponding to $\mathcal{R}$ in the agent's model $\mathcal{M}$. The robot publishes this information at a certain intervals to its subscribers. Each message consists of the following elements:

- The *position* of the robot in $\mathcal{G}$.
- The *battery* level of the robot between zero and one.
- The robot's *state*, which can be either free, error, charging, holding object, or whether it is in the process of picking up an object (its actuators are working).
- An *error* flag for whether the robot is able to follow the latest actions given by its agent. This is used to force the agent to replan.
- A set of the objects O around the robot, perceived through its sensors.

The four first elements of these messages are used to directly update R in the model layer by overwriting any old values. The sensor objects O are analyzed by the modeling layer, where every artifact $a \in \mathcal{A}$ is used to update φ with their new locations, while the remaining objects is stored directly in $\mathcal{M}$, i.e. $S \leftarrow O \setminus \mathcal{A}$.

RobotAction messages The RobotAction message is used by the agent to make the robot execute a specific action. The supported actions are to *pick up artifact* and *drop artifact*. These actions has a direct effect on the environment and the robot, and the resulting state of the action will be reflected a later *RobotState* message. Actions might not complete right away, so the agent will attempt the action several times before trying an alternative plan.

Path Message The last message which the robot can understand is the Path message, which is used to inform the robot of how it should move around in the environment. This message only consists of a path, i.e. a list of locations ($\langle p_1, p_2, \ldots, p_k \rangle$), which the robot will then visit in order. This message thereby adds an abstraction to the underlying forward, backward, and turn actions, which is handled an intermediate controller to the robot. If the robot at any time is unable to follow the received path, it will inform the agent through the *error* flag in the RobotState message, as described in Section 5.2.1.

5.2.2. Inter-agent communication

Communication between agents are used for informing each other about their intentions and changes to the environment. All of the following messages are sent as public announcement to every agent in the same workspace, unless otherwise specified.

Path message The Path message from above is reused to announce which path the agent currently intent to follow. It is broadcasted each time the agent sends the message to the robot to keep everyone as updated about its intentions as possible. When the message is received by an agent, it is used to update the model M by changing ϱ. The new function ϱ' will be $\varrho(a) = p$, where a is the agent that made the announcement, and $p = \langle p_1, p_2, \ldots, p_k \rangle$ is the path in the message.

When an agent receives a Path message from someone, it will therefore assume that its current position is at the first location of the path. As time progresses, the agent will update ϱ by removing the head of the path at fixed intervals, corresponding to the time it takes to traverse one location. While this does not give a perfect model for where other agents are, a threshold around the current believed location of an agent is added when searching to counter this. Whenever an agent makes adjustments to its plan, it will also announce and update everyone else, giving other agents a new up to date plan of where the agent will be.

CellInfo message The CellInfo message is used to coordinate usage of a specific location. It is used to announce an intent about what the agent will do at the given location. For example, if an agent a need to charge, it will find an appropriate charging station, and announce its intention to use this location. If other agents before it has sent announcements with intentions of using the same location, agent a will be aware and know which agents are using the location before it. This means if other agents are

already using the charging station, agent a will listen for when the last agent that send an intention of charging before it is done charging, before actually starting its plan to charge.

Artifact message Similar to the CellInfo message, the Artifact message is used by agents to inform others about changes it is doing to artifacts in the environment. When picking up or dropping artifacts, this will be announced, which other agents uses to update their model of the environment with the new location of the given object. More precisely, when an message for dropping picking up an artifact is received the model gets updated with $D' \leftarrow D \setminus \{a\}$, and similarly when a drop artifact message is announced, the model is updated with $D' \leftarrow D \cup \{a\}$, where a is the artifact from the message in both cases.

It is also used for coordinating intentions about using artifacts. Agents will announce that the need a given artifact, which tells others that they should not try to interact with it. This is done by marking the artifacts directly as $locked(A, X)$, describing that the artifact A can currently only be used by the agent X. If artifacts is already in use, agents can announce that they would like to access the artifact at some point in the future by adding themselves to the artifacts queue, i.e. announcing $queue(A, L')$ where $L' = \langle L, X \rangle$, X is the agent and A is the artifact.

Request and Reply message The Request is used to ask for information or ask other agents to move. A Reply contains the information requested or an acknowledgement accepting or rejecting the move request. Agent might be unable to find certain artifacts within their own model of the environment and can thereby ask other agents if they have the given information. For the second case, agents can request help to have other agents interact with certain objects, or simply move out of the way. One situation where this is necessary is when an agent have completed its task and is now idling it a position required by another. Agents either must wait until a new task arrives for the idling agent, causing it to move, or they can send a request for it to move out of the way.

Practically, agents receiving a Request message will search their model $\mathcal{M}$ to see if they have the information asked for and send a Reply message if so. Receiving a Reply mostly consists of updating D and φ in the model, as this is the most common use case, however cases where the entire model is overwritten based on the reply is also seen, for example when a agent process has failed and needs to be re-instantiated during execution.

5.2.3. Workspaces - Limiting communication to local agents

Coordination and communication in MAS can prove to be a bottleneck and becomes increasingly challenging as the number of agents increase. As discussed in Section 3.1.1, the LRA* algorithm to solve the MAPF problem increases exponetially in the number of agents. When planning, every agent in a MAS has to formulate their own plan, and

the coordination with every other agent has to be done, to ensure plans are compatible. However, it can be the case that agents' plans does not overlap, and therefore coordination can be avoid. In our case with the MiR robots, coordinating between distant robots or robots in different rooms could be unnecessary.

A strategy for limiting coordination in multi-room environments have therefore been created. Instead of coordinating with every robot in the environment, coordination is limited to the room which the robot is currently in, thereby limiting the complexity of the coordination problem. Robots might still need to move between rooms to complete their tasks, which they can achieve using the JoinRoom(a, r) and LeaveRoom(a, r) announcements. The two messages will update the model by $\omega'(a) = r$ and $\omega'(a) = \emptyset$ respectively. The agent will plan for the robot to move to the location of a door between its current room and the desired room, announce to everyone in its current room that it is leaving, and then announce to everyone in the new room that it is joining. These two announcements will inform other agents whether or not they should inform the agent about changes in the given room. After the agent has entered the new room, it can coordinate with the agent within this new room to achieve its goal.

Rooms in the environment are defined from doors defined in the user specified map. Agents translates the map into a grid-based graph, modelling the environment, where each cell in the graph belongs to exactly one room. When building this model of the environment, each reachable cell is traversed to determine which room it belongs to. When a cell containing a door is encountered, it is known that a new room will be on the other side.

5.3. Distribution of tasks

To control which agents are working on which tasks, a specialized *task manager* (TM) agent have been implemented to track free agents and the tasks that is currently being worked on. It does not create the tasks but are only responsible for distributing tasks between agents. It is built as such to allow other systems to integrate and control what the tasks are. The simulation includes two different methods for generating tasks, given in Section 4.3. The controller acts as a special type of agent, which communicate through the same method as all the other agents, but with a specific set of messages.

Tasks can be added to the TM by sending a AddTask message containing the given task. The TM will then determine which agent is best suited to carry out the task. Currently, all robots have the same tools, enabling everyone to work on all tasks. Two approaches have been considered and tested: a centralized approach, where the TM builds its own model of the world and decides who, and a decentralized approach using an auction system to determine the best agent. When the decision has been made, an OfferTask is send to the selected agent, which can then choose to either response with a AcceptTask or a RejectTask message. If accepted, the agent will incorporate the task into its current plans, and otherwise the deliberation process of choosing an agent will restart. The TM is also able to send a CancelTask message to an agent, which

will cause the agent to abandon its task.

5.3.1. Auction based task distribution

To allow each agent to consider if it wants to do a task, an auction system is used to determine the best available agent. This is done by having the TM announce AUCTIONTASK to every agent. Each agent will then consider the task announced, consider if it is available to and capable of complete the task, and using different heuristic to determine the cost of completing said task. This cost varies between tasks but is in general based on the distance to artifacts needed for the task. The bid is sent to the mission control with a BIDONTASK message.

If the agent is free, it will always give a bid for the task, if it believe it can complete it. If the agent belief that it cannot complete the task, it will not give a bid. Agents working on other tasks might also consider if it is possible to merge the auctioned task with their own current task, which might allow the agent two complete both task with limited overhead, as discussed in Section 5.4.3. Here the agent will create a bid based on the estimated extra cost of merging the task into its current.

An issue with this approach is that it might be the case that no agent believes that it can complete the task, hence the TM will receive no bids. However, the TM can either wait and try to auction the task again at a later time, or if important or time critical, force the task upon some agent. No real prioritization of the tasks generated by the simulator are currently made, and tasks are therefore only prioritized based on their published time.

5.4. Planning and coordination

One of the central parts of a multi-agent system is to ensure that the agent can work together in a shared environment. The basic requirements is that the agents can move around without colliding with each other and can access different resources when required.

5.4.1. Search strategies

To find paths and objects in the environment, different heuristic searches is applied, depending on what the agent is looking for. The underlying search algorithm however, remains similar between the different kinds of searches that are being done. It works by using a hash-set for *visited* locations, along with a priority queue (*frontier*) containing an ordered structure over the search nodes based on a given *heuristic*. The algorithm is based on the general GRAPHSEARCH from [**?**, p. 77] and outlined in Algorithm 1. The search nodes contain a location $p \in \mathcal{G}$, along with a reference to its parent. In the outline of the algorithm, the parent will always be assigned to the *current* variable when adding a node to the $frontier$. When a node satisfying the *goal criteria* is found, the path can be extracted by traversing the nodes' parents until the root is found (the EXTRACTPATH

function). The HeuristicSearch has been designed as a general higher-order function, which can take several parameters that allow for varied search strategies. These are:

- The agent's current model $\mathcal{M}$ of the environment.
- (x, y)-coordinate, the position from where the search should *start*. This will be transformed into a node within the search algorithm and added to the frontier before beginning the search.
- A *goal criteria* function ($node \mapsto boolean$), mapping a search node to whether it contains a goal state or not.
- The *heuristic* function ($node \mapsto integer$) that evaluates how good the given node is, the lower the better. At each iteration of the search, the node with the lowest heuristic will be explored.
- A function ($node \mapsto \langle x, y \rangle$) to find the *neighbours* to a given node. This should return all the valid neighbours for a given node, taking only the static elements of the world into account.
- A boolean flag for whether the search should consider *waiting*. This defaults to $false$. If this is true, each location will be added to the frontier again after being explored, with a penalty to its cost. Note that the heuristic function needs to take this into account to avoid continuing to analyzing the same location repeatedly.

Algorithm 1 HeuristicSearch

```
function HeuristicSearch(M, start, isGoal, heuristic, neighbours, wait)
    visited ← ∅
    frontier ← priority queue instantiated with start
    while frontier ≠ ∅ do
        current ← minimum element in frontier
        if isGoal(current) then
            return ExtractPath(current)
        visited ← visited ∪ {current}
        for all neighbour ∈ neighbours(M, current) do
            if neighbour ∈ visited then continue
            frontier ← frontier ∪ {neighbour}
        if wait then
            frontier ← frontier ∪ {current}
    return failure
```

Note that if the frontier is empty and no result have been found yet, the algorithm will return a failure to inform that no solution is found. For practical use, a time limit is also introduced when searching with the *wait* flag, as there is no guaranty that the algorithm will terminate. This also means that if the heuristic does not penalized enough, the range of the search in 2D space will be limited (e.g. when using a simple breath-first

search). The details of how the HeuristicSearch is applied will be given later together with its use cases.

5.4.2. Tasks and plans

Tasks describes what an agent can work on, along with the set of plans for completing the given task, which allows for easily extending the types of tasks that an agent can handle. The library of plans to solve these tasks are created using a hierarchical planning approach. A task itself can be abstract with no concrete actions that a robot can execute but consist of multiple more or less concrete steps. This allows for more complex task to be build from simpler ones.

The agents are designed to understand an abstract task type, that enforces a protocol with the following properties. Each task type has to describe a *start*, *update*, and *abort* function, taking an *agent* and model $\mathcal{M}$. The *start* function is called exactly once when the agents first start considering the task. Its purpose is to verify that the preconditions for completing the task is satisfied. The *update* function is called one or more times while the agent is working on the task. This function is the body of the plan and is responsible for sending actions to the robot or communicating with other agents. When it verifies that the postconditions of the task is satisfied, it will signal that the task has been completed. The *abort* function is called if the agent should no longer consider the task. The goal for this function is to ensure that the robot and environment is in a state similar to before the task was started. Note that this does not mean that the state after aborting has to be equivalent to before the start of the task.

PickUpArtifact and DropArtifact Some concrete tasks are defined, which closely correspond to the actions available for the robot, and will directly execute said actions on the robot. The PickUpArtifact and DropArtifact task are two of such. The first is handled by having the agent repeatably tell the robot to pickup the object at its current location, until it receives a message from the robot that it actually carry the given object. The second works in the same manor, but instead sending the drop object action. As the actions might not be executed right away, the robot will simply repeat the action until the robot's state changes. Both of which takes the type of object the robot should pick up, in case there are multiple options to choose from.

GoToLocation Another atomic task is the GoToLocation(l), which adds a goal to reach the given location and contains a general plan for how to reach it. This already becomes more complex, as the agent needs to assure that the given location it reaches is free some time in the future. The outline of the algorithm for doing so can be found in Algorithm 2. Note that the FindQueueLocation function is applying the HeuristicSearch algorithm to find an appropriate location to wait at.

The task will start by enqueueing the agent for the given location. This is done by using announcements, such that other agents are informed of the agents intentions. While enqueued, the agent will move to a position close to the goal location, and wait until the

queue to the location becomes empty. The agent will know this from announcements from other agents. It will then announce that it is entering the location and that it is dequeuing. As long as its precepts does not tell the agent that anything is blocking it, it can then enter the location.

The task itself interacts with the agents by setting its *Goal* property, which is a (x, y)-coordinate of where the agent should move to. The agent's planning layer will handle the path-finding. This allows the agent to skip the planning if it already have a plan and has not perceived anything that should make the plan fail.

Algorithm 2 GoToLocation($goal \in \mathcal{G}$)

```
artifact ← φ(goal)
position ← p ∈ R
function START(agent, M)
    ANNOUNCE(agent, ENQUEUE, artifact)
function UPDATE(agent, M)
    if position = goal then ANNOUNCE(agent, UNLOCK, artifact)
    else if agent is first in queue(artifact, L) then
        agent's goal ← goal
        ANNOUNCE(agent, LOCK, artifact)
        ANNOUNCE(agent, DEQUEUE, artifact)
    else if agent ∈ L where queue(artifact, L) then
        agent's goal ← FINDQUEUELOCATION(M, agent, goal)
function ABORT(agent, M)
    if agent ∈ L where queue(artifact, L) then
        ANNOUNCE(agent, DEQUEUE, artifact)
    if locked(artifact, agent) then
        ANNOUNCE(agent, UNLOCK, artifact)
```

Wait The last concrete task that have been defined is the WAIT task, which makes the agent wait until some condition has been fulfilled. This task is used when some external entity is interacting with the robot, and the agent should therefore not move until it gets a signal that it is clear to do so. This could for example be when a human is unloading items from a shelf that the robot is carrying. For simulation purposes, the wait task uses a timer to simulate something interacting with the robot for some amount of time.

Finding artifacts in the environment

Finding objects in the environment is encapsulated in a FINDARTIFACT task. Note that it is assumed that the given objects that are being searched for is always preset in the environment, however its location might be unknown or it can be in use by another agent This task can be quite simple, if the agent's model $\mathcal{M}$ contain a location of the given artifact, but can become more complex if it does not or if it contain incorrect beliefs.

Before introducing unpredictable elements into the simulation, the agents could assume that their beliefs about the world where always correct. Only other agents could move around objects, and when doing so, the agent would always announce that it is move the artifact (assuming announcements are reliable). This made finding artifacts quite simple, as the agent's view of the world could simply be searched using the heuristic search algorithm defined in Section 5.4.1. A function doing so can be found in Algorithm 3.

Algorithm 3 FindArtifact

```
function FindArtifact(𝒲, start, type, id)
    function isGoal(node)
        artifacts ← φ(position(node))
        for all artifact ∈ artifacts do
            if artifact is type and artifact.id = id and ¬∃x locked(artifact, x) then
                return TRUE
        return FALSE

    function heuristic(node) return ManhattanDistance(position(node), start)

    return HeuristicSearch(ℳ, start, isGoal, heuritic)
```

The agent's model $\mathcal{M}$ and *start* position is directly used to begin the search. In this search, a node is considered a goal if it contains an artifact that is of the type searched for, the artifact's id matches the searched id, and that it is unlocked. As agent does not have any idea of which direction is should search in, and it is interested in finding the closest possible object, the heuristic can be implemented as a *breath-first-search* using the Manhattan distance between the node and the start position.

However, as artifacts can be moved without the knowledge of the agent, the task also need to contain steps to handle this. There are two cases to handle: the agent might not be able to find the artifact at all, or it might find it but upon reaching the found location, the artifact is not there. In both cases, agent begin two approaches to solve the problem. First, the agent will try to ask other agents if they are aware of the location of the given artifact using a Request message. It might be the case that other agents have observed the change in the environment and can therefore reply with the correct location. If this is not the case, the agent will begin a ScanEnvironment task, which will make the robot move around and use its sensors to detect artifact. It will continue to do so until a matching artifact is found. This can be quite slow it large environments, but a possible extension is to publish this request to other agents, using the shared sensor information from all robots to look for the artifact.

Storing artifacts

Some artifacts, such as shelves, need to be stored again after having being used by an agent. A simple solution to this problem would be to always store the artifact in the

same location as it was retrieved from. However, as the frequency in which an artifact might need to accessed can change, it is preferable that the artifacts are not locked to any position. Instead, the agents can rearrange artifacts based on when they expect to need them the next time, thereby optimizing future tasks.

For the general case where the agents have no way of estimating when an object is required next, they will simply try to find the closes location to drop it of, thereby freeing themselves to solve other tasks. This is done in a very similar way as the FindArtifact, where first a search within the agent's model of the world will be done, and failing that, a scan of the environment will begin. The call to the HeuristicSearch function is also the same, except for the goal criteria. Here, a node is a goal if it is of type Storage and it currently does not have any artifact nor is locked by any other agent.

Delivering shelves and products

Planning for delivering artifacts to given location is started from a abstract plan, which is then filled out continuously as needed. The planning tree is illustrated in Figure 5.4. The planning start from the abstract DeliverAndStoreShelf plan, which is divided into several steps. The first and last steps are more complex and can be solved by other abstract plans. The three steps in the middle are three of the concrete tasks defined earlier, corresponding to actions that the agent can carry out directly by sending commands to the robot. Note that the FindArtifact and FindStorage tasks both contain a step where they first tries to search their model of the world to find their goal, which if unsuccessful will have the agents attempt alternative strategies to fulfill their goal. These strategies are described in Section 5.4.2.

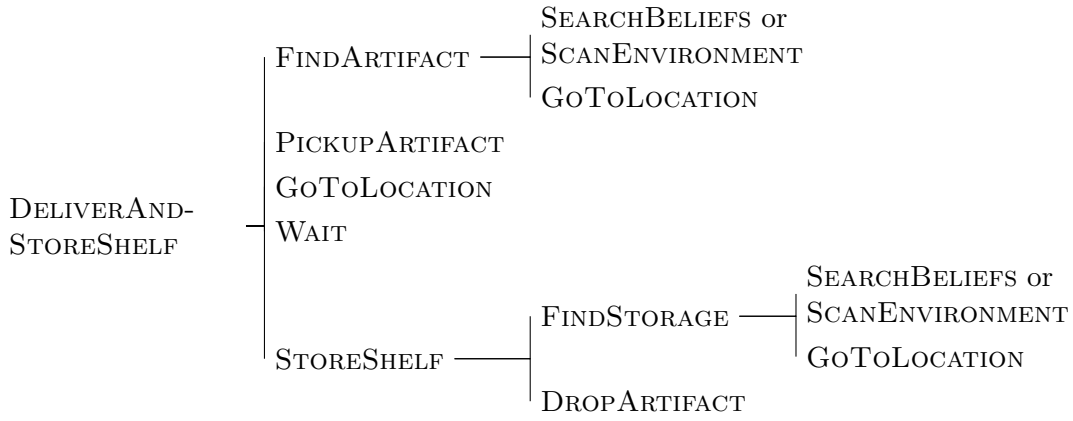


Figure 5.4.: Overview of a DeliverAndStoreShelf. Each step consists of increasingly concrete tasks, until a concrete plan is formulated. The agent can then execute the plan by executing the leaves from top to bottom.

5.4.3. Merging tasks

Agents should be able to have multiple goals at the same time, which corresponds to multiple tasks. To have this capability, the agents need a method to combine or merge several tasks into one large plan. A naive solution for this is to simply build a list or queue of plans, concatenating new tasks to the end. The agent then simply starts from the beginning of the list and work its way through all the tasks.

However, this is not always the optimal solution. Consider for example an agent currently working on a task to carry a shelf to a given location. A new need for this shelf at a different location appears. The simple solution would require the agent to first deliver the shelf to the original location, then find a storage location for the shelf, dropping it, only to find and pick up the same shelf once again. Clearly the last steps are unnecessary. Instead, the agent should be first go to the original mission's location, then the next, and then continue on with storing the shelf. This enables the agent to complete the new mission with lower cost than the naive approach.

This can in general be solved by analyzing the pre- and postconditions involved in carrying out the new tasks, and searching for places in the agent current plan where these are fulfilled. In the example where two DeliverAndStoreShelf tasks are being merged involving the same shelf, the two first and last (FindArtifact, PickUpArtifact, and StoreShelf) are the same and will have the same effect on the environment. The remaining steps (GoToLocation and Wait) are different and will have different effects on the environment. To simplify the algorithm, it is only considering injecting all of the new steps at the beginning or end of the current task. A heuristic is used to evaluate which is most cost efficient.

In the example, the new goal location could be on the way to the current goal location, making it more efficient to visit the new location first. On the other hand, if the new location is far off, then it will be more efficient to inject the new goal after the current.

Note that the merging of tasks might occur when part of the plan has already been executed. This is why using the effects of the tasks on the environment is useful for evaluating if the tasks can be merged. If the agent has already picked up the shelf, therefore removed the first steps of the plan, the merge algorithm can still find that the preconditions for the remaining steps is fulfilled and can merge the tasks into one plan as before.

5.4.4. Aborting tasks

The environment around robots can change at any time, which can cause goals to become unobtainable. Tasks might no longer be necessary to carry out, or the robot's battery might no longer contain enough energy to complete its task. Agents therefore need to consider if and when goals are no longer feasible, and they should abandon their current goal and plan. This introduces two challenges: the task might require some cleanup before it can be abandoned and finding a new agent to carry out the task, if it still needs

to be completed.

For the first part, the agent's plan might have changed the environment in some way, which now makes it impossible for other agents to complete their tasks. The agent will therefore needs to build a new plan to revert the state of the environment to a state that other agents can continue from. For example, if a robot is currently carrying an artifact, it must safely store this somewhere in the environment, where it can later be accessed.

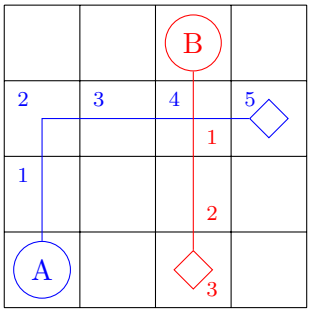
Secondly, the agent must inform other agents in the MAS that it was unable to achieve its goal, such that a new solution to achieve the goal can be found. The agent will first announce that it is no longer able to complete the task, along with the current state of the task. This is done with the AbortMission message. From there, the decision process of choosing which agent should work on the task can be started over again, as discussed in Section 5.3.

5.5. Path finding

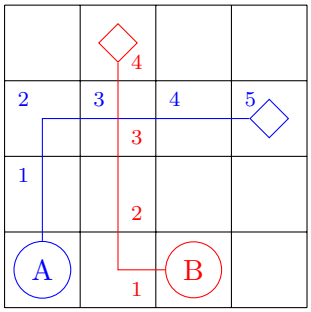
Central to the MAS is the ability for the agents to plan routes with in the environment that does not conflicts with each other. The general MAPF problem is introduced in Section 3.1.1, where an extended version is applicable here, where new goals are created throughout the life-time of the system. Formally, each agent a_i will not have just one goal, but a sequence of goals, denoted g_i^x, which are not known at the beginning of the problem. Note that the sequence of goals can have different lengths for each agent, and that it is possible that there exists a time t where no goal is known for a given agent. The goal for the algorithm will therefore be to find a set of paths for the agents, that takes them from their current position $\pi_i(t)$ at time t to their current goal g_i^x. Each time a new goal is published, it can be seen as a new instances of this MAPF problem, where the goals are g_j^{x+1} for agent a_j and g_i^x for all a_i where $i \in [0, k] \wedge i \neq j$. Here it is worth noting that for all but one agent, their goal stays the same, hence the path found in the previous iteration is still valid.

As described in Section 5.1.2, the world is modelled using a graph $\mathcal{G}$, which contains all of the vertices in the graph representing (x, y)-coordinates. Edges in the graph are not stored but computed on demand. Because each vertex can have at most four edges, corresponding to the four adjacent neighbours, they can be found it constant time. Vertices therefore only consider five actions: *north*, *south*, *east*, *west*, and *wait* (staying in the same location). The path can afterwards be converted to actions that the robot can carry out. However, the existence of an edge is also dependent on the time, i.e. at time t the neighbour *west* might be occupied by another agent, and therefore unavailable as an action, while at time $t + 1$ the agent could have moved, and the neighbour *west* is available. The search through the graph is therefore done as a three dimensional space-time search.

The strategy used in this MAS is similar to the Cooperative A* which Silver describes in [**?**], using known paths from other agents to avoid conflicts. The algorithm for each



(a) Agent A finding a valid path overlapping with agent B.



(b) Invalid planned paths between agent A and B.

Figure 5.5.: Illustrations of path-finding.

agent can be outlined as:

1. **Local**: Perform local A* from current position $\pi_i(t)$ to the current goal g_i^x. Known paths from other agents is used to consider which locations are available in 3D space-time.
2. **Merge**: Consult known plans for other agents to verify that no other agent has published a path that creates a conflict. If a conflict is found, the agent will to restart its local planning.
3. **Announce**: Publish the path to other agents and begin execution.

The algorithm is executed several times throughout the life of the MAS. An agent will first execute the algorithm when it receives its first goal. When this goal is reached, the algorithm will be executed again if or when a new goal arrives. If the robot encounter any obstacles on its path, it will force the agent to consider replanning. First the reactive layer of the agent will analyze if the obstacle will move in the near future, choosing to simply wait a few actions. If this is not the case, the agent's planning layer will execute a rerun of the path-finding algorithm. An attempt to repair the old path is done; when a location after the obstacle from the old path is explored, it will analyze the remaining part of the old path for conflicts. If none are found, it will return the detour around the obstacle plus the remainder of the old path.

Figure 5.5 illustrates two examples of path-finding with two agents. In the example, agent A must reach its goal at $(3, 2)$. In Figure 5.5a agent B has already planned and announced its path to reach its goal at $(0, 2)$. The number indicate the time-step in the future where the agent will occupy the given cell, top-left corner for A and bottom-right for B In the first example, even though the paths overlap at $(2, 2)$, no conflicts happen as A will be at the location at $t = 4$, while B uses the cell at $t = 1$. In contrast, Figure 5.5b illustrates an example where B's goal is at $(1, 3)$, and where A is unable to find a path does not create a conflict. The current path suggested by A causes both agents to be at $(1, 2)$ at $t = 3$. Agent A must therefore replan to find a valid path (e.g. A could WAIT

one time-step before following the illustrated path).

One of the issues with this algorithm is that an agent might be unable to find a path to its goal which does not cause conflicts. Other agents can earlier have commit to a path that now always will cause a conflict. To handle this, agents have the ability to ask others for altering their path, making them move and resolve the conflicts. This does help in many cases, however agents are not always able or willing to alter their path. This is often seen as deadlocks, where two agents want to pass by each other, with neither being willing to move.

The benefit of the simplicity of this algorithm is that the path-finding problem becomes fast, as it only needs to consider one agent. The branching factor is $b = 5$, hence the A* search can be executed in $O(5^d)$. Checking for conflicts has to be done for every agent, and can for each be done in linear time of the length of their planned paths, $O(k \cdot |P|)$ where P is the path. As the algorithm can potentially run forever, a limit for how many states should be explored is used to terminate it.

5.5.1. Weighed graph search

The A* algorithm uses a evaluation function f to evaluate each node n when exploring the graph. This is defined as:

$$
\begin{aligned}
f(n) &= g(n) + h(n) \\
g(n) &= \text{cost to reach } n \\
h(n) &= \text{estimated cost from } n \text{ to } g_i^x
\end{aligned}
$$

The standard algorithm uses $h(n) = \textsc{ManhattanDistance}(n, g_i^x)$ as the heuristic, as the Manhattan distance is an admissible heuristic for the grid-based world that are close to the actual cost. The cost to reach n is calculated as $g(n) = |P|$ where P is the path traversed from the initial position to n.

To guide the search, weights can be added to the underlying graph. One strategy for a general grid-based graph is to penalize movement in certain directions based on the row and column. This is achieved by setting $g(n) = \textsc{WeighedCost}(n, \mathit{penalty})$, given in Algorithm 4. Here, the *west* action is penalized in even rows, the *east* in odd rows. This will encourage agents to travel in the same direction in the same rows and columns, minimize the chance for head-on collisions. The *penalty* parameter can be adjusted to set the willingness of the agent to travel against the desired stream.

Using the WeighedCost the A* algorithm no longer find the shortest paths between locations. A simple example is trying to reach a location that is one step in a penalized direction, e.g. *north* in an odd column. If the *penalty* parameter is four or above, the action *north* will have a cost of four, while the action sequence $\langle \mathit{west}, \mathit{north}, \mathit{east} \rangle$ has a lower cost of three.

Algorithm 4 WeighedCost

```
function WeighedCost(node, penalty)
    parentCost ← WeighedCost(parent(node), penalty)
    if position(node) is even column and action(node) = south then
        return parentCost + penalty
    if position(node) is odd column and action(node) = north then
        return parentCost + penalty
    if position(node) is even row and action(node) = west then
        return parentCost + penalty
    if position(node) is odd row and action(node) = east then
        return parentCost + penalty
    return parentCost + 1
```

5.6. Summary

This chapter explores how a MAS can be designed for the MiR robots. The layered architecture of the robot agents is described, with the responsibilities for each of the reactive, planning, and modelling layer. The reactive layer allows the agent to make fast decisions about changes, while the planning layer can build plans using the agent's model of the environment to achieve long term goals. A communication language is designed to be used by the agents to coordinate their plans and beliefs about the environment, along with how information sharing can be limited to nearby agents. Another agent is designed to distribute tasks, experimenting with both a centralized approach and an auction system.

The plan library for the agents are given, describing how agents can go from abstract plans to concrete actions with the use of hierarchical planning. These plans include where coordination is done at the different levels. A look into how tasks could be combined, allowing a robot that has already obtained an artifact to adapt their goal to solve tasks faster. As tasks can become impossible for an agent to solve, a method for aborting tasks is described, ensuring used artifacts is made available to others and that the task can be redistributed to another agent if necessary. Lastly, an algorithm for solving the MAPF problem is described, which is capable of solving many instances of the problems created by the simulation tool, while still failing to handle some conflicts.

CHAPTER 6

Evaluation and discussion

This chapter will evaluate the MAS system designed in Chapter 5, using the simulation tool to create different scenarios. Here the system will be evaluated on several parameters, including how fast robots are able to solve tasks, the distance they travel, and how active agents are. This is followed by suggestions for further work, discussing what is necessary before applying the MAS to physical robots and other suggested changes to improve the system.

6.1. Experiments and results

To analyze the MAS that has been designed, several experiments has been designed using the simulator to execute the tests. The focus have been on testing several factors of the system. First, how the system behaves with an increasing number of robots sharing a small space, measuring global output and the number of conflicts. The weighed graph search algorithm will be tested with different penalty values and compared to the unweighed search algorithm. How the system handles unpredictable elements in the environment is tested, analyzing how well the agents can respond and replan. A test of how many robots the system can scale to will also be done.

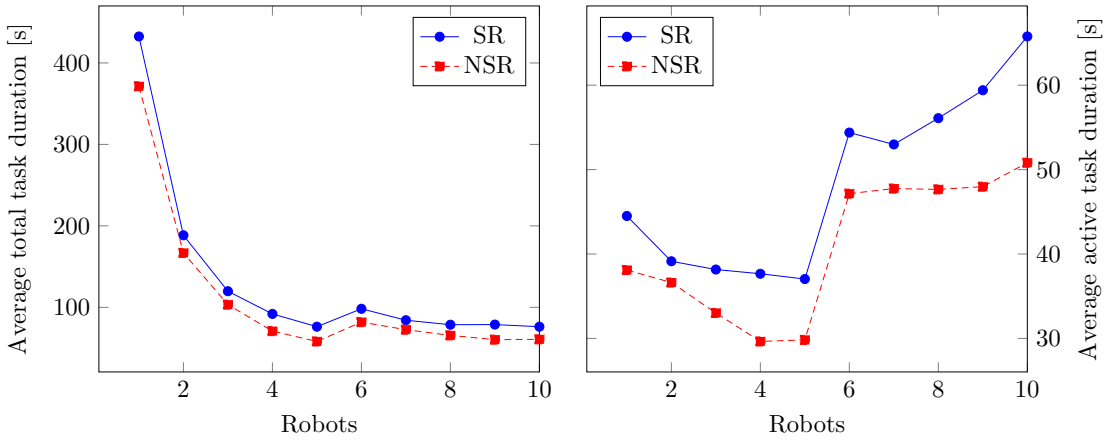
The explanations and analysis of the experiments are given in this section with the relevant data. The raw data can be found in Chapter B. The MAS will be evaluated on several factors. The *activity* of the agent is measured, together with how much *distance* the robots travel, corresponding to the length of the solution to the MAPF problem. Together with this is how fast the system can complete tasks. Here are two measurements included: the *total task duration*, measured from when a task has been published till its completion, and the *active task duration*, measured from when a agent is assigned the task to its completion.

6.1.1. Confined space

The goals of these test are to evaluate how well the MAS perform when space is limited, and secondly how the simulator can be used as a tool to find the optimal number of

robots to use in a given environment with a given load. Experiments with are done in a twelve by ten square environment with 48 unique shelves that can be moved around by the robots. Three goals are placed in the environment. To provide comparable results, three lists of 20 tasks have first been generated randomly, and then used to publish the same tasks at the same time points. All 20 tasks are published in the span of one minute. The lists is used to run three simulations for each number of robot, where the average of the three simulations are used to account for the variations between simulations. Starting with one robot in the environment, the amount of robots are increased until the problem becomes unsolvable or too many robots are idle. The problem is considered unsolvable if an agent is stuck in a conflict for one minute.

To provide a base line for how fast the tasks can be completed, the *non-space restricted* (NSR) case is first considered. This relaxes the problem such that robots can occupy the same space at the same time. Conflicts are thereby impossible, and every agent can simply choose the most direct route to their goal (the MAPF algorithm is altered such that the agents does not consider other agents plans). Note that artifacts are still restricted to only one robot at a time, meaning that goals and shelves can only be used by one robot at a time. This is then used to compare the performance of the MAS in the *space restricted* (SR) setting.



(a) The average total task duration as a function of robots.

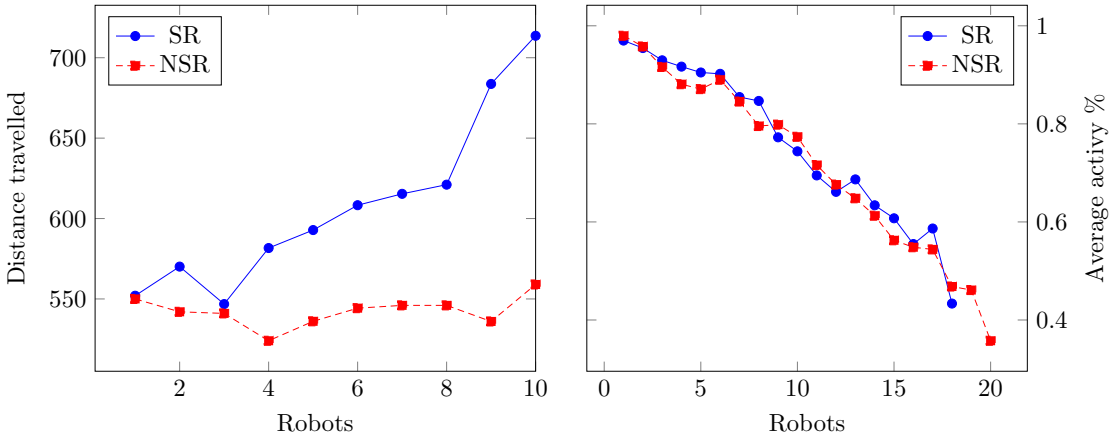
(b) The average active task duration as a function of robots.

Figure 6.1.: Comparison of the duration for completing task between SR and NSR.

Figure 6.1a shows that the average total time for the robots to complete all the tasks generally decreases with time, hence also the overall time to solve all problems, as would be expected. It is clear that both the SR and NSR simulations follow the same general trend, where the SR are slightly worse in all simulations. From Figure 6.1b it can be found that at five robots, the average time for a robot to work on a task increases drastically. This is because the goal locations becomes congested, forcing the robots to spend time in queue before being able to deliver their products. This however seems relatively constant for more than five robots in the NSR simulations, while it continues

to increase for the SR. Because multiple agents waiting in queue cause a higher density of robots in that area, more conflicts and replanning occur when entering or leaving the area.

However, for warehouse environments, keeping the robots busy are generally not the goal. Optimizing the total output is desired, which would require adding more workstations (goal locations).



(a) The total distance travelled by all of the robots in the simulation.

(b) Percentage of time agents are active as a function robots.

Figure 6.2.: Comparison of robot related statistics from SR and NSR simulations.

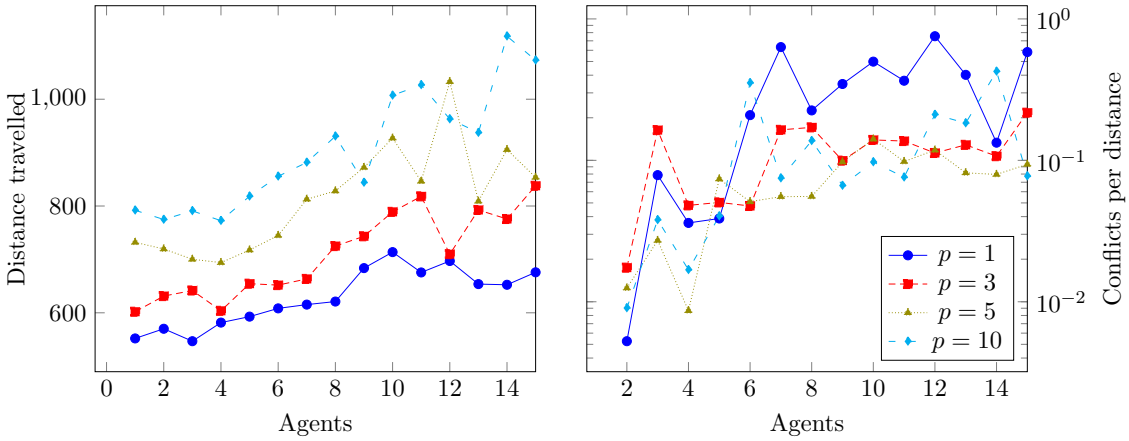
The distance travelled by the robots (Figure 6.2a) shows the largest difference between the two tests. The distance in the NSR simulations remains relatively constant with the number of robots, as agents can always take the shortest path as no conflicts can occur. The SR simulations instead show an increase in distance travelled with more robots. More robots causes fewer paths to be available, and increase conflicts causing agents to replan and take long detours to reach their goals.

Figure 6.2b shows the activeness of the agents themselves, i.e. the percentage of time they have an active task. Agents that choose to do nothing while having a task are still regarded as active, as adjustments to plans are often not necessary. The activeness of the agents decreases with the number of agents, where no significant difference can be seen between the SR and NSR simulations.

Overall the MAS performs similarly in both simulation types, showing that the MAPF problem is being solved. While the length of the solution is clearly worse in the SR cases, the overall duration to solve tasks are similar. Generally the average time to complete a task are slightly worse in the SR simulations, with the worst case at 16 robots being 85% longer.

6.1.2. Weighed graph

The weighed graph search described in Section 5.5.1 has been evaluated with different penalty p values ($p \in \{1, 3, 5, 10\}$). A penalty value of 1 is equivalent to the standard search. With a penalty of 3 agents will still prefer to go directly to adjacent cells, but prefer to follow the desired directions in longer routes. With 5 agents primarily consider going against the desired direction if they can avoid to wait, while a penalty of 10 should only be in worst case. The same environment and tasks as in the previous simulations are used. Results are shown in Figure 6.3, showing the distance travelled by the robots and the conflicts per distance.



(a) Distance travelled by all the robot with weighed graph search

(b) Conflicts per distance with weighed graph search

Figure 6.3.: Comparision between weighed graph searches with different penalty values

As expected, the distance travelled increases with the penalty value (Figure 6.3a) while the amount of conflicts per distance decreases (Figure 6.3b). Table 6.1 summarizes the differences between no penalty ($p = 1$) and the results from simulations with different penalties. Interestingly this shows that $p = 5$ performs better in terms of both distance and conflicts compared to $p = 10$, showing that being stricter about routes does not necessarily decrease conflicts.

Table 6.1.: Mean difference in distance and conflicts for weighed graph search compared to $p = 1$.

p	Distance	Conflicts	Duration
3	+12.10%	−38.56%	+7.81%
5	+28.63%	−63.71%	+26.76%
10	+43.68%	−38.96%	+22.49%

However, none of the simulations with weighed graph search improve on the duration. While less conflicts decreases the computational requirement as less replanning is necessary, minimizing conflicts is not a goal. The speed of which tasks can be solved is.

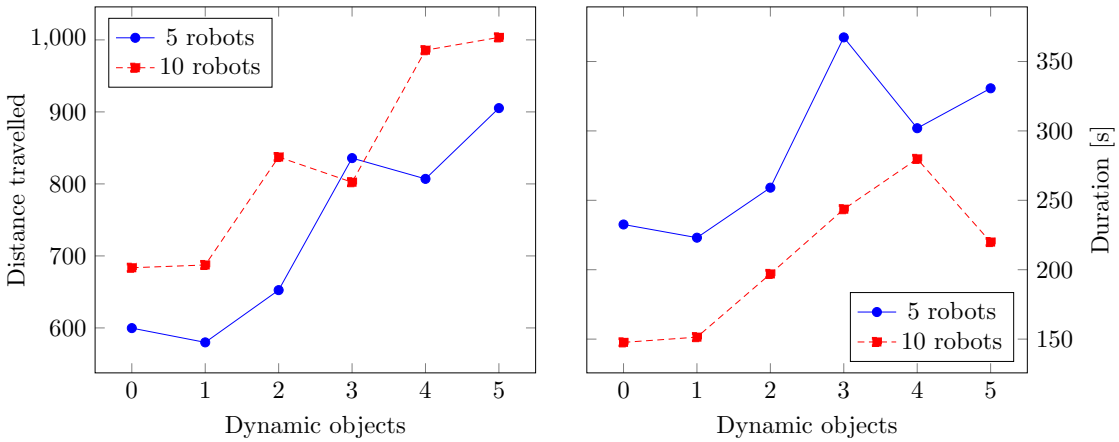
An issue found with this algorithm is that it can be forced to search in the wrong

direction. When nearby positions in rows or columns going in the desired direction is blocked, a very long detour might be required. As the search is done in both space and time, choosing the wait action in or near the starting position becomes more desirable than searching in the wrong direction. One suggestion to improve this is to penalize the wait action together with the edges in the graph, making waiting less desirable.

6.1.3. Dynamic elements

Dynamic, unpredictable elements in the environment can be difficult for agents to handle, as they have no way of knowing the element's next actions. As described in Chapter 4, the positions of these dynamic, unpredictable objects are not known to the agent unless they directly perceive them through their sensors.

In the experiments here, an increasing number of unpredictable objects are introduced into simulation. Instead of just following a random path, they have been designed to move randomly within certain path, making them occupy more similar position in the different simulations. The objects are set to travel most used positions based on the SR simulations from Section 6.1.1.



(a) Distance travelled by the robots.

(b) Duration to solve all tasks.

Figure 6.4.: Comparison of MAS in environment with dynamic objects.

Figure 6.4 shows the results from simulations with five and ten robots and an increasing number of unpredictable objects. While the MAS can handle the dynamic objects, both the time to solve all tasks and the distance travelled is increased. This is due to the agents have to replan more often and must take detours around the dynamic objects.

Adding more than five unpredictable elements to the simulation caused the MAS to fail often. The most observed case has been when no route could be found around an object. One approach to solve this problem could be to introduce random actions for the robots, trying to move them out of the way for the dynamic objects. An attempt to solve the conflict can then be done from a slightly different state, where the dynamic object has

possibly moved.

The modelling of the dynamic objects inside the agents are also rather simple. Currently, the model will only retain information about these object as long as they are within their sensors range. As such, they are thought as not existing as soon they cannot be seen. This means that agents can alternate between paths blocked by different objects, without finding a valid solution before one of the dynamic objects has moved. The model could retain the positional information about the objects longer, however it would quickly become outdated as the objects move. A degradation function over time could be used to determine when the information should be disregarded.

Another limitation of the MAS is that the agents does not share information about these objects. Agents could help each other by announcing which locations they currently observe to be blocked.

A heat map of the locations where conflicts occur can be found in Figure 6.5. From this, the most congested locations can be found to be at the openings of the roads between shelves. As agents would often have to move around inside these shallow roads to reach shelves, conflicts with other robots and unpredictable element would often appear here. This map could be used to identify where the environment's layout can be optimized, using the simulator to test and compare different layouts. An abstract algorithm for this could be to clear space around congested areas repeatedly, until the amount of conflicts is minimized.

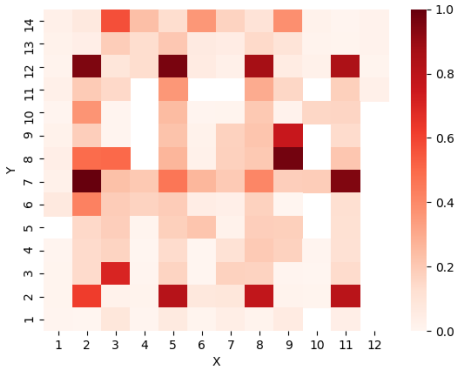


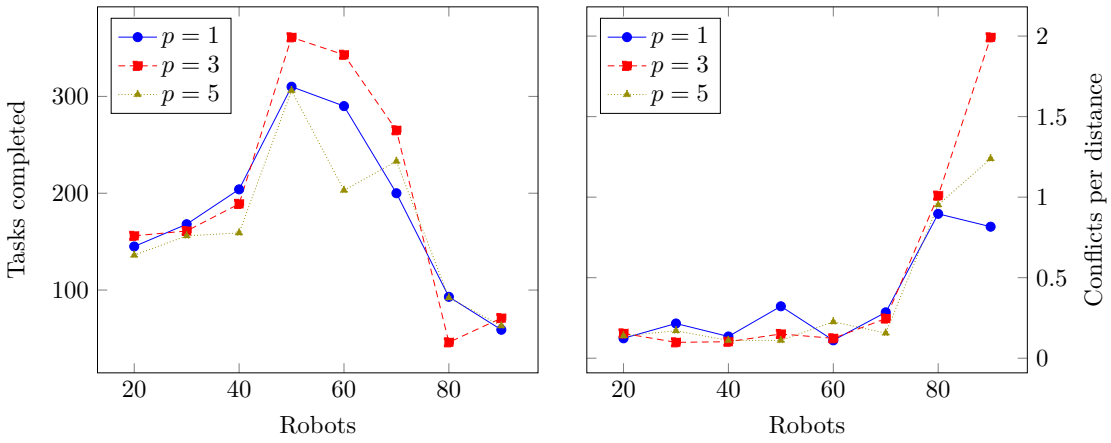
Figure 6.5.: Heat map conflicts in simulation with five agents and five unpredictable elements.

This information could also be used to penalized edges' weights in areas with a high level of conflicts. Higher weights on the in-going edges to a cell would make agents less likely to choose a path going through the cell, lessening the load in that area. Repeated simulations could be used to converge toward an optimal weighed graph.

6.1.4. Scaling

Lastly the MAS should be able to scale to larger environments and number of robots. In the following simulations, both the environment and number of robots have been increased in each subsequent test, each being allowed to run ten minutes. The density of robots, shelves and goal locations has been kept constant. As the sizes of environments varies in the simulations, the distances agents have to travel to complete tasks also varies. As such, the number of tasks that the systems is able to complete will be used

for comparing the MAS performance.



(a) The number of tasks completed by the MAS.

(b) Conflicts per distance with different scales of robots.

Figure 6.6.: Comparison of the MAS with different amount of robots.

Around 60 robots in a simulation is the limited for the MAS still performing well, while running on a single machine. However, the robots are using noticeably longer to resolve conflicts, hence the increase in the number of conflicts per distance. With more robots, allocating enough compute time for each agent is not possible, resulting in several agents being unable to resolve the conflicts they are involved in. However, this also shows that agents in the MAS can continue working and complete tasks, even when a subset of the agents in the system has failed.

6.2. Further work

A discussion of the necessary steps to move from simulation to real robots are given here, followed by suggested extensions for the simulation tool to represent more of the MiR robots abilities. Lastly, some improvements for the MAS are considered.

Real robots One of the next steps for the system would be to test it on a set of real robots. As the system has proven to work for at least a few agents in a confined space, testing on five real robots seems feasible. As the MAS is currently not setup to send commands to the robot, there is some work to be done here; either through integration with the Robot Operating System (ROS) or the robot's web application programming interface (API). The simulation is designed to model the robots as closely as possible, and only contains limitations to simplify how actions are interpreted. Actions should be directly convertible into actual actions on a robot, which also consists of a FORWARD, BACKWARDS, and TURN action. Similarly, the sensor information needs to be mapped to the precepts. Here some assumption has been made about how good robots are at

recognizing objects that might not currently be true in the real-world. Kiva Systems solve this recognition problem by using QR codes on objects.

To test the MAS further at a larger scale, the most feasible solution would be to apply containerized agent processes to multiple computer units, allowing the system to scale to more than what has been achieved here. This would also add a more accurate picture of how the agents would operate in the real-world, introducing problems such as network delays.

Simulating different types of robots and tools The simulation currently only supports one type of robot, which is able to handle all the tasks defined. MiR builds several different types of robots of multiple sizes, capable of carrying different loads and using different attachments, and thereby able handling different types of tasks. The primary case that should be handled is the MiRHook200 [**?**], which allows the robots to pull heavier load instead of carrying.[1] This could be done by adding a PULL action. However, as the simulation is now, it provides a good general abstraction of how the robots are able operate. With the distributed auction-based task delegation, each agent can by itself determine if it has the capabilities to carry out the task. This therefore solves the problem of correctly assigning tasks to robots that is actual able to solve the task (or at least believe they can).

Path finding and conflict resolution The two main factors constraining the MAS currently is path finding and conflict resolution. While the path finding algorithm discussed in Section 5.5 is able to account for other agents in the environment and find paths, it does not achieve this perfectly as conflicts still occur. The additional weight constraint on the graph lowers the amount of conflicts, but does not resolve them overall.

Agents in the system are currently able to ask each other to move, however detecting when this is possible is limited, meaning that agents can still become trapped. A more advanced conflict detection method is needed for the agents to know when they are stuck, along with a protocol for the agents to coordinate and solve the problem. One of the simpler solution suggested is to add small movements to create slightly altered states to solve the problem from. This might allow one agent to move out of the conflicted area, allowing others to solve their path-finding problem.

As mentioned in Section 6.1.3, the simulation tool is able to gather information about congested areas. A suggestion for an algorithm is given, which uses this information to build a new iteration of the graph, altering the weights to penalize moving through high density areas.

Task delegation The MAS designed in this project is currently using a centralized entity to delegate tasks, which exposes it to single point failures. However, the way the task manager is designed and implemented, it acts as a special agent within the system,

[1] The MiRHook200 can carry 1.5 times more than the MiR200 robot alone [**?**, **?**]

interpreting a special set of messages, as discussed in Section 5.3. As such, the workload of the task manager can be distributed to several agents in the system. Solutions for such include using a voting-based scheme, where an agent is elected to handle these responsibilities. Upon a failure, which can be detected by non-responsiveness from the current task manager, an election can be held to find a new agent capable of handling the responsibilities.

The MAS also has to ensure that task related information is not lost during a failure. Active mission will be distributed knowledge among all working agents, but completed and non-started tasks are not known to any other agent than the task manager. As the system is designed now, announcements made about adding or completing a task is made publicly, allowing everyone to store this information as needed. A solution to this problem could therefore be to elect, along with the task manager, one or more *secretary* agents, which will retain task related information, thereby building a hierarchy of agents. Each time either the task manager or a secretary agent fails, a new would be elected to retain the information. An alternative could be to ensure that each task data point is retained by at least some number k agents, having the agents act as a peer-to-peer system. Data can then be located through a distributed hash-table [**?**].

However, based on discussions with MiR, it is worth noting that users of a MAS controlling robots, in for example a warehouse, would prefer a clear overview of what the robots are doing at all time. Using a central persistent database for storage could be preferred for having rapid accesses to this information, benefiting both human users and agents in the MAS.

6.3. Summary

This chapter has evaluated the designed MAS using the simulation tool to create comparable results. The system is analyzed in a confined area, where the optimal number of agents is found based on the duration to complete tasks. The weighed graph search is tested with different penalty values, showing that it does lower the amount of conflicts, but still results in longer solution than the standard MAPF algorithm. Dynamic objects are added to the simulation, showing the agents capable of handling these but do find longer solutions. However, improvements to handling conflicts with both these objects and other agents are still possible. The MAS is scaled to problems with more agents, showing that around 50 agents are possible to run on a single machine before no more tasks are completed with more robots. Lastly, the steps to move the MAS to the real robots are given, along with suggestions for extensions to the simulation tool and improvements for the MAS.

CHAPTER 7

Conclusion

This project has focused on designing and implementing a MAS, with focus on how multi-agent methods and theories can be applied to Mobile Industrial Robots. Different applications of MAS within robotics has been investigated, including Kiva Systems which has proved to be one of the most successful applications of a large-scale MAS.

A tool to simulate the MiR robots have been build, allowing for a idealized version of their actions and environment. This simulation tool implemented has been able to run hundreds of robots in one instance, where different environments can be specified for testing multiple types and variations of layout. Each robot is provided with limited information of its surroundings through its sensors, making the environment partial observable. The simulator is currently limited to one type of robot but can be extended to include different kinds along with attachments.

A Multi-Agent System has been designed and implemented to solve tasks in the simulator. To achieve this, a library of hierarchical plans has been created, enabling the agents to build from abstract plans to concrete actions. Coordination between agents in the MAS is achieved through a set of messages, defining a protocol for how cooperation is done. This allows agents to inform others of their plans and actions, along with requesting alterations to others' plans. Tasks are distributed through the MAS's task manager, a special agent using a subset of the messages to instruct other agents.

The Multi-Agent Path-Finding problem is regarded in a continuous environment, where new goals are being generated. A algorithm based on a local A* search is given, where other agents' plans are taking into consideration, along with a general method to constructing a weighed graph to reduce conflicts. The algorithm has been shown to find solutions to the MAPF with several robots. The weighed graph search finds worse solution in terms of traveled distance and duration, but reduces the amount of conflicts and hence the overall computational load.

The MAS has been evaluated using the simulation tool to create different environments and scenarios. These has included working with a high density of robots, dynamic elements in the environment, and very large environments. The simulation tool can here be used to optimize the number of robots or layout in a given environment.

Appendices

APPENDIX A

Implementation notes

This appendix contains some notes and reasoning for the chosen tools and designs made when developing the MAS.

A.1. Simulation vs. real-world

The MAS has been designed from the beginning with the intention that it should be easy to transition from the simulation tool to real-world robots. Because of this, the design of each agent was from the beginning made able to be deployed either directly on the robot or on a cloud-based computer controlling the robot remotely through the robot's web application programming interface (API).

Each agent is build as an independent process and can run completely containerized, only communicating with other agents through using the language defined in Section 5.2. Technical, they are implemented in C# running on .NET core 3.0, allowing the agent program to run on multiple operating systems. Specifically, this allows the agent to be executed directly on the MiR robots, where it can interact with the Robot Operating System (ROS[1]).

Alternatively, every agent, including the robot and task manager agents, is containerized using Docker[2], allowing the agents to run and control the robots remotely. The MiR robots currently have an web application programming interface (API) that enables this. Note that the reactive layer of the robot agent should however run directly on the robots to minimize the delay from percept to emergency action. This also allows for simulating more robots by distributing agents to multiple machines.

The simulation tool has been writing using the Unity3D game engine, which contains a physics engine for simulating movement and interactions. This has been utilized to construct and execute the robots in their environment, including acceleration and deceleration curves. This adds an additional challenges for the MAS, which usually is

[1] `http://www.ros.org/about-ros/`
[2] `https://www.docker.com/`

simplified to assume that the robot move at a constant velocity and can do infinite accelerations.

A.1.1. Monitoring and statistics

An additional feature of the simulation tool is that it does not only allow for simulating the environment and the robots but can also be used for monitoring a real system. The data being sent to agents can be observed by the tool as well, creating an virtual illustration of the real-world environment. This can be used to create alerts for when a robot gets stuck and needs human help.

Statistics about the MAS are currently being gathered by a special *statistics agent*. This agent subscribes to the task manager, receiving updates about when new tasks are being published, started, and completed, and using these events to track time. Information about each agent is created and tracked by the individual agent and transmitted to the statistics agent upon request. Similarly, for the robots in the simulation.

A.1.2. Defining environments

The simulator uses a basic text format to define environment and objects. A complete definition of the supported objects can be found with the source code, but are summarized in Table A.1.

Charater	**Function**
r	Robot. Can be extended to more types
A	Shelf
=	Wall
#	Goal cell
+	Charging cell
*	Defines a storage location
x	Defines a no access zone where the robots are not allowed to enter
F	A feeder cell. This will periodically generate an object
P-Z	Dynamic objects that follows a path in the environment
H	A random walker. This will at each time step move to an adjacent cell

Table A.1.: Definition of the simulation's language to create environments

A.2. Centralized message delivery

As the MAS is implemented currently, it relies on centralized component *exchange* entity to ensure public announcement messages are delivered in the same order to every agent. This can make it difficult for the system to scale and will become a bottleneck for the MAS as the number of agents increases. The solution is current implemented using

the Open Source message broker RabbitMQ[3]. This is solution is widely used in many systems, and can easily and reliably send and receive messages from many hosts with guaranty for deliveries.

However, if a completely decentralized solution is desired, alternative solutions do exist which can solve the broadcast with a total ordering of the messages problem. [**?**] describes the problems for ensuring the order of arrival for the messages, along with rules for an algorithm to solve the given problems. A popular algorithm is the sequencer algorithm, where a process is elected to be the sequencer, and will on every broadcasted message attach a timestamp, defining the ordered of the messages [**?**, p. 20]. The sequencer can be fixed (essentially working as the centralized exchange based described above), or be selected through a voting scheme, allowing the sequencer to fail and a new selected dynamically. Several other approaches exist, and a survey of these distributed broadcast algorithms are given in [**?**]. However, for the purpose of simulating this MAS, the centralized approached have proved stable, but a decentralized algorithm should be implemented before applying the solution to real-world robots.

[3] `https://www.rabbitmq.com/`

APPENDIX B

Simulation data

The section contains the raw data from the experiments. The data for each type of simulation is divided into two tables: one for the task related data and one for the robot and agent related data. Both are given as a function of the number of robots/agents in the simulation. The first consists of:

- The *total duration* for the MAS to solve all the tasks.
- The amount of *completed tasks*.
- The average *total time*, measured from a task has been published to its completion.
- And the average *active time*, measured from a task has been assigned to an agent to its completion.

The second table consists of:

- The *total conflicts* that has occurred. Note that the number does is not precise, but gives a relative data point to compare the different tests.
- The total *distance traveled* by all of the robots.
- The *energy* consumed by the robots. The energy has not been a major factor in these tests, as robots have had too few tasks to require recharging.
- The percentage of time the *robot is active*. An robot is regarded when it executes an action.
- The percentage of time the *active* time for agents. An agent is regarded as active when it has a task.

Table B.1.: Task related data for the restricted space simulations on a small map (Section 6.1.1)

Agents	**Duration** [s]	**Completed tasks**	**Avg. task** [s]	**Avg. active** [s]
1	903.57	20	432.53	44.51
2	407.11	20	188.44	39.14
3	269.80	20	119.77	38.16
4	209.60	20	91.92	37.66
5	167.90	20	76.24	37.04
6	203.23	20	98.22	54.39
7	184.74	20	84.17	52.98
8	167.98	20	78.66	56.09
9	164.03	20	78.84	59.40
10	193.05	20	76.20	65.77
11	169.77	20	80.67	67.16
12	214.39	20	90.02	79.75
13	153.31	20	74.66	65.19
14	177.13	20	80.33	74.27
15	195.55	20	87.26	83.35
16	219.58	20	93.18	90.95
17	134.65	20	62.28	60.00
18	174.08	20	75.42	64.18

Table B.2.: Robot and agent related data for the restricted space simulations on a small map (Section 6.1.1)

Agents	**Conflicts**	**Distance**	**Energy**	**Robot idle**	**Agent idle**
1	0	552.00	0.09	0.52	0.97
2	3	570.11	0.08	0.39	0.95
3	43	546.81	0.08	0.58	0.93
4	21	581.66	0.08	0.60	0.92
5	23	592.84	0.08	0.60	0.90
6	127	608.29	0.12	0.43	0.90
7	389	615.36	0.13	0.41	0.85
8	140	621.07	0.13	0.40	0.85
9	237	683.71	0.16	0.36	0.77
10	356	713.65	0.19	0.33	0.74
11	247	675.57	0.20	0.30	0.69
12	526	697.01	0.25	0.23	0.66
13	263	653.75	0.20	0.29	0.69
14	87	652.49	0.25	0.23	0.63
15	394	675.83	0.30	0.21	0.61
16	1130	715.94	0.36	0.18	0.55
17	244	648.40	0.23	0.26	0.59
18	652	717.65	0.32	0.20	0.43

Table B.3.: Task related data for the non-restricted space simulations on a small map (Section 6.1.1)

Agents	**Duration [s]**	**Completed tasks**	**Avg. task [s]**	**Avg. active [s]**
1	782.87	20	371.35	38.10
2	366.92	20	166.80	36.62
3	239.38	20	103.32	33.02
4	176.35	20	70.80	29.64
5	146.52	20	58.12	29.83
6	181.24	20	81.84	47.15
7	168.00	20	72.59	47.75
8	159.77	20	65.57	47.66
9	141.75	20	60.34	47.98
10	139.48	20	60.69	50.82
11	134.70	20	56.26	49.54
12	128.62	20	52.68	47.36
13	131.18	20	53.65	49.37
14	131.15	20	53.15	50.40
15	126.08	20	53.26	49.24
16	128.09	20	51.32	49.26
17	123.51	20	51.92	49.96
18	139.44	20	54.41	51.55
19	132.02	20	51.39	49.42
20	104.26	20	31.58	29.29

Table B.4.: Robot and agent related data for the restricted space simulations on a small map (Section 6.1.1)

Agents	**Conflicts**	**Distance**	**Energy**	**Robot idle**	**Agent idle**
1	0	550.00	0.08	0.39	0.98
2	0	542	0.07	0.55	0.96
3	0	541.00	0.07	0.58	0.92
4	0	524.00	0.07	0.56	0.88
5	0	536.09	0.07	0.55	0.87
6	0	544.27	0.11	0.42	0.89
7	0	546.00	0.12	0.39	0.85
8	0	546.00	0.13	0.37	0.80
9	0	536.00	0.13	0.35	0.80
10	0	559.00	0.14	0.35	0.77
11	0	527.00	0.15	0.31	0.72
12	0	515.00	0.15	0.29	0.68
13	0	532.00	0.17	0.28	0.65
14	0	502.00	0.19	0.25	0.61
15	0	523.41	0.19	0.24	0.56
16	0	497.00	0.21	0.22	0.55
17	0	484.00	0.22	0.21	0.54
18	0	516.21	0.26	0.19	0.47
19	0	488.29	0.25	0.18	0.46
20	0	471.00	0.21	0.22	0.36

Table B.5.: Task related data using the WeighedGraph search with *penalty* = 3 (Section 6.1.2)

Agents	**Duration [s]**	**Completed tasks**	**Avg. task [s]**	**Avg. active [s]**
1	975.72	20	465.57	47.97
2	507.87	20	253.09	49.69
3	346.59	20	163.94	51.06
4	273.89	20	127.32	51.40
5	240.84	20	112.69	54.08
6	212.17	20	101.81	56.62
7	184.61	20	86.27	54.91
8	205.52	20	101.89	69.32
9	173.86	20	95.51	69.98
10	177.98	20	92.47	71.34
11	173.44	20	94.83	76.58
12	141.53	20	71.10	61.81
13	156.48	20	76.95	69.89
14	177.13	20	83.70	77.06
15	181.88	20	99.79	90.78

Table B.6.: Robot and agent related data using the WeighedGraph search with *penalty* = 3 (Section 6.1.2)

Agents	Conflicts	Distance	Energy	Robot idle	Agent idle
1	0	602.00	0.10	0.51	0.98
2	11	631.30	0.10	0.52	0.98
3	105	641.78	0.10	0.51	0.95
4	29	603.71	0.11	0.50	0.93
5	33	654.52	0.12	0.47	0.91
6	31	651.87	0.13	0.46	0.89
7	109	663.46	0.13	0.44	0.86
8	124	724.95	0.16	0.37	0.85
9	74	743.29	0.16	0.43	0.88
10	110	789.05	0.18	0.38	0.80
11	112	818.07	0.19	0.37	0.80
12	80	709.79	0.17	0.36	0.76
13	102	792.58	0.21	0.34	0.72
14	83	775.66	0.26	0.29	0.67
15	182	837.87	0.28	0.29	0.69

Table B.7.: Task related data using the WeighedGraph search with *penalty* = 5 (Section 6.1.2)

Agents	Duration [s]	Completed tasks	Avg. task [s]	Avg. active [s]
1	1101.10	20	551.53	52.91
2	565.51	20	275.29	54.74
3	378.08	20	172.94	54.84
4	294.10	20	136.12	54.19
5	251.85	20	118.92	57.47
6	240.20	20	105.05	59.30
7	234.60	20	125.04	75.41
8	197.58	20	98.58	67.95
9	213.41	20	102.19	73.59
10	201.50	20	101.64	78.45
11	197.39	20	94.41	78.88
12	218.81	20	100.05	84.51
13	341.48	20	103.07	95.03
14	173.24	20	88.50	81.73
15	163.56	20	80.30	76.45

Table B.8.: Robot and agent related data using the WeighedGraph search with *penalty* = 5 (Section 6.1.2)

Agents	Conflicts	Distance	Energy	Robot idle	Agent idle
1	0	732.00	0.11	0.55	0.99
2	9	719.53	0.11	0.53	0.96
3	19	700.25	0.11	0.52	0.93
4	6	694.00	0.12	0.50	0.92
5	53	717.54	0.13	0.49	0.91
6	38	744.86	0.15	0.45	0.84
7	45	812.70	0.17	0.42	0.89
8	46	828.19	0.16	0.46	0.85
9	85	872.26	0.19	0.40	0.81
10	131	927.09	0.20	0.39	0.80
11	83	846.39	0.22	0.35	0.76
12	122	1032.81	0.26	0.35	0.69
13	66	808.95	0.44	0.17	0.47
14	72	905.60	0.25	0.33	0.71
15	80	853.67	0.25	0.31	0.65

Table B.9.: Task related data using the WeighedGraph search with *penalty* = 10 (Section 6.1.2)

Agents	Duration [s]	Completed tasks	Avg. task [s]	Avg. active [s]
1	1148.84	20	580.46	55.34
2	580.47	20	290.21	57.17
3	394.87	20	195.03	60.51
4	323.86	20	148.79	58.85
5	257.88	20	133.59	64.54
6	231.87	20	110.98	63.11
7	219.30	20	114.47	72.62
8	215.82	20	109.97	74.27
9	184.39	20	85.95	65.49
10	191.61	20	105.28	84.79
11	205.62	20	103.59	84.65
12	183.78	20	92.90	77.94
13	172.50	20	89.36	80.03
14	228.78	20	108.26	102.55
15	219.70	20	125.77	119.44
16	245.28	20	123.86	121.08

Table B.10.: Robot and agent related data using the WeighedGraph search with *penalty* = 10 (Section 6.1.2)

Agents	Conflicts	Distance	Energy	Robot idle	Agent idle
1	0	792.00	0.11	0.57	0.99
2	7	774.71	0.12	0.55	0.97
3	30	790.88	0.12	0.56	0.96
4	13	772.51	0.13	0.51	0.90
5	33	818.26	0.13	0.53	0.95
6	302	855.60	0.14	0.53	0.92
7	66	881.68	0.15	0.48	0.89
8	128	930.65	0.17	0.46	0.86
9	56	844.03	0.17	0.43	0.81
10	98	1007.25	0.19	0.46	0.86
11	78	1026.83	0.23	0.39	0.77
12	203	962.82	0.23	0.39	0.75
13	172	937.49	0.23	0.37	0.74
14	476	1118.01	0.31	0.30	0.66
15	83	1072.99	0.33	0.33	0.73
16	78	1058.80	0.40	0.26	0.63

Table B.11.: Task related data with five robots and unpredictable elements (Section 6.1.3)

Agents	Duration [s]	Completed tasks	Avg. task [s]	Avg. active [s]
5	232.58	20	109.27	52.68
5	223.07	20	100.86	50.10
5	259.11	20	116.44	55.97
5	367.37	20	186.67	69.07
5	301.94	20	136.54	64.87
5	330.70	20	148.59	69.43

Table B.12.: Robot and agent related data with five robots and unpredictable elements (Section 6.1.3)

Agents	Conflicts	Distance	Energy	Robot idle	Agent idle
5	54	599.74	0.12	0.45	0.90
5	23	579.92	0.11	0.45	0.90
5	85	652.47	0.13	0.46	0.87
5	147	835.83	0.18	0.46	0.75
5	117	806.98	0.15	0.51	0.86
5	930	905.26	0.17	0.51	0.85

Table B.13.: Task related data with ten robots and unpredictable elements (Section 6.1.3)

Agents	Duration [s]	Completed tasks	Avg. task [s]	Avg. active [s]
10	147.65	20	70.18	57.28
10	151.41	20	70.74	57.78
10	196.96	20	96.06	72.92
10	243.59	20	105.34	67.02
10	279.85	20	149.59	109.42
10	219.93	20	108.14	86.50

Table B.14.: Robot and agent related data with ten robots and unpredictable elements (Section 6.1.3)

Agents	Conflicts	Distance	Energy	Robot idle	Agent idle
10	66	683.56	0.15	0.40	0.81
10	93	687.40	0.15	0.42	0.83
10	290	837.33	0.20	0.41	0.78
10	152	802.33	0.24	0.32	0.56
10	230	985.65	0.28	0.42	0.74
10	405	1003.46	0.22	0.44	0.80

Table B.15.: Task related data for simulations with many robots (Section 6.1.4)

Agents	Duration [s]	Completed tasks	Avg. task [s]	Avg. active [s]
20	601.88	145	77.90	75.40
30	602.39	168	72.16	68.71
40	601.36	204	86.07	82.65
50	600.23	310	70.45	66.92
60	602.17	290	82.62	78.88
70	601.73	200	82.18	77.30
80	612.02	93	106.81	98.81
90	598.74	59	144.39	134.45

Table B.16.: Robot and agent related data with many robots (Section 6.1.4)

Agents	Conflicts	Distance	Energy	Robot idle	Agent idle
20	862	6973.17	0.92	0.61	0.98
30	2818	13 070.61	1.69	0.62	0.97
40	1843	13 656.41	1.43	0.59	0.98
50	7220	22 392.07	2.55	0.57	0.97
60	2804	25 250.27	3.06	0.63	0.97
70	6055	21 259.26	3.64	0.67	0.97
80	15914	17 753.89	6.33	0.78	0.97
90	14307	17 520.73	6.24	0.78	0.97

Table B.17.: Task related data for scaling robots and *penalty* = 3 (Section 6.1.4)

Agents	Duration [s]	Completed tasks	Avg. task [s]	Avg. active [s]
20	606.92	156	59.01	56.32
30	602.39	161	66.89	63.52
40	606.19	189	81.22	77.65
50	639.99	361	68.97	65.42
60	603.73	343	75.72	71.34
70	623.07	265	81.71	76.75
80	588.70	46	181.48	168.00
90	774.18	71	291.57	251.97

Table B.18.: Robot and agent related data for scaling robots and *penalty* = 3 (Section 6.1.4)

Agents	Conflicts	Distance	Energy	Robot idle	Agent idle
20	1356	8802.89	1.14	0.49	0.98
30	1143	11 600.74	1.68	0.47	0.98
40	1928	18 834.63	2.24	0.59	0.98
50	4050	26 909.90	3.01	0.58	0.97
60	3695	29 909.99	3.25	0.62	0.97
70	7379	29 954.45	4.25	0.67	0.97
80	12554	12 438.08	4.54	0.70	0.96
90	29827	14 972.75	4.87	0.58	0.88

Table B.19.: Task related data for simulation with many robots and *penalty* = 5 (Section 6.1.4)

Agents	Duration [s]	Completed tasks	Avg. task [s]	Avg. active [s]
20	608.85	136	63.42	60.65
30	646.72	156	71.95	68.61
40	612.91	159	87.31	83.06
50	604.02	306	74.55	70.72
60	609.83	203	78.47	73.28
70	612.46	233	82.95	78.10
80	692.55	92	111.93	103.03
90	714.34	63	113.06	104.36

Table B.20.: Robot and agent related data with many robots and *penalty* = 5 (Section 6.1.4)

Agents	**Conflicts**	**Distance**	**Energy**	**Robot idle**	**Agent idle**
20	1128	8131.04	1.13	0.51	0.98
30	2180	12 813.86	1.88	0.63	0.98
40	1916	17 337.89	2.26	0.60	0.98
50	2503	22 638.32	2.73	0.58	0.97
60	3546	15 690.23	2.57	0.65	0.97
70	3690	23 682.86	3.82	0.64	0.97
80	12840	13 484.62	3.28	0.61	0.97
90	16641	13 437.38	5.71	0.67	0.97