DTU Compute
Department of Applied Mathematics and Computer Science

Multi-Agent Systems

Erik Kristian Gylling

Bachelor Thesis
Software Technology
December, 2020

Multi-Agent Systems

Bachelor Thesis
December, 2020

By
Erik Kristian Gylling

Published by: DTU, Department of Applied Mathematics and Computer Science, Richard Petersens Plads, Building 324, 2800 Kgs. Lyngby Denmark
www.compute.dtu.dk

Abstract

In this thesis improvements to a multi-agent system are implemented and evaluated. The aim is to improve the performance of GOAL-DTU's multi-agent system in the qualification to the Multi-Agent Programming Contest 2020. Three improvements are made. The first improvement focuses on movement and uses A*. This is a complete search algorithm that uses a heuristic function. The second improvement focuses on the communication between agents and uses communication protocols for sharing information. The third improvement focuses on the preparation of agents. This improvement can be used as a foundation for multiple strategies. The A* and communication protocols are implemented in GOAL-DTU's multi-agent system.

A* and the communications protocols were tested on GOAL-DTU's MAS. A* was an improvement on a small map with few agents but didn't work on a larger map with many agents. The communication protocols did improve communication of GOAL-DTU's MAS on a small map but worsened the overall score. Too many agents sharing information, doesn't perform well with an algorithm used by GOAL-DTU.

Preface

This thesis has been prepared from September 30th to December 16th 2020 at the Department of Applied Mathematics and Computer Science, at the Technical University of Denmark, DTU, in partial fulfilment for the degree Bachelor of Science in Engineering, BSc Eng. The thesis is equivalent of 15 ECTS points. A course of 5 ECTS points have been maintained concurrently with the thesis. It is assumed that the reader has a basic knowledge in the areas of logical programming and search algorithms to understand parts of the thesis. Knowledge equal to DTU Courses *02180 Introduction to Artificial Intelligence* and *02156 Logical Systems and Logic Programming* are sufficient.

Jørgen Villadsen, Associate Professor, Supervisor
I would like to thank Jørgen for giving me the opportunity to work on this project. His experience with projects was valuable to me when planning the project and improving the report.

Alexander Birch Jensen, PhD Student, Co-Supervisor
Alexander's experience with multi-agent systems was valuable for me when I needed a person to spare with. He had good suggestions for the report and helped reformulate complex sentences.

I would like to thank my mother for proofreading and showing great interest in the project. It helped me formulate myself in a simple manner.

Last but not least I would like to thank my girlfriend, Kathrine. She gave me extra space when I was deeply focused in the project and helped me relax when I wasn't supposed to work on the project.

Erik Kristian Gylling - s173896

....
Signature

....
Date

Contents

1 Introduction

A multi-agent system is composed of an environment and multiple intelligent agents, which strive to achieve goals in the environment. The agents can collaborate if a goal is too hard for a single agent to achieve. Multi-agent systems can be used in any scenario where multiple agents are in the same environment. Intelligent agents can be anything from humans to software. A scenario could be the traffic of Copenhagen where road users want to arrive safely to their destinations. The environment is the streets of Copenhagen and the agents are the road users. In the future, the agents can be self-driving cars. This year GOAL-DTU qualified in the Multi-Agent Programming Contest 2020 with a multi-agent system written in the agent-oriented programming language, GOAL. This multi-agent system was created by the GOAL-DTU team in 2019 and was later improved in 2020. The scenario in the Multi-Agent Programming Contest 2020 is called *Agents Assemble II*. In the scenario agents with limited local vision have to organize themselves to assemble and deliver complex structures made of blocks, in a grid world formed as a torus.

This thesis aims to improve the movement and communication of the agents in GOAL-DTU's multi-agent system. The goal of the improvements is to increase the points scored by GOAL-DTU in the qualification of the Multi-Agent Programming Contest 2020. Furthermore, an improvement will be made that makes the agents prepare for new tasks. This implementation can be seen as a foundation for a strategy.

2 Multi Agent Systems with GOAL

The multi-agent system used by GOAL-DTU in the qualification for the MAPC 2020 is based on code written in the agent-oriented programming language GOAL. (For full documentation on GOAL see: Hindriks 2018.)

2.1 Multi-Agent Systems

A multi-agent system (MAS) is composed of an environment and multiple intelligent and autonomous agents, which strives to achieve goals in the environment. The agents can collaborate if a goal is too hard for a single agent to achieve. MASs can be physical in the form of road users in the streets of Copenhagen or virtual as it is in the MAPC. The focus of the thesis will be on a virtual MAS.

2.1.1 Environment

The environment sets the scene for a MAS and is independent of the MAS. An environment always includes at least two entities and may include static or dynamic obstacles to make a goal harder to achieve. An entity is the body of an agent. An entity perceives percepts that match the entity's surroundings and the previous action. If the entity can't observe the entire environment at the same time, then the environment is said to be partially observable. An environment can even change independently from the entities' actions. This kind of environment is said to be dynamic. The environment focused on in this thesis, and used for the MAPC, is both dynamic and partially observable.

2.1.2 Agents

The agents are the brains behind the entities in a MAS. An agent is connected to an entity in the environment and will receive the percepts that are perceived by the entity. The agent will use the percepts to create beliefs, which the agent will use to create goals and to decide what action the entity should perform. All this is done without any intervention from humans and that makes the agents autonomous. The entities are not doing any reasoning themselves and can be seen as a simple proxy that takes commands from agents outside the environment. Figure 2.1 is a visualization of the relationship between an agent and an entity.

The agents must be intelligent and rational. An agent is intelligent if the agent can communicate with other agents and act on changes made to the environment. The agent should not wait for the environment to give a lead on how to achieve a goal but act proactively to find a way to achieve a goal. An agent is rational if the agent knows its capabilities and limits for what the agent can achieve both alone but also with other agents. A rational agent would never take on a goal that is unlikely to be achieved. An example of a goal that is unlikely to be achieved for a road user, is to drive twelve kilometers through the streets of Copenhagen in five minutes.

2.2 GOAL

An agent is defined by how the agent is using the given percepts. An agent can use the percepts smartly and perform well or don't use the percepts at all and perform poorly. GOAL is an agent-oriented programming language and is used to create intelligent agents.

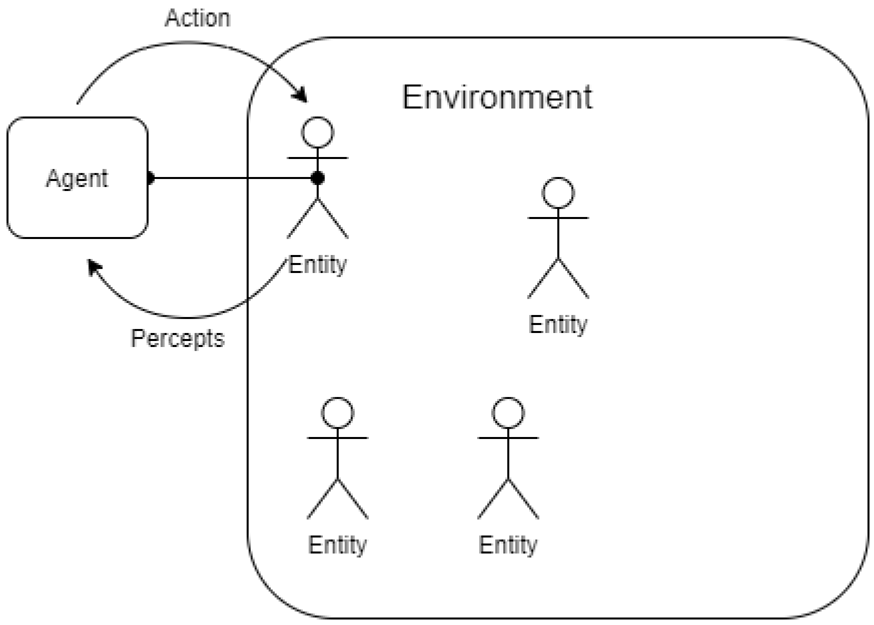


Figure 2.1: An agent is connected to an entity. The entity sends its percepts to the agent, and the agent sends an action to the entity.(From Hindriks 2018, page 52)

2.2.1 Creating a Multi-Agent System in GOAL

A multi-agent system must have one `.mas`-file. The `.mas`-file contains the definition of one or multiple agents and a launch policy that tells which agents to create. The `.mas`-file is the recipe for building and launching a MAS.

2.2.2 Belief-, Goal- and Knowledge Base

The belief base is a collection of beliefs of the agent. The agent creates its own beliefs and can remove the beliefs as well.

The goal base is very similar to the belief base. The goal base is a collection of goals of the agent. The agent adopts its own goals and can drop the goals as well. A goal is achieved when an agent has a belief that is identical to the goal. A goal is automatically dropped from the goal base when the goal is achieved.

The knowledge base is a collection of logical rules that the agent uses for reasoning.

The belief-, goal- and knowledge base are Prolog scripts where the beliefs and goals are defined as dynamic predicates and the logical rules correspond to predicates. The script used for the knowledge base can also be used for the belief base and goal base at the same time.

2.2.3 Agents in GOAL

Agents in GOAL are built by logical rules that tell what the agents should do in specific states. An agent must have at least one module and a knowledge base. A module is a collection of operations that may be triggered by new percepts, beliefs, or goals. Modules are the scripts of GOAL.

GOAL uses an execution cycle to control when the agent should process events and when the agent should choose an action. If an agent has an initial module, then the initial module is entered as the first module when the agent is created. The initial module tells which beliefs and goals the agent is created with. The initial module can also be used to perceive percepts in the environment that are only available in the first cycle. After the initial module, a fixed cycle is executed. The fixed cycle is presented on fig. 2.2.

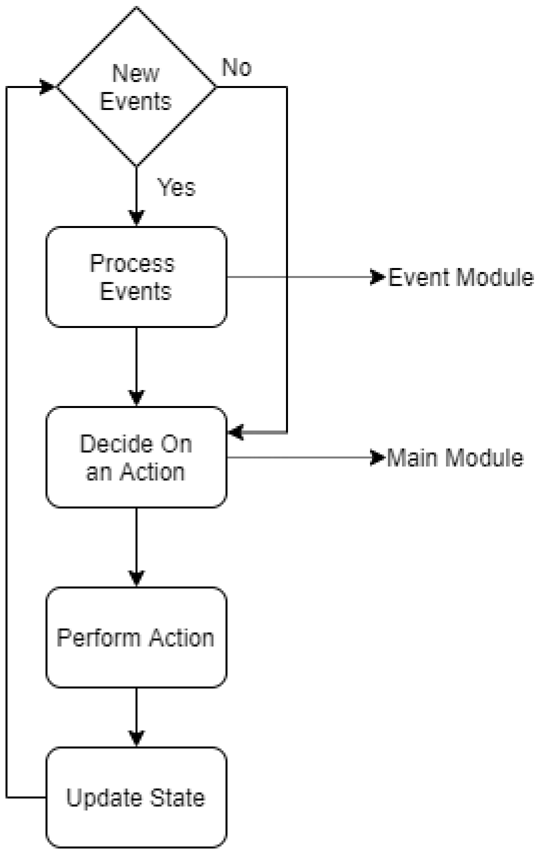


Figure 2.2: A cycle is performed every time the environment has changed. Only one cycle can be active for one agent. (From Hindriks 2018, page 63)

Step 1: New Events At the beginning of each cycle, events are collected. A check is made to see if any new events have occurred. An event is new if it wasn't received or occurred in the previous cycle. The event module will be entered if a new event has occurred and the agent has an event module. Otherwise, the main module would be entered.

Step 2: Process Events An agent may have an event module. The event module is used to process any new event that occurred in the environment. Percepts perceived by the entity in the environment, received messages from other agents, and actions performed by the entity all counts as events. These events are used to change the agent's beliefs in its belief base. This step is skipped if an agent doesn't have an event module.

Step 3: Decide On an Action At this step, the agent uses its beliefs and goals to reason about what action the entity must perform. This is normally done in the main module. An agent must have a main module and an agent will terminate when the main module terminates.

Step 4: Perform Action The agent sends a request to the entity to perform the chosen action from step 3.

Step 5: Update State Actions have preconditions and postconditions. This step updates the state of the agent such that the postcondition of the performed action is true. The precondition of an action must be true for an action to be performed. Achieved goals are removed in this step as well.

2.2.4 Cognitive State Queries

Cognitive state queries are used in modules to check if the agent has a specific combination of percepts, beliefs, or goals. A cognitive state query is used as a guard before performing an action and the query is using the same syntax as the knowledge base. Figure 2.3 shows three examples of state queries for the percept base, belief base, and goal base.

```
if percept(traficlight(green)) then insert(traficlight(green)).
if bel(traficlight(green)) then drive().
if goal(speed(42)) then adjustSpeed(42).
```

Figure 2.3: Three state queries and their corresponding action.

2.2.5 Actions

Two kinds of actions exist: external and internal. An external action is defined by the environment and the entity will perform the action when told so. An external action for a road user is ”Turn Right” because the action ”Turn Right” is defined in the streets of Copenhagen.

An internal action is defined by the developer in GOAL and is performed by the agent. Creating or deleting beliefs are examples of internal actions. They are independent of the environment.

2.2.6 Communication

Agents can use internal actions to send and receive messages. Messages are sent to the receiver’s mailbox and will be available for the agent in the next cycle. Messages are only available for one cycle because the mailbox is emptied after every cycle. An agent has the opportunity to send a message to a specific agent, or a channel. To receive messages from a channel an agent must subscribe to the channel. An agent will receive all messages sent to a channel when the agent has subscribed to the channel. Whether channels can be used to lower the number of messages sent is not certain.

3 The Multi-Agent Programming Contest

The Multi-Agent Programming Contest is organized by Clausthal University of Technology and has the purpose to stimulate research in the area of multi-agent systems. A book about the contest is published every year. The latest contest is documented in Ahlbrecht et al. 2020. Information about the contest in 2020 is found on their website[1].

3.1 The Scenario: Agents Assemble (2020)

Two teams of agents are moving around trying to solve tasks. A task is to find some specific blocks and align them correctly inside a goalzone to create a pattern. The team gets points when a task is completed. How many points a task gives are based on how difficult the pattern is to create and how much time the task has been available. The team with the most points in the end wins.

3.2 Environment

The environment used for the MAPC 2020 is partially observable and dynamic. So the agents can not be sure if previous percepts are valid. The environment is a rectangular grid that is composed of cells. The dimensions of the grid are not known by the agents. The grid is a torus, i.e. if an entity moved off the right edge of the map, it will appear on the left side instead.

The environment uses simulation steps to synchronize all actions performed by entities. A simulation starts with the simulation step, 0. This value is increased for every cycle taken in the simulation until the simulation has finished. If the environment has received an action request from all agents the environment will continue to the next simulation step. Agents have a configurable time interval to request an action. The time interval used in the MAPC 2020 is 4 seconds.

3.2.1 Entities

Multiple entities are created when starting the environment. The number of entities is proportional to the number of agents trying to connect to the environment up to a configurable value. Each agent controls one entity in the simulation. The agent is not told the absolute location of the entity in the environment before connecting to the entity. The only information the agents know about their entity is the current energy level of the entity and whether the entity is disabled or not. The energy level is used for `Clear` actions. The entity automatically recharges 1 energy per simulation step. When a `Clear` event hits an entity, all its attachments are lost and the entity is disabled for a configurable number of simulation steps. A disabled entity can't perform any actions. The entity repairs itself such that it works normally after a few steps. An entity can perceive cells in a configurable range. The value of the configurable range is in euclidean distance. A cell located at (X,Y) relative to the entity will be in vision for a range R, if $\mid X \mid + \mid Y \mid \leq R$.

3.2.2 Things

Cells in the grid may inhabit one out of four different things.

Block: A block is a thing that is used to create patterns and complete tasks. Every block has a type. The number of different types of blocks is configurable. A cell is impassable if it contains a block.

[1] https://multiagentcontest.org/: visited on December 12th, 2020

Dispenser: A dispenser is a thing where a block can be requested. A dispenser has a type and the blocks coming from the dispenser have the same type.

Entity: Each agent controls one entity in the simulation. Entities move around in the environment and can attach themselves to blocks. A cell is impassable if it contains an entity.

Markers: A marker is a cell that will be cleared in one of the next cycles. The agent can use the markers to avoid the `Clear` event such that the entity won't be disabled.

3.2.3 Terrain

All cells in the grid have one of four different terrains.

Obstacle: An obstacle is an impassable cell. Obstacles can be removed by a `Clear` event. Several obstacles are located in the environment.

Task board: A task board is a cell where an agent can accept tasks by using the `A` action. An entity needs to be at most two cells away to accept a task from the task board. Multiple task boards are normally present in the environment.

Goal cell: A goal cell is a cell where a task can be submitted. Goal cells are positioned in bigger groups. These groups are called goalzones. Multiple goalzones are normally present in the environment.

Empty: A cell is just a cell if no terrain is specified.

3.2.4 Task

The main challenge in this environment is to complete tasks. A task is composed of a unique name, a pattern, a reward, and a deadline. The tasks must be accepted at a task board by the submit-agent, who is the agent submitting the task. The pattern is composed of tuples containing coordinates and types of blocks. A tuple tells that a block with the given type should be located at the coordinates relative to the submit-agent. The blocks also need to be acknowledged as a pattern. The submit-agent must be located in a goalzone to submit a task. The team gets the reward when the task has been successfully submitted. The reward decays with time. This means that a task should be completed as fast as possible to get the highest reward.

3.2.5 Events

The environment randomly creates events. In the MAPC 2020, only one event is created.

`Clear`: This event removes any obstacle or block and disables any entity in a configurable range of the origin of the event. Entities will be disabled for a few simulation steps. Markers are located in cells that are soon to be cleared and warn the entity about the `Clear` event.

3.2.6 Actions

The agent can interact with the entity through actions. In the environment for the MAPC 2020, the following actions are defined. All the actions have the same configurable risk to randomly fail. An action won't be performed by the entity if the action fails. The agent receives three percepts about actions. The name of the action that was performed in the previous cycle, what the action-parameters were, and if the action was successful.

`Move(Direction)`: This action is used to move the entity in the environment. The `Move` action needs a parameter to tell which direction the entity should move. `Direction` must be one of [`n, s, w, e`]. Standing for north, south, west, and east. Failures of `Move(Direction)` will occur if one of the following is true:

- The entity or one of the attachments are blocked.

`Request(Direction)`: This action is used to request a block from a dispenser that is located in the adjacent cell corresponding to `Direction`. `Direction` must be one of [`n, s, w, e`]. `Request(Direction)` will fail if one of the following is true:

- A `Thing` is on top of the dispenser.
- The dispenser is not found.

`Attach(Direction)`: This action is used to attach a block that is located in the adjacent cell corresponding to `Direction`. `Direction` must be one of [`n, s, w, e`]. Failures of `Attach(Direction)` will occur if one of the following is true:

- The block is not found.
- The block is attached to another agent.

`Detach(Direction)`: This action is used to detach a block that is located in the adjacent cell corresponding to `Direction`. `Direction` must be one of [`n, s, w, e`]. Failures of `Detach(Direction)` will occur if one of the following is true:

- The block is not found.
- The block is not attached to the agent.

`Accept(Task)`: This action is used to accept a task from a task board. The parameter, `Task`, is the name of the task the entity wants to accept. `Accept(Task)` will fail if one of the following is true:

- No task has the name, `Task`.
- The distance from entity to task board is longer than two cells.

`Submit(Task)`: This action is used to submit an action at a goal cell. The parameter, `Task`, is the name of the task the entity wants to submit. `Submit(Task)` will fail if one of the following is true:

- No task has the name, `Task`.
- The entity is not located on a goal cell.
- The pattern of the task is not correctly created.

`Skip`: The skip action is used to make the entity wait for one cycle. This action will always succeed.

`Rotate(Direction)`: This action is used to rotate the entity and the blocks attached to the entity. If an entity has no attached blocks, this action will be equal to the action `Skip`. The `Rotate` action needs a parameter to tell which direction the entity should rotate. `Direction` must be `cw` or `ccw`. Standing for clockwise and counterclockwise. `Rotate(Direction)` will fail if the following is true:

- Another entity, block, or obstacle is located in a cell, where one of the attached blocks have to go through to complete the rotation.

`Clear(X,Y)`: This action is used to clear a target location and the adjacent cells. The parameters `X` and `Y` are used to locate the target relative to the entity. This action is a smaller and less powerful `Clear` event. Multiple `Clear` actions must be made on the same location before the cells are cleared. The `Clear` action consumes a configurable amount of the entity's energy. `Clear(X,Y)` will fail if one of the following is true:

- The entity doesn't have enough energy.
- The target is not in the entity's vision.

`Connect(Name,X,Y)`: This action is used to connect an attached block to a block that's attached to another entity. `Name` is the name of the other entity to cooperate with. The parameters `X` and `Y` are the coordinates of the entity's block used to connect. This action needs both entities to perform this action with matching parameters to be successful. `Connect(Name,X,Y)` will fail if one of the following is true:

- The other entity is not performing a matching `connect` action.
- The blocks are already connected.
- The block is not attached.
- The block is not found.

`Disconnect(X,Y, X1, Y1)`: This action is used to disconnect two blocks attached to the entity. The parameters `(X,Y)` is the location of the first block and `(X1,Y1)` is the location of the second block. `Disconnect(X,Y, X1, Y1)` will fail if one of the following is true:

- The locations are not attachments of the entity.
- The blocks are not connected directly.

3.2.7 Percepts

In simulations, agents receive percepts from their entity in the form of `JSON` objects.

Initial Percepts The first percepts an agent receives is the initial percepts. Initial percepts give information that doesn't change through the simulation. The initial percepts include:

- The name of the entity.
- The team of the entity.
- The size of the team where the entity belongs.
- The number of steps in the simulation.
- The range of the entity's vision.

Step Percepts For every simulation step, agents receive a JSON object containing the percepts of their entities for the current simulation step. The step percepts include:

- The score of the team where the entity belongs.
- The last action performed.
- Status of the last action. Was it successful or not?
- Parameters of the last action.
- The current level of energy.
- Whether the entity is disabled.
- The most recently accepted task by the entity.
- Information about things in vision of the entity.
- Information about the terrain around the entity.
- Information about tasks.
- Information about attachments of the entity.

3.2.8 Configurations

A simulation is defined by a configuration file. The configuration file is a JSON-file. The configurable values, but the time interval, that are mentioned through chapter 3, are included in this configuration file. The configuration file contains information about the design of the grid, information about the setup of entities, information used for creating events and tasks.

3.2.9 The MASSim Server

Simulations are executed by a server to which teams must connect to in order to compete. The server is called MASSim, short for Multi-Agent System Simulation. The source code for MASSim is available at GitHub[2].

The code is written in Java 13 and must be refactored to Java 8 to be compatible with GOAL. To execute the server the code must be compiled into a `.jar`-file. In the terminal write:

```
java -jar ./path-To-jar-file/file-name
```

It's possible to show the simulation. For this write the following where `port` is a number higher than 8000.

```
java -jar ./path-To-jar-file/file-name  --monitor port
```

Then open your browser and write `localhost:port`. `port` must match the number written in the terminal.

[2]https://github.com/agentcontest/massim_2020: Visited December 12th 2020

3.2.10 Configurations

MASSim will ask for a `JSON`-file as configuration to start a simulation. The `JSON`-file includes following information:

- Information about the server that will be used in all simulations.
- Information about the tournament. Who should compete against who and in which order.
- Information about every simulation that will be performed.
- Information about the teams used to authenticate a connection between an agent and an entity.

The `JSON`-file must be in the same folder as the `.jar`-file or in a subfolder of this folder.

3.2.11 Communication

The communication between GOAL and MASSim is controlled by EISMASSim (Environment Interface Standard Multi-Agent System Simulation). EISMASSim maps the communication between GOAL and MASSim to Java method calls. EISMASSim works as a proxy for the environment on the client side, which handles the communication to MASSim by itself. The communication includes information used to authenticate an agent when connecting, start/end-signals at the start/end of a simulation, information when all simulations are done and the messages between agent and entity showed at fig. 2.1. EISMASSim needs a configuration file as well.This configuration file is a `JSON`-file. The file contains information about the MASSim server and information about the entities that the agents will connect to. The EISMASSim file and the corresponding configuration file should be located in the same folder as all the GOAL modules.

3.3 Qualification

To qualify for the MAPC 2020 a team needs to get points in two different simulations. One with fifteen agents and the second with fifty agents. The team isn't allowed to have more than 30% of the simulation steps in both simulations result in `No_Action`.

4 GOAL-DTU

GOAL-DTU qualified for the MAPC 2020, and this thesis is about improving the MAS used for the qualification. In the previous chapters, entities and agents were separated to make the explanation more clear. But for the rest of this thesis, the term, agent, will refer to the unity of the agent and the entity, to remove a layer of distraction. Entity will still be used but only in the reference of a percept.

4.1 Agents

GOAL-DTU is only having one definition of an agent called, `universal`. In simulations the universal agent is split into three types of agents if no more than fifteen agents are present on a team, otherwise, the split results in four types of agents. The types are:

- Master-agent: Master-agent creates and delegates taskplans. The agent called GOAL-DTU-1 is hardcoded to be the master-agent.
- Active-agent: Active-agent tries to solve tasks and earn points for the team. Agents with the names GOAL-DTU-1, ..., GOAL-DTU-15 are active-agents.
- Inactive-agent: Inactive-agents will spread out as much as possible through the first fifty simulation steps. Thereafter they will skip. An agent is an inactive-agent if the agent is not an active-agent.
- Submit-agent: An active-agent, which knows the location of a task board and a goal cell, is chosen to be the submit-agent throughout a task.

Inactive-agents are necessary because the MAS is executed on one computer. The operations to decide on an action for all fifty agents will take more time than four seconds, which is the time interval for choosing an action to perform in the next simulation step. If no action has been chosen a `No_Action` is given. Recall that GOAL-DTU weren't allowed to have more than 30% of the simulation steps result in `No_Action` to qualify.

4.2 Strategy

The strategy used by GOAL-DTU to qualify is based on all agents moving around semi-randomly trying to get four blocks attached and exploring the grid. An agent sends information to the master-agent when the agent has at least one attachment. Agents that are suitable to be a submit-agent, will also send a message with the names of previous agents met. The master-agent can use this information together with the perceived tasks to create plans for tasks. This kind of plan will be called a taskplan.

4.2.1 Taskplans

A taskplan is composed of the name of a specific task, a list containing a plan for each of the contributing agents, and the name of the submit-agent. The other contributing agents must be known by the submit-agent and have the correct attachments according to the task to be a part of the taskplan. The master-agent creates at most one taskplan per cycle. If the master-agent finds a taskplan then a message with the taskplan will be sent to each of the contributing agents.

4.2.2 Following a Taskplan

When the submit-agent receives a taskplan it calculates the location of the nearest task board and from the task board the location of the nearest goal cell. The location of the goal cell is sent to the other contributing agents, such that all the agents know where to meet.

All agents will detach all unnecessary blocks as soon as possible when receiving a taskplan because it's easier to move around with one attachment compared to multiple attachments. Multiple taskplans can be active at a time.

When the contributing agents meet at the meeting point, the agents can't be certain that they are sharing the same taskplan because the percept about an entity doesn't specify the name of the entity, but only the team. Entities of the same team with a block used in the taskplan are assumed to be a contributing agent. The agents will try to connect according to the taskplan. They are connecting by moving towards each other until their blocks are adjacent.

4.3 Coordination of Agents

The implementations that are used to coordinate agents are described in this section. These implementations ensure that the agents will act by specific rules.

4.3.1 Coordinate System

Every percept that has a location is local to the agent. These percepts are inserted in the belief base without translating the coordinates. The locations of the beliefs are updated when the agent moves. The agent keeps track of its location relative to its spawn location. The spawn location is the origin of the agent's coordinate system. The origin is used to translate other agents' beliefs and location into the agent's coordinate system.

Translation A translation can only take place if the agent knows the origin of the sender's coordinate system. When the agent knows the distance between its origin and the sender's origin, then the agent can simply add the distance to the received coordinates to locate the belief relative to its origin.

4.3.2 Meeting Agents

Agents share information to get knowledge of unexplored cells. The information sharing happens when an agent perceives an entity of the same team. The agent collects all the percepts shared with the entity in a message together with information about task boards, origins of other agents, and the agent's origin. The message is sent to all other agents on the team. As the first thing to do in the next cycle the agent will check if it got a message from another agent with matching percepts. If this is the case the information in the message will be translated into the agent's coordinate system using the other agent's origin and then inserted in the belief base. The two agents are then connected and can translate each other's coordinates.

4.3.3 Messages

Messages are used by agents to communicate internally and are assumed to be one simulation step delayed because an agent will first check its mailbox in the next cycle. But an agent can receive a message sent in the same simulation step. This happens when an agent starts a cycle and sends a message to an agent that hasn't started its cycle yet. The receiving agent starts its cycle before the simulation step is over. Now the receiving agent has received a message in the same simulation step as it was sent. To solve this a timestamp and the name of the sender are attached to all messages where the delay is crucial. The timestamp is equal to the current simulation step when the message was sent. If an agent receives a message with a timestamp equal to the current simulation step, then the agent will send the message to itself. The message will then be received in the next cycle with the assumed delay.

4.3.4 Movement

An agent remembers which cells it has passed through since last `attach`, `detach`, `submit` or `accept` action. The agent uses heuristic functions to give each adjacent cell a value. The agent will move into the cell with the lowest value. If multiple cells have the lowest value, a random choice will be made between them.

Moving Towards a Target Location If an agent has a target location it wants to reach, then the heuristic function in eq. (4.1) is used to calculate the value of the cell.

$$H = D + \sqrt{N} \tag{4.1}$$

H is the heuristic value, *D* is the distance to the target location from the cell, *N* is how many times the agent has passed through the cell. So the heuristic value tells how well an agent will do, if it entered this cell, according to reaching a target location. A low value signals a good cell to move into because the agent wants to move closer to the target location without moving into the same cells several times.

Exploring The agent is exploring when not moving towards a target location. While exploring the agent is also searching for dispensers to get blocks from. When exploring the cells will get a value calculated by the heuristic function in eq. (4.2).

$$H = \sum_{x=-30}^{30} \sum_{y=-30}^{30} \sum_{i=0}^{N} \left\{ \frac{\mid x \mid + \mid y \mid}{(S_c - S_i)^2} \mid\mid x \mid + \mid y \mid \leq 30 \right\} \tag{4.2}$$

H is the heuristic value, *x,y* are coordinates of cells that aren't further away than thirty steps. *N* is how many times the agent has visited the cell located at (*x,y*). S_c is the current simulation step and S_i is the simulation step when the agent visited the cell. The heuristic value tells how well an agent will explore an area if it enters this cell. A low value signals a good cell to move into because the agent wants to move into unexplored cells or cells that the agent might have outdated information about.

5 Improvements for GOAL-DTU

The purpose of the solutions explained in this chapter, is to improve the overall performance of GOAL-DTU in the MAPC.

5.1 Moving with Heuristics

Movement is critical in the MAPC 2020 because agents should complete as many tasks as possible. For every task, the agent will have to move to reach a certain location. The time used to cover this distance must be minimized because the reward of a task decays for every simulation step.

5.1.1 Problem: Movement and Environment

An agent remembers where it has been since last `attach`, `detach`, `submit` or `accept` action. It's using its knowledge about obstacles to check whether an action can be performed or not. But the knowledge about obstacles isn't used to choose an action.

An agent may move into dead-ends, when not using its knowledge. The agent will eventually move out of the dead-end. But this requires that cells in the dead-end are visited several times before the agent finds a way out. Recall the heuristic function for moving towards a target location eq. (4.1). The heuristic value of a cell is increased every time the agent visits the cell. This means that all cells in the dead-end must have a greater heuristic value than the cells not in the dead-end before the agent leaves the dead-end. On fig. 5.1.a an agent is moving into a dead-end without being aware of it. The yellow squares indicate the agent's vision. On fig. 5.1.b the agent continues to move into the dead-end but is now aware that it's inside a dead-end. The agent became aware when it was located on one of the crosses. To get out of the dead-end all the red squares need to be visited several times.

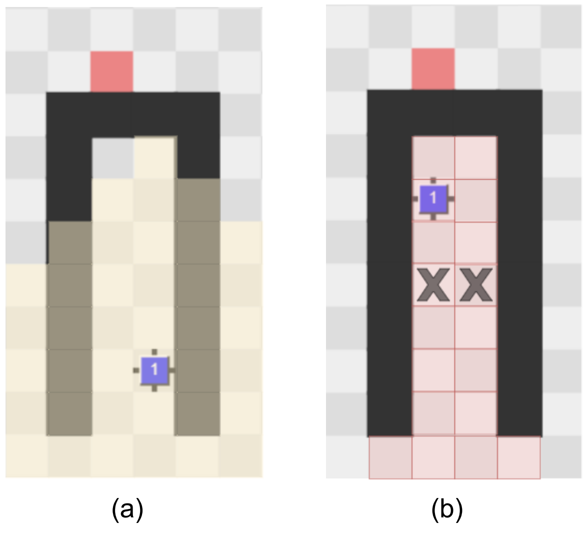


Figure 5.1: a: An agent wants to go to the goal cell (red square). The black squares are obstacles. The agent can only perceive the yellow shaded cells. The agent decides to move straight towards the goal cell. b: The agent has moved too far into the dead-end. The red shaded cells have to be visited several times to get out.

5.1.2 Solution: A*

The problem was caused by the agent not using its knowledge about the environment and by using a heuristic function that doesn't take the number of steps used into account. It's not possible to avoid going into unexplored dead-ends. But an agent that uses its percepts to make the next move will turn around when it has confirmed that it's inside a dead-end.

The cells of the grid can be modeled as a graph, where a cell is connected to its four adjacent cells. Meaning that an agent can move from cell A to cell B and from B to A. A* is a complete search algorithm that can be used on graphs. The algorithm will always find a path if one exists because it's complete. The heuristic function, $f(x)$, is defined by eq. (5.1), where $g(x)$ is the step cost to reach the cell, x, from the start cell, C_0, and $h(x)$ is the euclidean distance from x to the target cell, C_t. The step cost is equal to the number of cycles it will take before reaching x.

$$f(x) = g(x) + h(x) \tag{5.1}$$

When using A*, an agent will first find a path to the target location, C_t, before moving. A* starts by expanding C_0. This means $f(x)$ of adjacent cells to C_0 are calculated, and for each cell, an object is created and added to the frontier. The frontier includes cells that may be the next cell to be expanded. The object contains the cell, the heuristic value, and the path used to reach the cell. The cell in the frontier with the lowest heuristic value is continuously expanded until C_t is expanded. Note that objects containing the same cell can be present in the frontier because a cell can be reached by different paths.

$h(x)$ must be consistent for A* being optimal. An optimal search algorithm will always find the shortest path. $h(x)$ is consistent if, for every cell, C_i, and every successor C_i' of C_i created by any move, m, the estimated cost of reaching the goal from C_i is no greater than the step cost of m added with the estimated cost of reaching the goal from C_i'. See Russell and Norvig 2010, Chapter 3.5.2 for a full description on A*.

5.1.3 A* and the MAPC

A few things have to be kept in mind when using A* in the environment of the MAPC.

- The locations of obstacles have to be memorized.
- An agent may have attachments and can't make the same moves as an agent with no attachments.
- The environment is dynamic, so the path may contain impassable cells.
- Clear actions can remove dead-ends. But the energy and how many times a clear action must be performed needs to be calculated.

The location of obstacles must be memorized. This is necessary if the agent should find a passable path outside its vision.

Instead of using cells as the nodes of the graph, states can be used. A state must include the locations of the agent and its attachments because an agent with attachments will change state after a `Rotate` action. On some paths, agents with attachments must rotate to reach C_t; agents with no attachments will never have a reason to rotate.

Since states are used instead of cells, the grid is modeled as a tree instead of a graph. An agent must never go back to an expanded state in a tree. This means a list with all previous expanded states must be included in a state.

Removing obstacles with `Clear` actions may be faster than moving around obstacles. This will be effective if thin but long groups of obstacles are present. Before clearing an agent must be certain that it has enough energy to clear the obstacles. This can be calculated by using the current energy level of the agent and two configurable variables,`clearSteps` and `clearEnergyCost`, that are included in the simulation configuration file. Thus the energy level of the agent and a list of previously cleared cells must be included in a state.

An agent should calculate the path once and follow it until it reaches C_t. This will cause the agent to move all the way into a dead-end before turning around. This solution will save a lot of computation time when calculating more complex paths compared to a solution, where an agent should calculate a new path for every step. Figure 5.2 shows the difference between the two approaches in the same scenario as fig. 5.1. The agent at fig. 5.2.`a` calculates A* twenty times. This is a lot compared to the agent at fig. 5.2.`b` that calculates A* twice. If an agent can't perform an action according to the plan, then the agent should calculate a new path. The environment may have changed such that a cell on the path has become impassable. The agent has updated its beliefs before calculating the new path, such that the agent will move around the impassable cell. It's a different situation if an entity is blocking the path because an entity will probably move in the next simulation step. But it's not certain because the entity could be connected to an inactive agent. So for now a new path will also be calculated if an entity blocks the path.

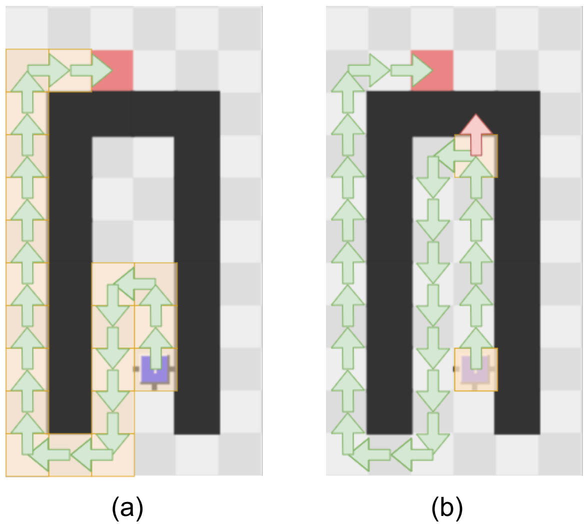


Figure 5.2: An agent wants to go to the goal cell and knows how to use A*. The agent doesn't have any energy to clear obstacles. The orange shaded cells indicate where the agent calculates A* to find a path and the green arrows indicate the agent's path. The red arrow indicates an invalid move. a: The agent calculates a new path for every step. b: The agent calculates a new path when the agent doesn't have a path or when the path can't be followed anymore.

5.2 New Communication Protocols

Sharing information is crucial when cooperating. Trivial information for one agent can be valuable information for another agent. It could be information about the environment, which is the focus of these problems.

5.2.1 Problem: Dynamic Environment

It's a problem that agents are not sharing information about obstacles. Multiple agents may move into the same alley at different times because the first agent to move into the alley didn't warn the other agents.

When an obstacle is removed a shortcut might appear. The agent, who perceives this, should inform its team, such that the whole team can take advantage of the shortcut.

The problem with obstacles is that `Clear` events are removing obstacles in an area and placing new obstacles in that area. This means an agent's beliefs about an obstacle can be outdated.

GOAL-DTU decided on a conservative approach regarding information about obstacles. Agents of GOAL-DTU's MAS are not remembering the location of obstacles. They only know the obstacles they perceive in each cycle.

The solution in section 5.1.2 also needs the obstacles to be memorized to work correctly and is most efficient if the agent has an overview of the whole grid. Unfortunately, it will take a long time to explore the whole grid and a lot of effort to keep knowledge updated. Instead, the agent can get information from connected teammates.

5.2.2 Solution: Communication

Communication between agents will be a solution to the problem stated in section 5.2.1. An agent is already sending its updated location relative to its spawn location to connected teammates every time it performs a `Move` action. The teammates know the distance from their spawn location to the sender's spawn location, such they can translate information received from the sender. This functionality can be used to transfer knowledge about dynamic and static environment as well.

When agents meet they can have different beliefs about how the environment looks like. One of the agent's beliefs might be outdated compared to the other agent's beliefs, they can't decide who is right without a timestamp. Beliefs about obstacles should include a timestamp that indicates the simulation step of the latest confirmation of the belief and whether the obstacle is present or not. This means that an agent should tell connected teammates when the agent acquires information about an obstacle or a cell, that once was an obstacle, no matter if the information is new or not. This will sum up a lot of messages being sent back and forth between everyone. A on fig. 5.3 are using this approach.

To avoid spamming the communication channels an agent should instead send a message to connected teammates when it perceives a cell that contradicts its beliefs. B on fig. 5.3 are using this approach. The message should include the coordinates of the cell, a timestamp, and whether an obstacle was present or not. When an agent perceives a cell that confirms a belief of the agent, then the timestamp of the belief should be updated. The agent shouldn't send this information to anyone.

Now when agents meet for the first time and they have contradicting beliefs about the environment, they can agree on the most updated environment and send new information to their connected teammates. The teammates are comparing the new information with their own beliefs. If a teammate has a more updated belief that contradicts the information received, a message is sent to its connected teammates with the updated information. This results in a lot of messages being sent when agents meet but will also reduce the number of messages when agents are not meeting each other.

5.2.3 Problem: Meeting Agents

The agents of GOAL-DTU's MAS are sending messages to all other agents on the team when perceiving another agent as explained in section 4.3.2. These messages are used to connect agents.

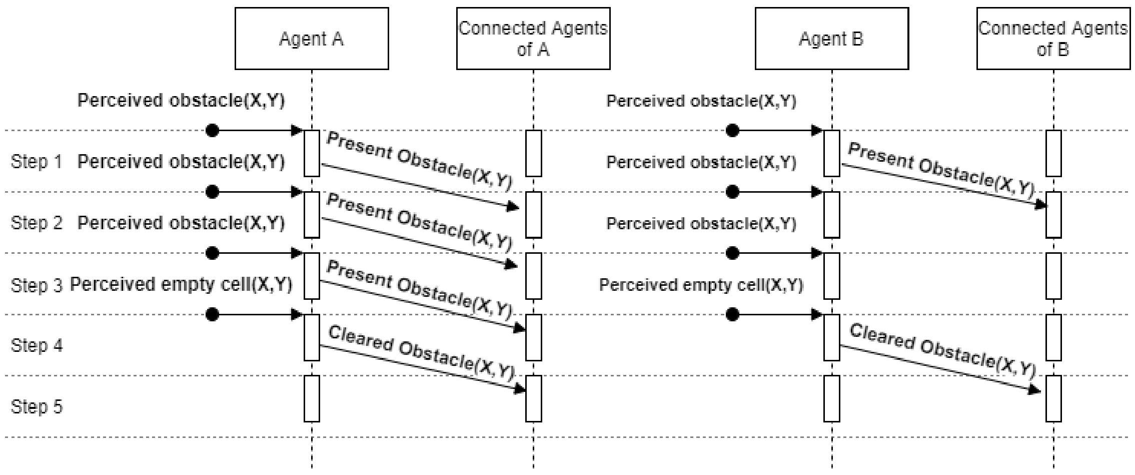


Figure 5.3: A sends information about an obstacle located at (X, Y) every cycle no matter if the information is new or old. B sends information about the same obstacle but only when the appearance of the obstacle changes. B is sending two messages less than A. Connected agents of both agents are equally updated on the appearance of the obstacle. The blocks below each agent indicate a cycle.

A problem occurs when one of them can begin a new cycle before the other has sent the message with the percepts they have in common. The slow agent will successfully connect to the other agent, and get all the information that the other agent has. The fast agent will fail to connect because it's connecting with more updated percepts. This means the fast agent won't get the information from the slow agent. The problem is shown on fig. 5.4.

GOAL-DTU ensures that both agents know each other's origin, by sending the origin offset after an agent completes the connection. On fig. 5.4 B won't insert the information into its belief base because B receives matching percepts from A one cycle late. Instead, A sends the origin offset between their origins such that B can translate messages from A. But B is missing out on a lot of information. This means that agents that are connected to B will miss out on the information as well, because of the solution in section 5.2.2.

5.2.4 Solution: Meeting Agents

At the moment an agent uses two to three cycles when connecting to other agents. One cycle for sending a message with percepts, one for receiving the percepts and sending a message with the origin offset, and one to receive the origin offset. Maybe it's a better idea to use one more cycle, to make sure the other agent has received the information?

The origin offset will be sent no matter when the connection is complete. So this message can include the information of the first message, to ensure that the other agent received it. This message is only sent to one agent, so it won't be spam. This will result in a lot of duplicated data being transferred in the first and last message. It doesn't make a big difference if the agent has to wait for an additional cycle to receive the information, thus the information shouldn't be included in the first message. Figure 5.5 shows this solution.

This solution will use one more step but now an agent won't miss out on any information and it can always share new information with its connected teammates when connecting with a new agent.

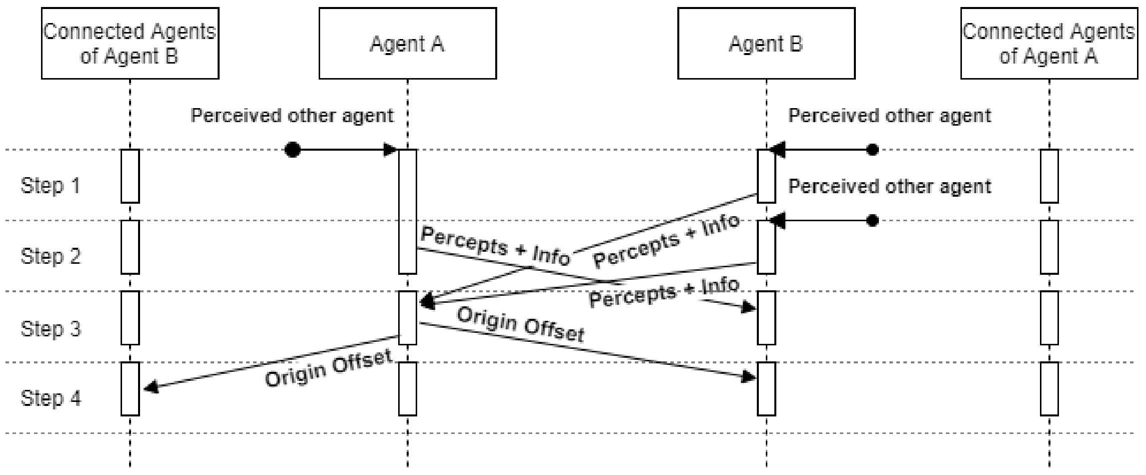


Figure 5.4: Situation where A and B try to connect but only A succeeds. The blocks below each agent indicates a cycle. A sends its origin offset to B after completing the connection.

5.3 Preparing for Tasks

A good strategy is important for a MAS. It defines how agents work together and has a huge impact on the final result. Many hours and days can be used to find a good strategy. In this section, a strategy won't be explained. But a foundation that can be used for multiple strategies will be explained.

5.3.1 Problem: Distances When Completing a Task

The agents are not waiting for a taskplan; they are moveing around and exploring the map. When a submit-agent receives a taskplan it must go to a taskboard, accept the task and then go to a specified meeting point in a goal zone. This is shown at fig. 5.6, where agent 2 is the submit-agent. The other agents on the taskplan just have to go to the meeting point. This may cause the other agents to wait for the submit-agent at the meeting point for an unnecessarily large amount of time. In some scenarios, a submit-agent waits for several simulation steps for other agents to arrive because the other agents were on the other side of the map.

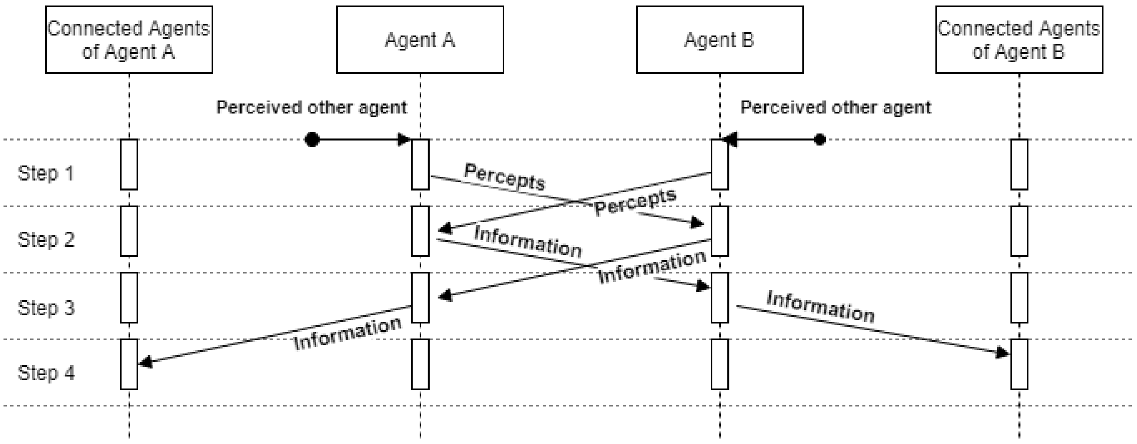
The agents are detaching unnecessary blocks when they are assigned to a taskplan. Agents can't move through blocks and have to move around or clear them. This causes an extra distance to cover when moving towards the meeting point. Distances from agents to a meeting point should be as short as possible because the completion time of a task is proportional to the number of cycles needed for all agents to arrive at the meeting point.

5.3.2 Solution: More Types of Agents

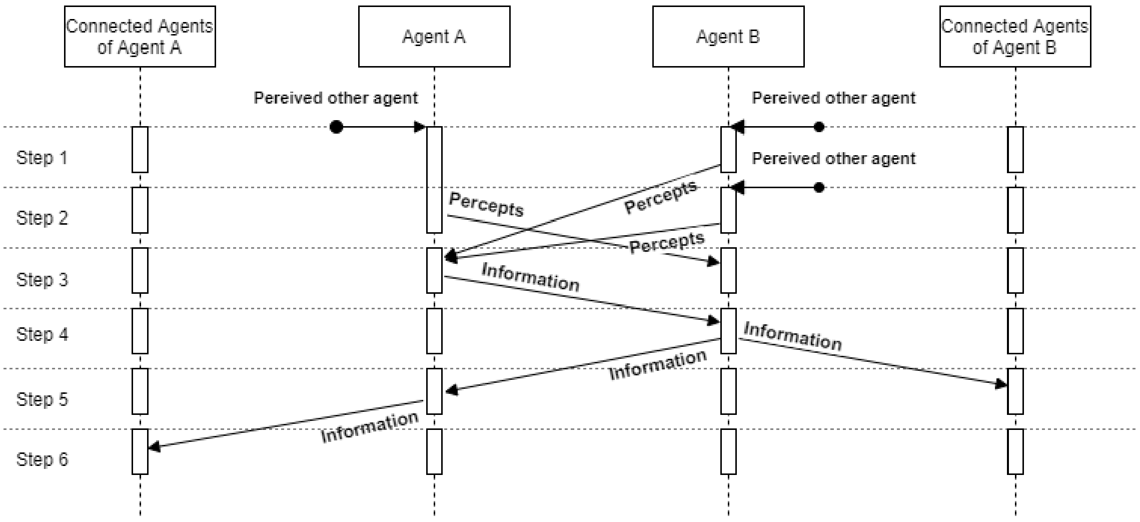
GOAL-DTU used four different types of agents in their MAS. These types are described at section 4.1. This solution will use more types because it will give agents time to prepare for their specific tasks in a taskplan. Two new types of agents are block-agent and explore-agent. The submit-agent and master-agent have minor changes. The relationships between the types is visualized at fig. 5.7.

Explore-agent Explore-agent has the job to explore the map and find teammates. The explore agent should eventually stop exploring and evolve into a block-agent or submit-agent when all of the following is true:

- Connected to an agent for each dispenser type known.
- Knows the locations of a task board and a goal zone.



(a)



(b)

Figure 5.5: a: A and B connect without problems because their cycles are synchronized. b: B has two cycles while A only has one cycle. This makes it impossible for B to connect to A on its own. The information sent by A will make it possible for B to connect to A, such that no information is missed.

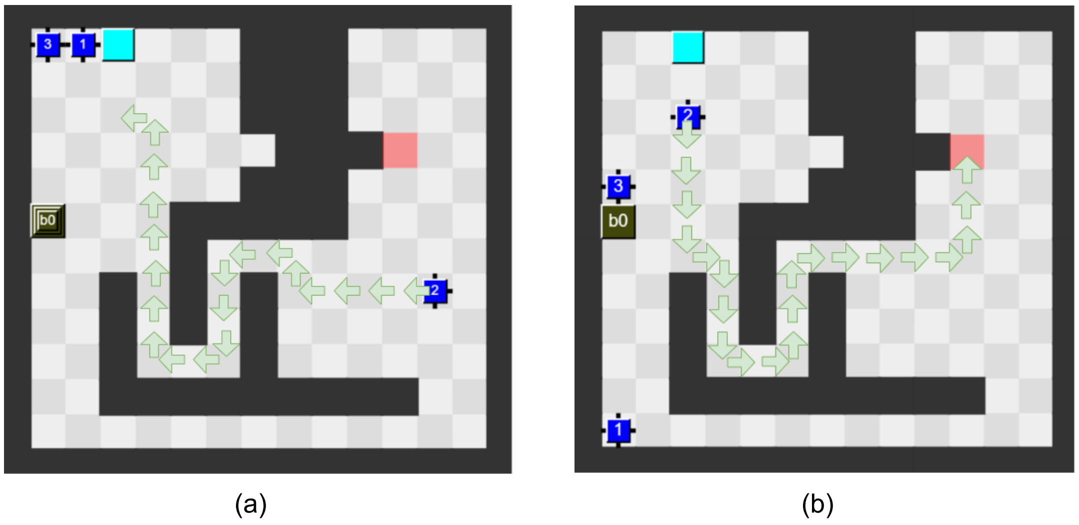


Figure 5.6: a: A taskplan is made with agent 2 and agent 3. The submit-agent, Agent 2, has to get in range of a taskboard to accept a task. b. After accepting a task the submit-agent has to go to the goal cell.

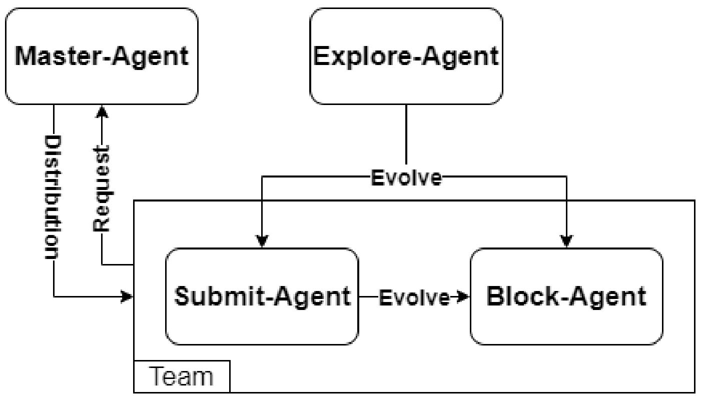


Figure 5.7: Explore-agents will eventually evolve into a block-agent or a submit-agent and be a part of a team that completes tasks. The master-agent won't change type throughout a simulation and is not a part of a team.

Submit-agent Submit-agent will wait in range of a task board ready to accept a task.

Block-agent Block-agent has the job to stand by a dispenser with a block, of the same type, on top of the dispenser, waiting for further commands from a submit-agent. The block is positioned on top of the dispenser to make it impossible for other agents to use the dispenser.

Master-agent Master-agent is responsible for allocating types to explore-agents, block-agents and submit-agents.

This makes sure that a submit-agent can skip the distance to the taskboard and go directly to the meeting point because it's already in range of a taskboard. Agent 2 could have skipped fig. 5.6.a and started at fig. 5.6.b if it was prepared. Furthermore, block-agents won't create extra obstacles by detaching blocks, because they are not attaching more than one block before further commands are received.

The first team is created by *N+1* explore-agents, because the team knows *N* dispensers. The team has a team leader. The team leader's only job is to request a distribution of types from the master-agent when new types are needed. The agent with the lowest number in its name is chosen as the team leader. Because of the solutions in section 5.2 the team leader knows the locations of all connected agents such that it can send their location to the master-agent together with information about dispensers and task boards. The master-agent uses the locations to find a distribution of types in the team. The types are chosen such that the team is ready for tasks as soon as possible. This is done by minimizing the euclidean distance from the furthest agent to its wanted dispenser or task board.

A team is started with *N* block-agents and one submit-agent. One dispenser type for each block-agent. The team will grow dynamically throughout the simulation. The team prioritizes to have *N* block-agents for every submit-agent and to get the submit-agent before its corresponding block-agents. This means that the next explore-agent to join the team evolves into a submit-agent. The reason is that many small tasks are present compared to big tasks. A strategy can be built on top of this foundation that makes it possible for one block-agent and one submit-agent to complete tasks of up to four identical blocks. Requests from existing teams with multiple explore-agents will be handled by minimizing the shortest euclidean distance from an explore-agent to its wanted dispenser or task board.

A scenario may occur where two teams of *N* block-agents and two submit-agents merge into one team. This results in a team with *2*N* block-agents and four submit-agents. One of the submit-agents has to evolve into a block-agent. The team leader of the merged team requests the master-agent for a new type to one of the submit-agents. The submit-agent that changes type must not be on a taskplan and should be closest to a dispenser. If multiple submit-agents match the criteria one of them is chosen randomly.

6 Implementations of Improvements

The solutions will be implemented in GOAL and Prolog.

6.1 A*

In section 5.1.2 a solution to the movement problem was given. The solution used the A* algorithm to create a path from one point to another point for an agent.

6.1.1 Knowledge Base

The search algorithm is implemented in the knowledge base of the agent.

Frontier Like many search algorithms, A* uses a frontier. Expanding and popping of the frontier happens several times through A*. The time used on these operations must be minimized to achieve the best implementation. A pairing-heap is used as frontier because the state with the lowest heuristic value is always popped from the frontier. A pairing-heap uses logarithmic time, $O(log(n))$, to pop the state with the lowest heuristic value. And it uses constant time, $\Theta(1)$, when inserting a new state in the frontier. For further information see: Fredman et al. 1986. Prolog has a predefined predicate called `heaps`, that implements pairing heaps.

Expanding as Few States as Possible Expanding states takes time. The fastest A* algorithm expands as few states as possible. When popping the next state off the frontier, the state with the lowest heuristic value is popped. If multiple states have the lowest heuristic value, then normally the state that was added first to the heap will be popped, following the principle of FIFO. Imagine a grid of NxM cells with no obstacles, where $N, M \in \mathbb{N}$. The agent is positioned at (0,0) with no attachments and wants to go to (N, M). States and cells have a one to one relationship in this case. All states will have the same heuristic value because no obstacles are present. This means that all other states are expanded before the last state to be added. The last state added to the frontier is the goal state at (N, M). It's extremely important that if multiple states share the lowest heuristic value then the latest added state must be popped, following the principle of LIFO. It will make sure $N + M$ states were expanded in this scenario compared to $N \cdot M$ states.

States A state needs to store information when being popped from the frontier, such that the search can switch to the context of the state. The state must store the following information:

1. Location of previously cleared cells. Coordinates are relative to the spawn location.
2. Location of the agent. Coordinates are relative to the spawn location.
3. Locations of the agent's attachments. Coordinates are local to the agent's location.
4. A list of actions.
5. A list of previously visited locations. Coordinates are relative to the spawn location.
6. The agent's energy level.
7. The step cost, $g(x)$ to reach the cell.

The first three attributes are used to figure which states that the agent can move into from the state. The list of actions is returned when the search has reached the goal state. An agent can reach the goal by performing the actions in sequence. The list of previously visited locations is used to validate that the actions are performed correctly. The agent's energy level may vary through the search because of `Clear` actions. It's important to know the agent's energy level before clearing an area since a certain amount of energy is needed to clear an area. $g(x)$ of the state doesn't need to be stored in the state but will be a fast and simple solution to find $g(x)$ of the state. An alternative way is to go through the list of actions and calculate how many steps an agent will use to complete all actions.

Expanded States The algorithm has a list of previous expanded states such that states in this list won't be added to the frontier again. This ensures that the agent is not entering a loop. This list is implemented as a simple list. It takes linear time, $\Theta(n)$, to check whether a state has already been expanded. Prolog had no predefined predicates for a hash-list, but a hash-list is preferable since a state could be checked in constant time, $O(1)$ in the best case. Only the most defining attributes of a state will be inserted in the list of previous expanded states. For example, the energy level of the agent is included in a state of the frontier. If the energy level is included in a state of the list of previous expanded states, then two states may be present in the frontier, where the only difference is the energy level. This is not good because an agent can move back and forth between two cells while the agent's energy level is increasing for each step, instead of facing the reality and move to a state with a worse heuristic value. An agent should use the `Skip` action if the agent waits for enough energy to clear an area. The current version of the solution doesn't include the `Skip` action. The most defining attributes of a state in this algorithm are the locations of the agent, its attachments, and previously cleared cells. States in the list of previous expanded states will only include these three attributes. These three attributes are also the results of the three actions, `Move(D)`, `Rotate(D)` and `Clear(X,Y)`, that are included in the algorithm.

Actions The algorithm expands the frontier with the resulting states that an agent can reach from the last popped state by performing one of three actions. The locations of the agent's attachments are used to validate if the agent can move or rotate in a direction without an attachment is blocked. GOAL-DTU has already implemented this validation. This validation uses the locations of the attachments local to the agent's location. This means that the coordinates of the attachments are kept local to the agent's location throughout the search and are only updated when the agent wants to rotate. This is possible because the attachments are moved alongside the agent, which is different from obstacles and cleared cells. Obstacles and cleared cells are statically throughout the search. If they should be local to the agent's location then all obstacles and cleared cells should be updated every time a `Move` action is chosen in the search. This means their location will change through the search, so all obstacles and cleared cells must be included in every state of the search. It can't be avoided that a state must include a list of cleared cells, because cleared cells are created throughout the search. But it can be avoided to update cleared cells location and include obstacles in states by using coordinates relative to the agent's spawn location for their locations. The agent's location must be relative to the agent's spawn location for this to succeed. Now only the agent's location is updated for every `Move` action instead of all locations of obstacles and cleared cells.

The step cost, $g(x)$, varies for the actions. `Move` and `Rotate` actions always use one step to complete, while `Clear` action use the number of steps equal to the configurable variable `clearSteps`. It's not possible to fetch `clearSteps` nor the configurable variable clearEnergyCost. So an initial belief of clearSteps and clearEnergyCost are inserted into the belief base in the initial module. These beliefs are updated when the agent receives the percepts after completing the first Clear action. It's important that the values of these beliefs are optimistic. If the values are to high the Clear action will never be performed.

6.1.2 Implementation in GOAL

GOAL uses percepts to create the beliefs and goals that are used to find an efficient path with the A* solution.

As mentioned earlier initial beliefs of clearSteps and clearEnergyCost are inserted into the belief base in the initial module, such that the agent can calculate an approximate $g(x)$ for the first Clear action.

The agent's percepts have to be handled in this sequence:

1. Percepts about the environment that will be deleted and inserted every cycle.
2. Percepts about the latest action.
3. Percepts about the environment that won't be deleted and inserted every cycle.

Percepts about the latest action have to be handled after updating the energy level and before updating beliefs that use coordinates relative to the agent's spawn location. The energy level should be updated every cycle independently of the latest action performed. A Clear action has been completed when the energy level drops. A flag can be set to indicate that the action is complete, such that the agent can continue to the next action. The agent counts how many Clear actions must be performed before the area is cleared. The agent uses this information together with the drop of energy to calculate precise beliefs of clearSteps and clearEnergyCost. The percepts about the latest action, are used to update the agent's location together with many other things. The agent perceives the environment local to its location. This means that the agent's location must be updated before inserting beliefs that use the agent's spawn location as the origin. The agent uses its location to calculate the offset of the environmental beliefs.

An agent tries to find a path with Prolog when the agent has a goal to be somewhere else and don't know a path. The agent will delete the path from its belief base and find a new path if the first action in the list of actions fails with a different error than failed_random. The agent will also delete the path when the agent achieved its goal.

The agent will perform the first action in the list of actions in the main module. In the next cycle, the agent will receive a percept about whether the action was performed successfully or not. If the action was performed successfully then the action will be removed from the list of actions, such that another action will be the first in the list. The Clear action won't be deleted from the list before a drop in energy has been perceived.

6.2 Information Sharing

An improvement to the communication between agents was explained in section 5.2.2. The following section will be about how the communication protocols are implemented.

6.2.1 Modules

The communication protocols are implemented in the event module. Sub-modules in the event module are created to maintain a clean structure. This will help with navigating in the code and keeping a clear idea of the agent's flow through a cycle. Figure 6.1 shows the sub-modules in the event module.

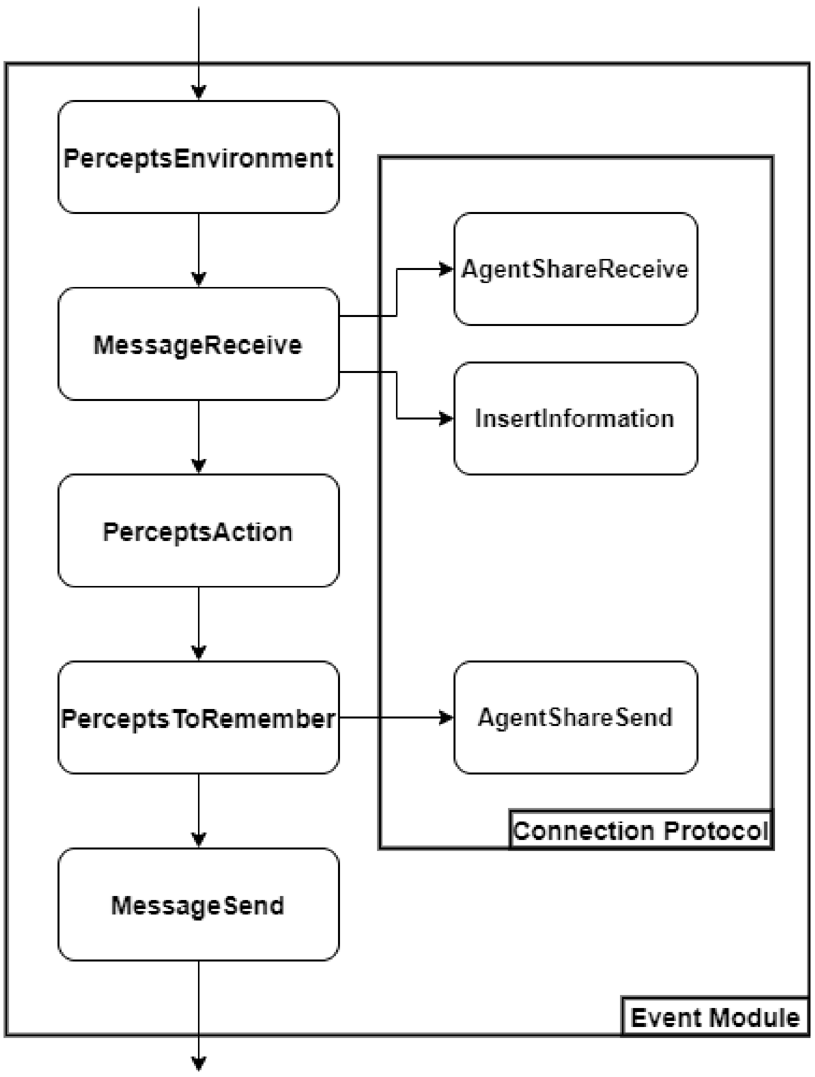


Figure 6.1: The event module is split up in smaller modules that each has a special purpose.

- `PerceptsEnvironment` is used for percepts that will be deleted and inserted into the belief base every cycle.
- `Messagereceive` is used to receive messages from other agents.
- `AgentShareReceive` is used to validate percepts received from other agents in the connection protocol.
- `InsertInformation` is used to handle information received from other agents in the connection protocol.
- `PerceptsAction` is used to handle percepts related to actions.
- `PerceptsToRemember` is used for percepts that won't be deleted and inserted into the belief base every cycle.
- `AgentShareSend` is used to find percepts that the agent has in common with another agent in the connection protocol.
- `MessageSend` is used to share information with other agents.

6.2.2 Sending Messages

This improvement causes the agent to send a lot of messages to its team. The message might be generated in different modules, such that the connected agents must be found multiple times. To avoid this all messages are sent in `MessageSend` that only has the purpose to send the messages.

`MessageSend` is entered at the end of the event module if the agent wants to send a message. A new compound `shareInfo` is added. An agent inserts a `shareInfo` belief into the belief base when the agent wants to send a message. This belief contains the message and an atom that tells who the message must be sent to. If the atom is unified with `team` then the message is sent to all connected agents, together with the agent's name and the current simulation step. The atom can also be unified with the name of a specific agent or the `allother` channel. The `shareInfo` belief is deleted when a corresponding message is sent.

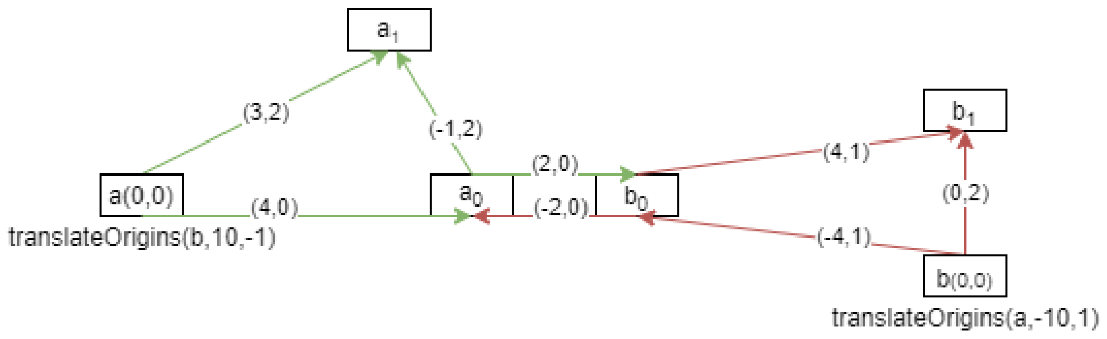
6.2.3 Obstacles

How obstacles are represented in GOAL-DTU's MAS must be changed such that agents can agree on the most updated environment that they can create together. The compound of obstacle does now have four arguments instead of two. The four arguments are the two coordinates of the obstacle relative to the agent's origin, a status that tells if the obstacle is present, and a timestamp that tells when the obstacle was perceived latest. The simulation step is used as the timestamp and the status is one of the two atoms `present` or `cleared`.

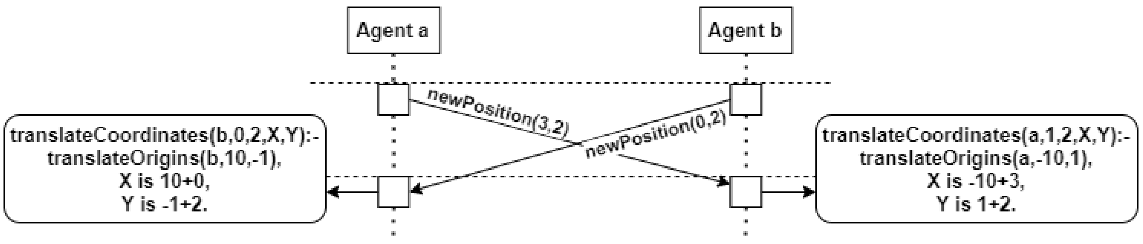
Obstacles must be perceived after the agent has updated its position after a `Move` action; the agent's location is used to calculate the position of an obstacle. This is why percepts about obstacles are handled in `PerceptsToRemember`. This module is entered after `PerceptsAction` that handles percepts related to actions.

An agent will never have two beliefs about two obstacles that share location. An old belief must always be deleted before inserting a new belief. If the agent perceives an obstacle and doesn't have a belief about the obstacle with the status `present`, then the agent updates its beliefs to match the percept.

To perceive present obstacles is straight forward, but perceiving cleared obstacles is not. It's simply not enough to query locations where no obstacle is perceived, because this



(a)



```
translateCoordinates(b,0,2,X,Y):-
translateOrigins(b,10,-1),
X is 10+0,
Y is -1+2.
```

```
translateCoordinates(a,1,2,X,Y):-
translateOrigins(a,-10,1),
X is -10+3,
Y is 1+2.
```

(b)

Figure 6.2: a: A and B have matched at position 0 in the connection protocol and have now received each other's information. The information includes the sender's location in its coordinate system. The receiver can use the distance between them and their locations in each of their coordinate systems to calculate the offset between the coordinate systems. b: Later when the agents send their new location at position 1, they can understand the received coordinates by adding the previously calculated offset.

will create free variables. The agent can find cleared obstacles after the timestamps of all the visible obstacles are updated. An obstacle has been cleared if the location of the obstacle is in vision while the timestamp is outdated. The agent updates its belief base with the new timestamps and statuses. The agent can't find cleared obstacles before the agent has perceived the present obstacles. If this happens all obstacles out of vision will have the status `cleared`.

The agent will inform its connected agents through `shareInfo`, if a new obstacle is perceived or if the status of an obstacle has changed from `present` to `cleared`.

6.2.4 Static Environment

Goal cells, dispensers, and goal cells are inserted in the belief base when they are perceived without the agent having beliefs about them. But now an agent will also share this new information with connected agents through `shareInfo`. All connected agents are now aware of this new information.

6.2.5 Receiving Messages

The agent can receive messages from connected agents. These messages can for example contain information about obstacles as mentioned in section 6.2.3. When receiving a message the timestamp is checked. The agent will send messages with timestamps equal to the current simulation step to itself. This is explained in section 4.3.3. Coordinates in the message will be translated, such that the agent can compare the information with its belief base. Figure 6.2 shows how the coordinates are translated.

If the message contains information about an obstacle, then the timestamp of the obstacle compound is checked. If the status of the obstacle contradicts with the agent's belief of the obstacle and the agent's belief is more updated, then the agent will send a message to its connected agents with its more updated information by using `shareInfo`. Otherwise, the agent will keep the most updated information in its belief base and discard outdated beliefs if any exists. The messages can also contain information about static environment (task boards, goal cells, and dispensers, origins of other agents). This information is straightforward to use because only the coordinates must be translated and then the information can be inserted in the belief base if it isn't already known. Agents are sending their location every time it changes, this is handled by translating the coordinates, deleting the old belief of the agent's location, and then inserting a new belief of the agent's new location. An agent will receive messages from other agents that have perceived another agent in the previous cycle. These messages have the purpose of connecting agents and are only handled if the agent perceived another agent in the previous cycle. The messages are handled by inserting them into the belief base and then `AgentShareReceive` will be entered, where the agent may connect to other agents.

6.2.6 Meeting Other Agents

When an agent perceives another agent, a `shareInfo` belief will be inserted with percepts the agents share, the information needed to validate the percepts, and `allother` as the channel. The agent inserts a belief with shared percepts for every agent it perceives. These beliefs are deleted after attempting to connect with new agents. This kind of belief will only be present if the agent perceived another agent in the previous cycle. And an agent won't attempt to validate any received percepts if it doesn't have any belief about shared percepts.

Then the agent will continue to act as before until the agent receives a message with information about the environment. The message can be received without the receiver being connected to the sender, but the sender must be connected to the receiver. So the sender includes the origin offset such that the receiver can translate the received information. This is showed in fig. 5.5. The information extends the information sent in GOAL-DTU's MAS, see section 4.3.2, by including the agent's beliefs about goal cells, obstacles, and dispensers.

New information is shared with connected agents. This results in every time the connection protocol is completed, two networks of connected agents are merged into one. Figure 6.3 shows how agents are connected in a network before and after the connection protocol is complete.

6.3 Preparation

How to prepare for a strategy is explained in section 5.3. This section goes through the implementation of the preparation.

6.3.1 Types

Beliefs are used to control the type of agents because types change dynamically. If they don't change dynamically new agents could have been defined by different modules in the `.mas`-file.

All agents, but the master-agent, are initialized with a belief about being an explore-agent. The belief is called `agentType(Agent, Type, Detail, Timestamp)`. An agent will also store the types of its connected agents. `Agent` is the name of the agent, `Type` is the type of the agent and `Timestamp` is the simulation step where the master-agent calculated the type. `Detail` is information about the type. `Detail` will be a block type for block-agents,

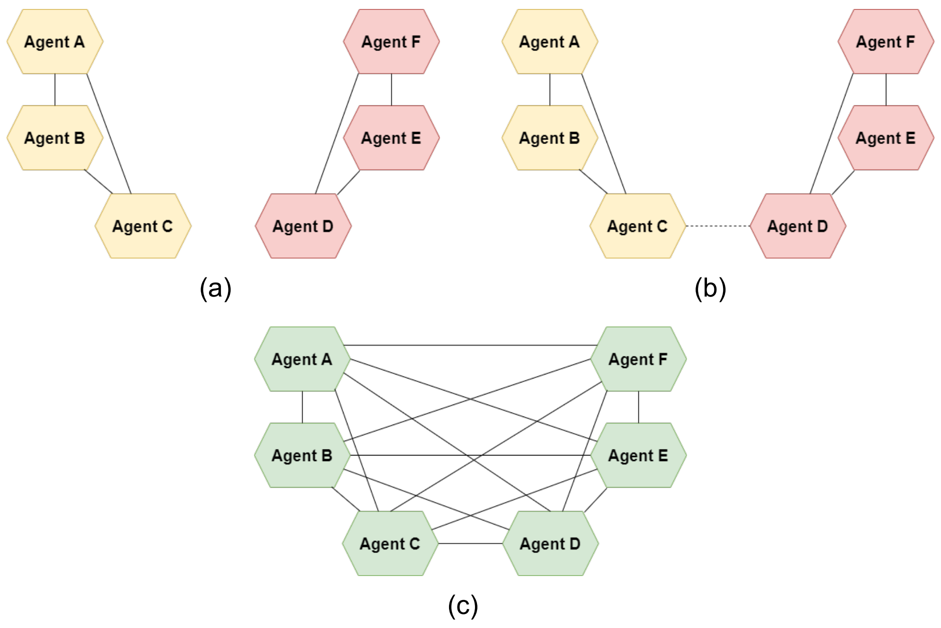


Figure 6.3: a: Two networks of connected agents. b: Agent C and Agent D perceive each other and try to connect. c: The two networks become one network after Agent C and Agent D succeed to validate each other percepts.

`tb` for submit-agents and `none` for explore-agents. The block type indicates which kind of dispenser the block-agent should be adjacent to. `tb` and `none` are constant placeholders used to match the format.

6.3.2 Protocol

The master-agent controls the allocation of types. The team leader sends a message to the master-agent when its team is ready to team up and types of its team are out of balance. The message includes necessary beliefs for the master-agent to calculate an optimal distribution. The beliefs include the locations of connected agents, types of connected agents, dispensers, and task boards. The master-agent needs this information because it's not certain that the master-agent and team leader are connected.

The predicate, `readyToTeamUp`, controls when the team is ready to team up. The predicate can be changed easily if more or fewer requirements are needed for future strategies. The predicate is true when the agent knows a dispenser, task board, goal cell and is connected to at least as many agents as known types of dispensers. The team can only be out of balance if the team is ready to team up. The predicate, `typesInBalance`, controls when the types are in balance. The predicate is true when the team has no explore-agents and when the following relationship between block-agents and submit-agents holds:

$$\frac{B}{D} \leq S \leq \frac{B}{D} + 1$$

B is the number of block-agents. D is the number of known types of dispensers. S is the number of submit-agents.

The master-agent remembers the types it allocates to agents. The reason is that the team leader will receive the response at least two cycles after it has sent the message to the master-agent. Figure 6.4 shows this delay and the protocol. The team leader sends a message to the master-agent when the types on the team are out of balance. The

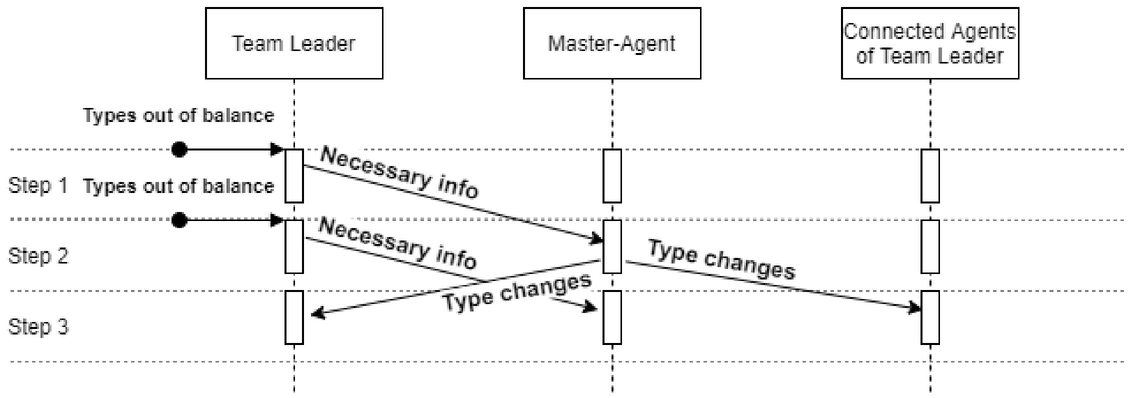


Figure 6.4: Step 1: Team leader sends a request to master-agent. Step 2: The team leader sends a new request. The master-agent sends the new types to the team leader and its connected agents. Step 3: The master-agent receives the second message. The master-agent corrects the types of agents. The corrected types are in balance so nothing more happens.

types are still out of balance in the next cycle so another message is sent. The master-agent receives the first message. It calculates the distribution of types and sends the type changes to all the connected agents of the team leader. Next cycle the master-agent will receive an identical message again. The master-agent corrects the type of each agent, such that it corresponds with the master-agents beliefs. The master-agent won't send a response to this message because the types are in balance according to its beliefs. If extra agents were on the second message types would have been allocated and sent to the agents.

6.3.3 Allocation of Types

The allocation and check of types happen inside a module, `allocateTypes`, where only the master-agent has access. This is to ensure that code isn't mixed.

The allocation of types happens after the types in the received message have been checked with the master-agent's beliefs. The allocation algorithm will be called no matter if the types are already in balance or not. If the types are already in balance an empty list of changes will be returned. Thus the master-agent has nothing to send to the agents on the team. On figure fig. 6.5 a flow chart is shown. The flow chart shows an overview of how types are being allocated.

Initialize team This box is entered if no team exists. This means that all agents are explore-agents. The explore-agents are given types such that the longest euclidean distance to a dispenser or taskboard is minimized. This is done by first finding all distances from every agent to its closest task board and closest dispenser of each type. Then all possible distributions of types are created. The distribution with the shortest longest distance is chosen. Meaning that the longest distance of the chosen distribution must not be longer than another distribution's longest distance.

Submit-agents to explore-agents This box is entered if two teams merge. Two teams merging may cause too many submit-agents. Some of the submit-agents have to evolve into block-agents. But they are first devolved into explore-agents to handle them fairly.

Block-agents don't have a corresponding box because too many block-agents will never occur. The reason is that submit-agents are prioritized before block-agents so a block-

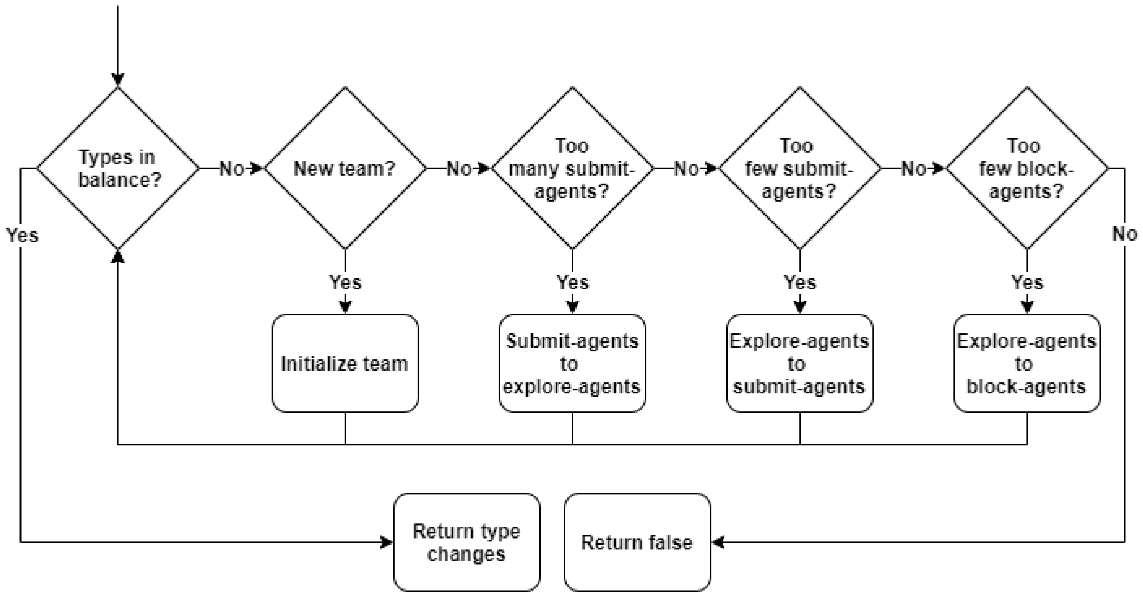


Figure 6.5: Flow chart for the allocation algorithm performed by the master-agent.

agent will always have a corresponding submit-agent. But a submit-agent will not always have N block-agents which can lead to devolving a submit-agent when merging teams, as explained earlier.

Explore-agents to submit-agents This box is entered when new explore-agents want to join a team and the team needs one or more submit-agents. The explore-agent who is closest to a task board is prioritized.

Explore-agents to block-agents This box is entered when new explore-agents want to join a team and the team needs one or more block-agents. The explore-agent who is closest to a dispenser is prioritized.

After agents have got new types they will also change goals. Block-agents will adopt a goal to stand next to the closest dispenser of a specific type while having a block of the same type attached on top of the dispenser. Submit-agents adopt a goal to be in range of the closest task board. When they have achieved their goal they will skip until further commands are received. This means that the goals need to be inserted again after leaving their location. The goals trigger the A* module, such that agents will use A* to find the fastest path to achieve their goals.

6.4 Improvements in GOAL-DTU

A* and the communication protocols are implemented in GOAL-DTU. A* is implemented to guide agents on longer distances. The MAS created by GOAL-DTU has an algorithm used to make the agents connect. This algorithm is prioritized over A* such that the agent won't follow the path found by A* when the connection algorithm is triggered. A* will be computed when agents are assigned to a taskplan and have detached unnecessary blocks.

The communication protocols are implemented to handle messages about the environment and to improve the connection protocol. The communication protocols won't handle messages about tasks because they are an integrated part of the task planning. Messages about tasks will be handled as in GOAL-DTU's MAS.

Figure 6.6 shows the structure of the code after the communication protocols and A* are implemented in GOAL-DTU. The modules, `TaskPlanning` and `TaskInformation`, are used for creating taskplans and handling the communication for tasks in GOAL-DTU's MAS.

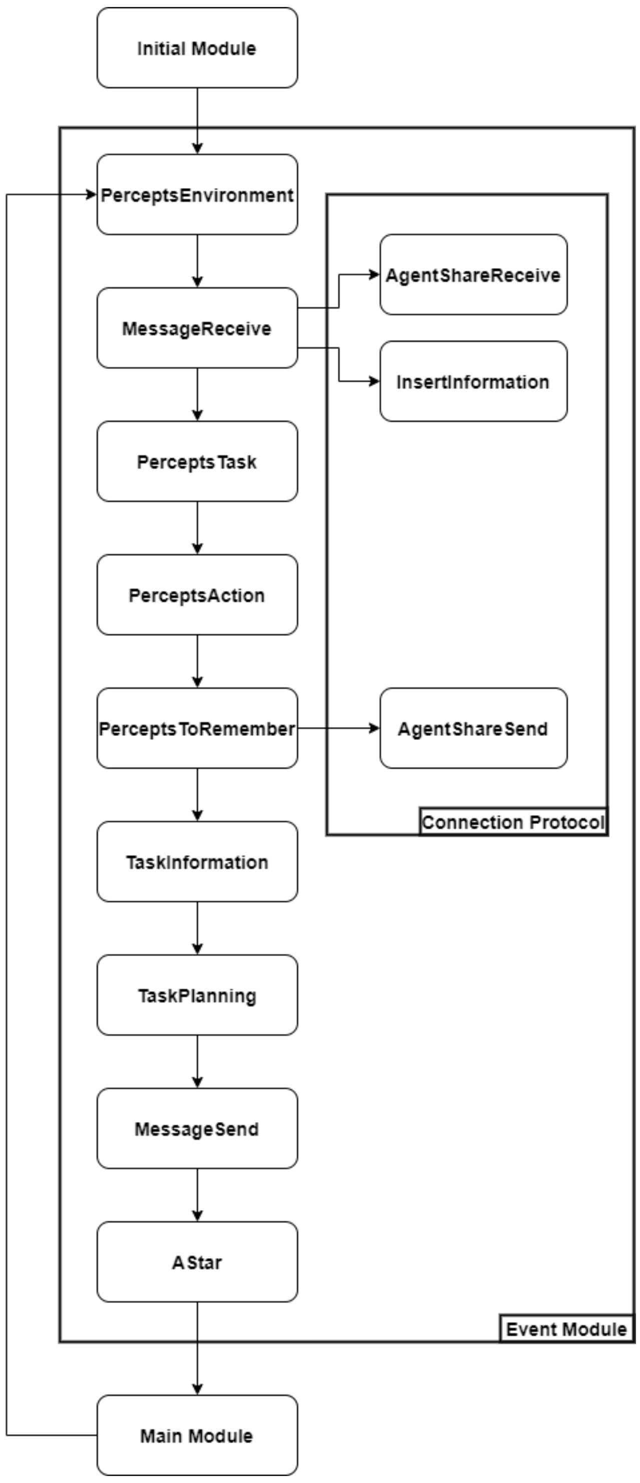


Figure 6.6: Overview of the implementation of A* and the communication protocols in GOAL-DTU.

7 Tests of Improvements

In this chapter, the implementations will be tested. Each test will focus on a specific part of the implementation.

7.1 Moving Through an Alley

Three tests are made to test each action used by the A* implementation. The same environment is used for all three tests and can be seen on fig. 7.1. The agent can receive a block from the dispenser at the top left corner and the red cell is a goal cell. The paths found by the agent are indicated by arrows. The initial goals and beliefs are changed for each test, such that the agent will perform a different sequence of actions. The agent can perceive cells that are up to five cells away.

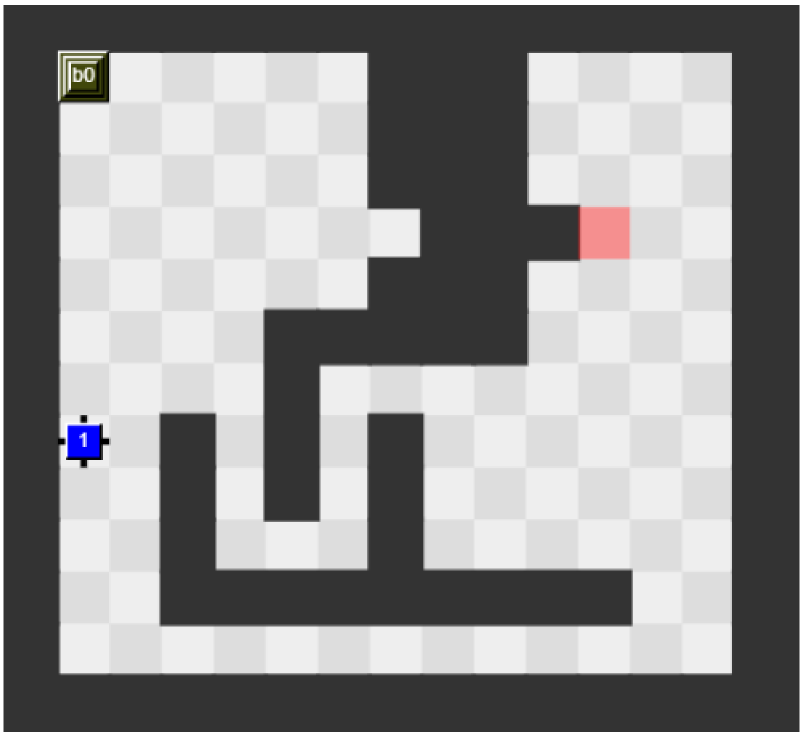


Figure 7.1: The environment used to test A*. A dispenser is located at the top left corner and a goal cell is marked with red colour.

7.1.1 The Move Action

In this test, the agent's goal is to be at the goal cell. The agent has been initialized with the belief that a `Clear` action takes twenty steps to complete to avoid `Clear` actions.

The first two paths calculated are very direct because the obstacles left to the goal cell aren't known yet. The heuristic value of the goal cell is 14 equal to the euclidean distance between the agent and the goal cell. The agent follows the two paths on fig. 7.2.a and fig. 7.2.b before the agent realizes a direct path doesn't work. The agent recalculates a new path and finds the path on fig. 7.2.c. The agent calculates the final path on fig. 7.2.d and reaches the goal. The goal cell has a heuristic value of 22 in the path found at fig. 7.2.c. The goal could also have been reached by using a `Clear` action removing the three obstacles to the right of the agent. But this path would give a heuristic value of 24 to the goal cell.

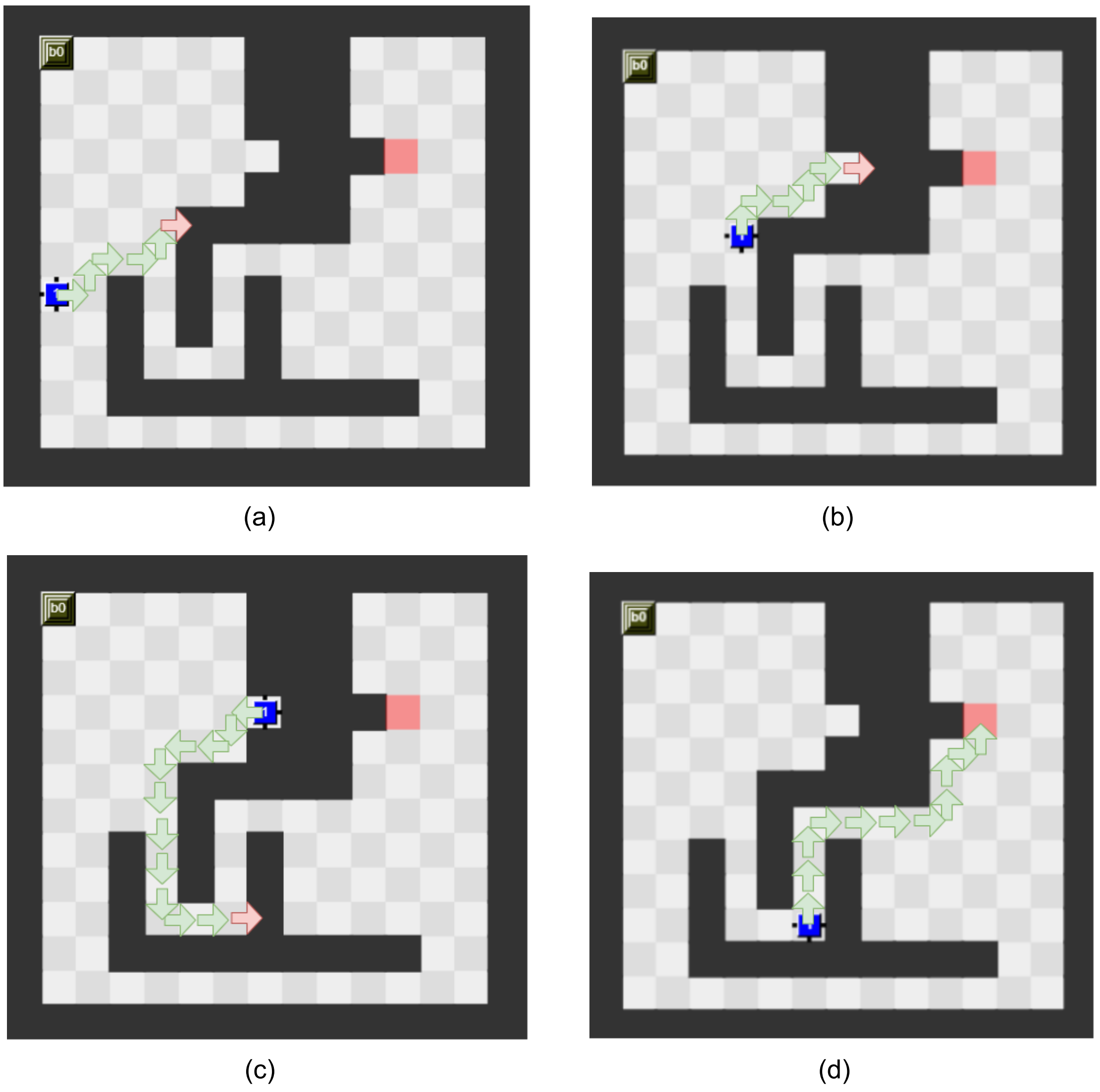


Figure 7.2: Movement of agent. Green arrows indicate valid moves and the red arrows indicate an invalid move. a: The agent follows a direct path and fails. b: A new direct path is calculated. An action fails five actions. c: The agent realizes it has to walk a detour. But it fails as well. d: The final path is followed and the agent reaches the goal.

The agent found the shortest path according to its knowledge of the environment in all four cases. The agent would have realized that the first and second path weren't good after one step if the agent calculated a new path every cycle. Recalculating a new path in this scenario doesn't take much time because the agent has no attachments and the map is small.

7.1.2 The Rotate Action

In this test, the agent's goal is to be at the goal cell with one attachment. The agent has been initialized with the belief that a `Clear` action takes twenty steps to complete to avoid `Clear` actions.

The first path calculated takes the agent to the dispenser, where the agent gets an attachment and can be seen on fig. 7.3.a. After getting an attachment a new path is calculated. The second path is a direct path to the goal cell because the agent doesn't know the obstacle in the middle of the environment. This path is shown on fig. 7.3.b The path fails after five actions and a new path shown at fig. 7.3.c is found. The agent doesn't follow this path but is following the path shown on fig. 7.4.a. The actions send to the server is delayed one cycle. This means that the action will be performed one cycle later. The agent will keep on requesting an action until the action completes because of the implementation of the algorithm. This will result in several extra unintended actions. The agent calculates four more paths before crossing the goal cell because of the unintended actions. These paths are compressed into one path shown at fig. 7.4.b. This path includes two `Rotate` actions indicated by the orange arrows. The agent is not located at the goal cell after performing the sequence of actions because of the delay. The cause of the delay is uncertain. A reason could be that the agent is sending a request just before the server goes on to the next simulation step. The request is received in the next step. This problem can be solved by not performing an action in a cycle where a path is calculated. Now the agent will follow the path shown at fig. 7.3.c and the following paths are shown at fig. 7.5.

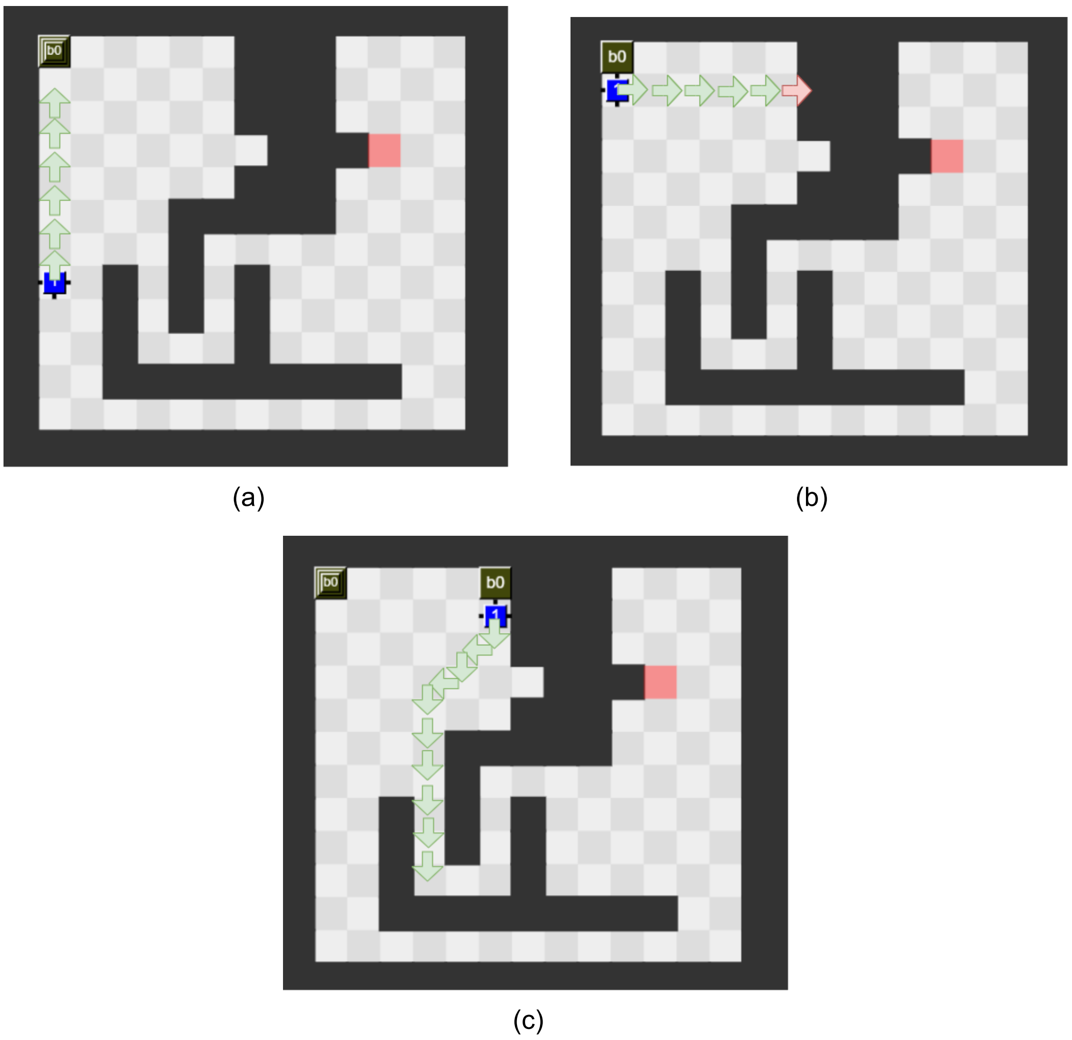
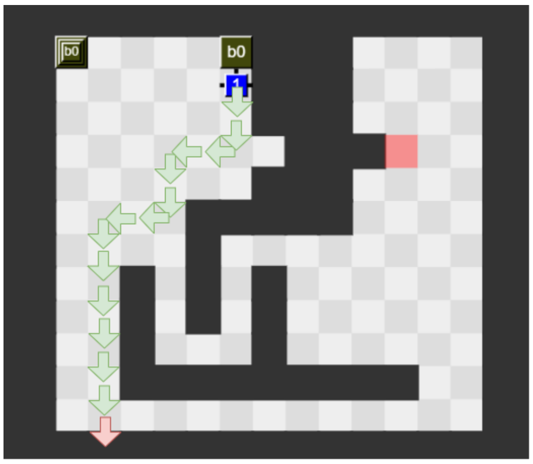
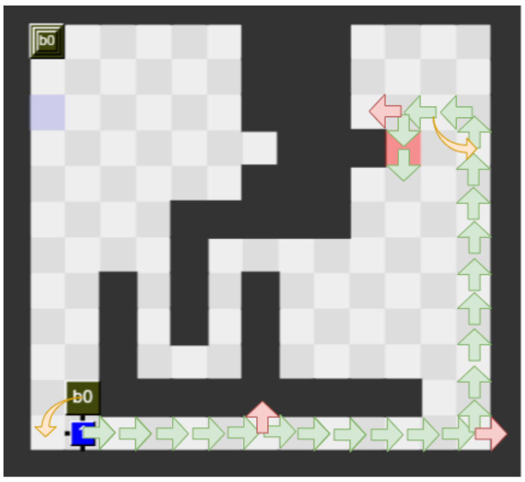


Figure 7.3: Movement of an agent with attachment. a: The agent follows the path and reaches the dispenser. b: The agent has an attachment and follows the path until a `Move` action fails. c: The agent has calculated a new path. This path is not followed.

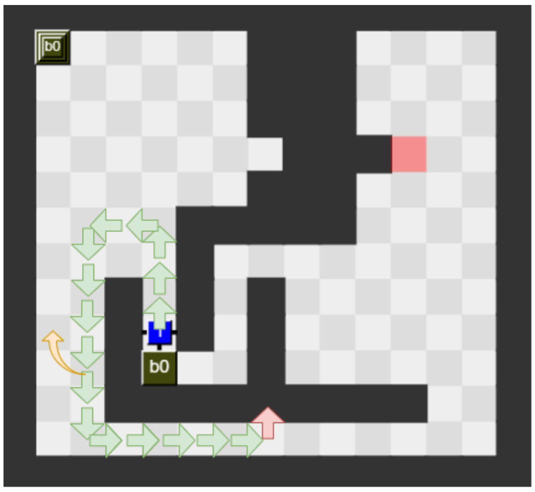


(a)

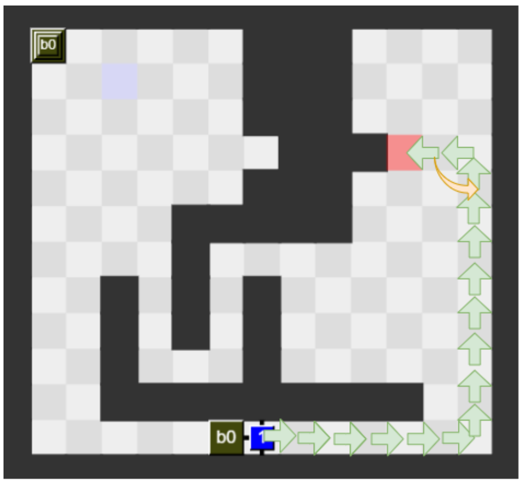


(b)

Figure 7.4: Movement of an agent with attachment. a: This is the path the agent follows instead of the path at fig. 7.3.c. b: The rest of the agent's movement before terminating. Rotations are a part of this movement and are indicated by the orange arrows. Green arrows indicate valid moves and red arrows indicate invalid moves.



(a)



(b)

Figure 7.5: Movement of an agent when the agent doesn't perform an action in the same cycle that a path is calculated. a: The position of the agent after following the path at fig. 7.3.c and the path calculated by the agent. d: The final path calculated. The agent is located at the goal cell after following the path.

7.1.3 The Clear Action

In this test, the agent's goal is to be at the goal cell with one attachment. The agent has been initialized with the belief that a `Clear` action takes five steps to complete, such that `Clear` actions will be preferred. The agent moves identical to the paths shown at fig. 7.3.a and fig. 7.3.b. The next path includes a `Clear` action and goes through the obstacles. The `Clear` action is performed such that only one area needs to be cleared. The cleared cells are pink and the location of the agent when performing the `Clear` action is marked with a "C" on fig. 7.6.

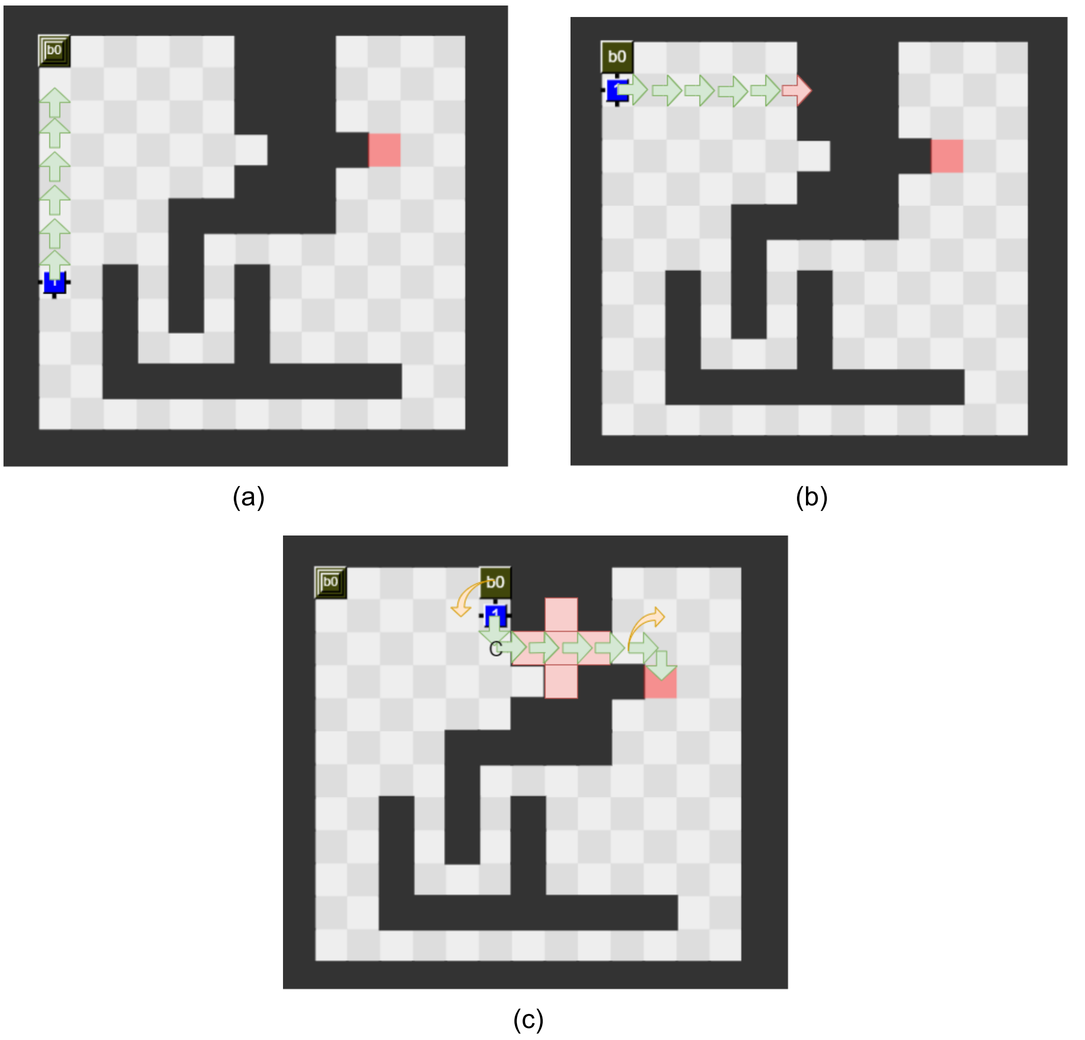


Figure 7.6: Movement of an agent with an attachment and allowed to use `Clear` action. a: The agent follows the path and reaches the dispenser. b: The agent has an attachment and follows the path until a `Move` action fails. c: The agent has calculated a new path. This path requires the agent to clear three obstacles. The cleared cells are pink and the location of the agent when performing the `Clear` action is marked with a ”C”.

7.2 Communication and Linear Temporal Logic

In section 6.2 an implementation for better communication was explained. It’s hard to test several parts of the communication protocols because concurrent agents are involved in the solution. GOAL supports linear temporal logic that is used to validate safety and liveness properties of concurrent programs. Safety properties are used to validate that the program does nothing wrong, and liveness properties are used to validate that the program does something good. All tests in this section are attached in appendix A.

7.2.1 Linear Temporal Logic in GOAL

GOAL can test a specific module by validating state queries before entering and after leaving the module. GOAL can also test a specific module by using invariants that are created by linear temporal logic. These can only be used to validate the program while being inside the module.

GOAL has four different operators for linear temporal logic:

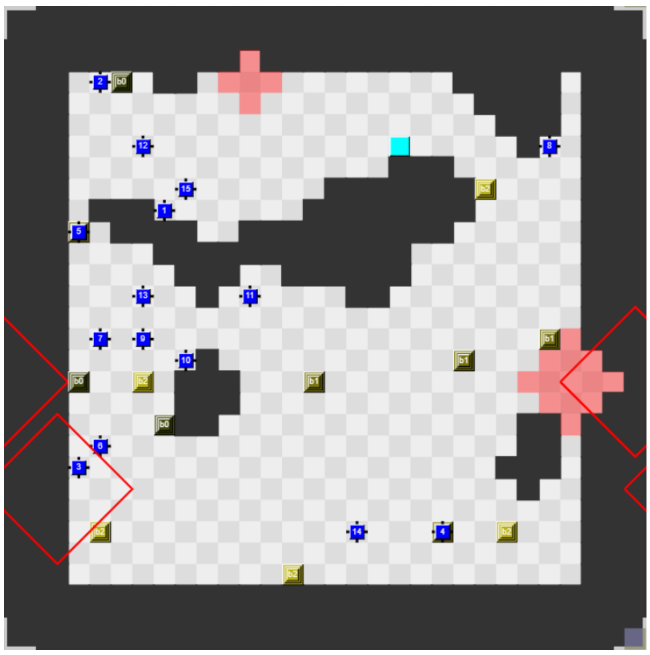


Figure 7.7: Test environment for the communication protocols.

1. `never SC`: The state condition, SC, must never occur.
2. `always SC`: The state condition, SC, must always hold in any state.
3. `eventually SC`: The state condition, SC, should hold at least once before the agent terminates.
4. `SC1 leadsto SC2`: If state condition, SC1, occurs then state condition, SC2, will eventually follow before the agent terminates.

These operators are used in invariants that test the program for safety and liveness properties.

The test module decides on its own when the second state condition of the `leadsto` operator is tested. This means that it can be difficult to test if a belief was present in the belief base for a small interval. An example of such belief is the `shareInfo` belief, which is only present until the end of the event module. Some state conditions may not be tested as often if many tests are active compared to only one test is active. The tests in the following sections are tested one at a time to have full focus on one test.

For more information about linear temporal logic in GOAL see Hindriks 2018, Chapter 10.

7.2.2 Test Environment

A map of 30x30 cells with 15 agents is used to test the communication protocols. The environment can be seen on fig. 7.7. A `Clear` event has a 15% chance to happen during the test simulation. The high chance will make the agents update the statuses of obstacles frequently.

7.2.3 Perceiving Obstacles

New information is inserted in the belief base when the agent perceives obstacles. The status and timestamp of obstacles are updated in `PerceptsToRemember`. `shareInfo` beliefs are inserted if new information is perceived. Recall that `PerceptsToRemember` is a module presented at fig. 6.1.

The obstacles compound was changed in the implementation because the obstacle must hold extra information about its status and timestamp. The obstacle compound is still defined by its first two arguments, which are the coordinates. This means that the agent can have two beliefs of the same obstacle, but where the beliefs differ on the status or timestamp. `PerceptsToRemember` must make sure that the agent can't have multiple beliefs about an obstacle. Using percepts in invariants caused problems. A new belief, `perceivedObstacle`, is used to copy the percepts about obstacles from the environment.

The following safety properties are tested to ensure `PerceptsToRemember` doesn't do anything wrong:

1. Visible cells that have contained or contain an obstacle must be unique by their location at any time. The invariant is shown at fig. A.1
2. If the agent has a `shareInfo` belief after leaving the module, then it must correspond to a percept. The invariant is shown at fig. A.2.

The following liveness property is tested to ensure that `PerceptsToRemember` is doing something good:

1. After leaving the module all perceivable cells that contain or has contained an obstacle must have the timestamp of the current step. If an obstacle is perceived then it will have the status 'present' otherwise 'cleared'. The invariant is shown at fig. A.3.

7.2.4 Receiving Messages

New information is inserted in the belief base when the agent receives messages. The agent receives messages in `MessageReceive`. This module has the purpose to handle information in messages that are sent in a previous cycle. If a message is about the environment then the information is filtered and inserted in the belief base. The messages can also be a part of the connection protocol showed at fig. 5.5. Such messages are inserted unfiltered into the belief base. The connection protocol is entered from this module. This module has a big rotation of beliefs about obstacles. This caused the first versions of the tests that used `leadsto` to test state conditions after the condition had been present in a state. The tests became slow and weren't fast enough to check state conditions. To counter this a test predicate, `obstacleBeforeMessageReceive`, is used to keep track of the old obstacles. Now old and new obstacles can be compared after the module has been left. The tests after leaving the module need a predicate to go through received messages. `receivedMsg` is created for this purpose.

The following properties are focused on obstacles because the implementation in section 6.2 was about a dynamic environment.

The following safety properties are tested to ensure `MessageReceive` doesn't do anything wrong:

1. Only one belief about an obstacle must be present in the belief base at any time. The invariant is shown at fig. A.4.
2. Contradicting information about obstacles with the same timestamp must never occur at any time. The invariant is shown at fig. A.5.

Both the invariant in fig. A.4 and the invariant in fig. A.1 are made because they test different modules. Only the invariant in fig. A.4 is needed if the two modules are tested at the same time.

The following liveness properties are tested to ensure that `MessageReceive` is doing something good:

1. An agent should update its knowledge about an obstacle in the module when more updated information arrives. The invariant is shown at fig. A.6.
2. An agent must share information with connected agents if the agent has contradicting and more updated information. The invariant is shown at fig. A.7.

7.2.5 Sending Messages

The agent is sending all its messages from `MessageSend`. `shareInfo` beliefs are used in other modules to remember what information the agent should send and to which channel. This is further explained in section 6.2.2. A `msgSend` belief is inserted when the agent has send the information found in a `shareInfo` belief. `msgSend` beliefs are only used for testing.

The following safety properties are tested to ensure `MessageSend` doesn't do anything wrong:

1. The agent must not have any `shareInfo` beliefs after leaving the module. The invariant is shown in fig. A.8.

The following liveness properties are tested to ensure that `MessageSend` is doing something good:

1. All messages must be sent to their corresponding channel with the sender's name and the current simulation step. The invariant is shown in fig. A.9.
2. A message is sent to all connected agents when team is the channel. The invariant is shown in fig. A.10.

7.2.6 The Connection Protocol: Sending Percepts

The connection protocol is split into three modules one for sending percepts, one for validating percepts, and one for receiving the information. These properties are regarding the module that sends percepts, `AgentShareSend`.

The agent will only enter this module if at least one agent from the same team is perceived this cycle. The purpose of this module is to find percepts shared with the perceived agents. The found percepts are remembered until the next cycle.

The following safety property is tested to ensure `AgentShareSend` doesn't do anything wrong:

1. The agent must have perceived another agent from the same team to enter this module. The invariant is shown in fig. A.11.

The following liveness property is tested to ensure that `AgentShareSend` is doing something good:

1. For every `shareInfo` belief a corresponding entity and shared percepts, `commonEnvironmentPercepts`, must be present after leaving the module. The invariant is shown in fig. A.12.

7.2.7 The Connection Protocol: Validating Percepts

These properties are regarding the module that validates percepts, `AgentShareReceive`.

This module should only be entered if the agent has received percepts from another agent. If the agent receives some percepts that are validated then the agent will be connected to the sender of the percepts and know where the sender's origin is. The percepts send between agents are not unique; messages with identical percepts are filtered. Beliefs about the percepts that the agent has to send and received are deleted in this module because they are outdated after one cycle.

The following safety properties are tested to ensure `AgentShareReceive` doesn't do anything wrong:

1. The agent should only enter the module if the agent has received another agent's percepts, `agentShare`. The invariant is shown in fig. A.13.
2. The agent must not have beliefs about percepts that the agent has send, `commonEnvironmentPercepts`, and received, `agentShare`, after leaving the module. The invariant is shown in fig. A.14.

The following liveness property is tested to ensure that `AgentShareReceive` is doing something good:

1. An agent will connect to another agent and receive information from the agent if the percepts received is validated successfully. The invariant is shown in fig. A.15.
2. Duplicates of `agentShare` beliefs must never be flagged as unique. The invariant is shown in fig. A.16.

7.2.8 The Connection Protocol: Receiving Information

These properties are regarding the module, `insertInformation`, that handles the received information.

This module should only be entered if the agent has received information from another agent. The agent should have beliefs about all the environment received when leaving the module. If the agent received information from an unconnected sender then the agent will be connected to the sender in this module. The received information might be outdated after one cycle, thus the agent deletes all beliefs about received information before leaving the module.

The following safety properties are tested to ensure `insertInformation` doesn't do anything wrong:

1. The agent should only enter this module if it has received information from another agent. The invariant is shown in fig. A.17.
2. The agent must not have any beliefs about received information, `receivedInformation` after leaving the module. The invariant is shown in fig. A.18.
3. Only one belief about an obstacle must be present in the belief base at any time. The invariant is shown in fig. A.4.

Beliefs about obstacles are also inserted in the belief base in this module, so the invariant shown in fig. A.4 must also hold in this module.

The following liveness properties are tested to ensure that `insertInformation` is doing what it's supposed to do:

1. The agent will know all the other agent's information after connecting. The invariant is shown in fig. A.19.
2. The agent will be connected to another agent if the agent has received information from the other agent. The invariant is shown in fig. A.20.

7.3 Eventually Agents are Prepared

This section will be about testing the implementation made in section 6.3. The tests are similar to the tests in section 7.2. These tests are also based on LTL formulas and the same environment is used. The tests in this section can be split into two parts. One part focusing on the master-agent and another part focusing on the remaining agents. All tests in this section are attached in appendix B.

7.3.1 Master-agent

The master-agent enters `allocateTypes` if it has received any requests. The master-agent shouldn't enter if no request is received. After handling a request it must be deleted since it shouldn't be handled anymore. This can be seen as the safety properties of the master-agent's part of the preparation.

1. The agent must have a belief about a request if `allocateTypes` is entered. The invariant is shown at fig. B.1.
2. The agent must not have any beliefs about requests when leaving `allocateTypes`. The invariant is shown at fig. B.2.

When the master-agent has entered the module it must find a distribution for every request received. Furthermore, the changes made to types must be sent to every agent included in the request. This can be seen as the liveness properties that ensure that the master-agent does something good.

1. A distribution must be found for every request. The invariant is shown at fig. B.3.
2. Changes to types must be sent to every agent included in the request. The invariant is shown at fig. B.4.

7.3.2 Remaining Agents

The remaining agents are tested differently. Different from the master-agent a belief, `agentType`, with their name and type exist. An agent must only have one `agentType` belief for each agent. This is the only safety property for the remaining agents.

1. Any agent must never have more than one type. The invariant is shown at fig. B.5.

The team leader will request the master-agent for new types when the existing types are out of balance. This will eventually result in the types being in balance. Furthermore, an agent must adopt a corresponding goal when it's told to change type by the master-agent. Two beliefs, `goalSubmitAgent` and `goalBlockAgent`, are used to test the goals because problems occurred with state queries to the goal base. Three liveness properties are created from these requirements.

1. The team leader will request the master-agent for new types when they are out of balance. The invariant is shown at fig. B.6.
2. Eventually the types will be in balance. The invariant is shown at fig. B.7.
3. An agent should adopt a corresponding goal if it evolves into a block-agent or submit-agent. The invariant is shown at fig. B.8.

7.4 GOAL-DTU vs. GOAL-DTU with A* & Improved Communication

In this section, thirty simulations will be performed. The simulations will be identical to the qualification of the MAPC 2020. The MAS created by GOAL-DTU will be used as the baseline in the first ten simulations. GOAL-DTU's MAS with A* (A) is used for the next ten simulations. The remaining simulations will be performed by GOAL-DTU's MAS with the communication protocols (CP). Agents will not stall after computing A*; it will cause too many simulations steps with `No_Action`. To qualify for the MAPC a maximum of 30% of the simulation steps must include `No_Action`.

The map of the first fifteen simulations has the dimensions of 70x70 cells. It has 5 to 10 dispensers of 3 different types, 3 task boards, and 3 goal zones. A team is composed of 15 agents. The simulations have 300 simulation steps and the time interval for a simulation step is 4 seconds.

Qualification Simulation 1				
MAS	Sim. No.	Points	First Submission	Steps with No Actions
GOAL-DTU	1	9	241	0
	2	13	201	0
	3	9	253	0
	4	9	243	0
	5	9	257	0
	avg.	9.8	239.0	0.0
A* (A)	1	13	246	4
	2	11	192	0
	3	13	246	4
	4	13	246	5
	5	13	246	5
	avg.	12.6	235.0	3.6
Communication Protocols (CP)	1	0	-	117
	2	0	-	47
	3	0	-	67
	4	0	-	89
	5	0	-	155
	avg.	0	-	95

Table 7.1: Results for Qualification Simulation 1

The results for Qualification Simulation 1 are seen on table 7.1. GOAL-DTU had an average score of 9.8, A had an average score of 12.6, and CP had an average score of 0. Both GOAL-DTU and A achieved the highest score of 13 points, but A was more consistent. GOAL-DTU's first submission came on average in simulation step 239. This is 4 simulation steps later than A. The minimum score of A was 11 points while it was 9 points for GOAL-DTU. GOAL-DTU didn't have any cycles with `No_Action`. A had on average 3.6 simulation steps with `No_Action` while CP had 95 on average. The high number of simulation steps with `No_Action` for CP was caused by the master-agent creating task-plans.

The second map is 60x100 cells. It has 5 to 10 dispensers of 3 different types, 5 task boards, and 4 goal zones. A team is composed of 50 agents. The simulations have 300 simulation steps and the time interval for a simulation step is 4 seconds.

The result of Qualification Simulation 2 can be seen on table 7.2. The results show that A didn't score any points. The high number of simulation steps with `No_Action` for A is eye-catching. This was caused by one or more agents crashing in simulation 1,2 and 4, due to the computation of A*. The state query used too much time and GOAL threw an exception. In simulation 3 for GOAL-DTU, three submit-agents tried to assemble a pattern

Qualification Simulation 2				
MAS	Sim. No.	Points	First Submission	Steps with No Actions
GOAL-DTU	1	0	-	9
	2	12	108	2
	3	0	-	54
	4	4	184	8
	5	4	108	13
	avg.	4.0	133.3	17.2
A* (A)	1	0	-	189
	2	0	-	189
	3	0	-	2
	4	0	-	176
	5	0	-	38
	avg.	0	-	118.8
Communication Protocols (CP)	1	0	-	3
	2	0	-	1
	3	4	85	8
	4	0	-	3
	5	0	-	2
	avg.	0.8	85	3.4

Table 7.2: Results for Qualification Simulation 2

in the same goal zone. This resulted in eight agents in a big cluster trying to connect to the wrong agents. They tried to connect every cycle until the deadlines of the tasks. CP didn't have the same problem as in Qualification Simulation 2. This was caused by agents couldn't find each other and connect.

8 Discussion

In this chapter, the tests and design choices of the implementations will be discussed.

8.1 Missing Actions and Speed

An improvement to make the agents move better was implemented in section 6.1 and tested in section 7.1.

8.1.1 Rotation Constrain

The agent uses 120-130 seconds on calculating the path at fig. 7.3.c. This is slow and can be caused by the agent not wanting to move away from the goal cell. Recall that the heuristic value is getting worse when moving away from the goal cell. This results in the agent preferring to rotate over than moving away from the goal cell. A constrain has to be made about the rotation of an agent. The agent can rotate clockwise and anticlockwise. This means rotating more than twice at the same location is unnecessary. The agent should be constrained to do two rotations while moving in an area with no obstacles in one of the eight surrounding cells. When an obstacle is located in one of the eight surrounding cells the constrain is lifted. The agent will be constrained again when the eight surrounding cells is clear. However, the constrain only reduced the computation time by 5%. The constrain must be changed to one rotation. This reduced the computation time by 80%. With this constrain the agent will have a problem when two rotations are needed to continue on a path. When this happens the agent must find an obstacle in between the two rotations.

8.1.2 Calculate A* for Every Action

In section 5.1.3 it's assumed that computing A* will use a lot of time. But the agent would have performed better by performing an A* for every step in all three tests in section 7.1.

Why must the agent fail an action before calculating a new path? The test environment is a small map with edges and only one agent is present. The agent doesn't compete with other agents to get CPU time. A map with edges ensures that the state space is finite. In a normal simulation, the map will be a torus and the state space infinite. The paths can in theory be infinitely long. In practice, they are not. A path can cross the entire map multiple times before ending at the target location. In a normal simulation, more than one agent will be present. All agents may calculate an A* concurrently. This will probably be slower than four seconds. Four seconds was used as the time interval for the simulation steps in the MAPC qualification. Imagine all agents calculating A* every cycle to figure, which action should be performed next. After some time an agent will have found an action and performed it. Then it starts over to calculate the next action. The waiting time between starting and ending the A* must not exceed the time interval of a simulation step. If this happens no agent will use less than four seconds in the majority of the simulation steps and give CPU time to other agents. Most simulation steps will be used to calculate new paths. This results in a minimal number of actions performed by the entire team. If A* was calculated once and followed until it fails, then all agents will perform a sequence of actions before calculating a new path. Agents use less than four seconds in a cycle when following a path. Thus the majority of the simulation steps will result in actions. This means that extra CPU time is available for the agents who need it. Furthermore, their paths may not fail at the same time, so calculations of new paths may happen in different simulation steps.

8.1.3 Delay of Actions

A problem occurred when testing the `Rotate` action. Actions got delayed with one cycle. The cause is unknown but was solved by making the agent stall after computing an A*. This only happened when A* wasn't computed every cycle. When calculating A* once the agent will request an action continuously until it succeeds. This error has been seen before by GOAL-DTU in the MAPC qualification where they also solved the problem by stalling.

8.1.4 Missing Actions

Only three actions are present in the A*. The purpose of the A* was to make the agents move better. The three actions included are `Clear`, `Move` and `Rotate`. These are the most relevant actions for movement. `Skip` is relevant as well. Sometimes an agent is better off to skip. The motive to skip in the current A* can be to restore its energy, such that a `Clear` action can be performed. This assumes that the `Clear` action is included in the A*. So the `Skip` action is less relevant than the `Clear` action. The remaining actions were more relevant for completing tasks. It could be great if the A* included all actions, but the algorithm had to be constrained in this thesis because of time.

8.1.5 Avoiding Other Agents

The A* doesn't take the movement of other agents into account when calculating a path. The agents may move every round by following a path or not. Imagine an agent is aware of other agents' paths. When calculating its path, then it will try to calculate where other agents are in a given simulation step. This will make it possible for the agent to avoid them. But the agent can't be sure where other agents are at a given time. The agent doesn't know when it's done calculating its path. From this, it follows that the agent doesn't know in what simulation step the first action of the path will be performed. This means that the agent may be delayed compared to the paths of other agents. Furthermore, other agents may calculate a new path due to a failed action. Such that an avoided agent may not even reach a location that crosses the agent's path. That's why the movement of other agents are not taken into account when calculating A*.

8.2 Sending Too Many Messages

An improvement to make the agents share information was implemented in section 6.2 and tested in section 7.2.

8.2.1 Dynamic Environment

Agents do now send information to connected agents when they acquire new information about the environment. This minimized the messages sent as much as possible without connected agents coordinate with each other. Duplicated messages are sent if two connected agents perceive the same change of environment in the same cycle. Coordination of connected agents can remove these duplicated messages. The coordination could be that the agent who has the lowest number in its name sends the message. But the coordination may just add extra complexity to the code without adding much to the MAS.

8.2.2 Sending Identical Messages to Multiple Agents

An agent has to send an identical message to every agent it's connected to when it wants to share information. Instead of sending the same message multiple times, communication channels can be used. The agent will then send the message once. The receivers will receive the message as before. But they need to subscribe to the same communication channel. The communication channels can be implemented if all agents in a network subscribes to the same channel. The channel can be called the same as the lowest number found in the name of a connected agent. When agents connect to new agents they have to reevaluate the communication channel and unsubscribe from old channels.

8.2.3 The Connection Protocol

Duplicated messages occur when two agents, A and B, connect and send newly received information to their connected agents. This happens because A and B are now connected to the same network of agents.

To avoid duplicated messages A and B could send new information to agents that were connected to them before the connection protocol. But then a new problem arises when A connects to B and another agent, C, in the same cycle. C is not connected to A or B. A receives information from B and C and forwards it to its connected agents before the connection protocol. Neither of B and C is included in this group of agents. This means that B and C won't get each other's information indirectly through A. But both B and C will be connected to A. This destroys the idea of merging networks consistently until all agents are connected. The trade-off is now whether the connected agents should receive information twice or to constrain the agent to one new connected network per cycle.

More information should be received as fast as possible to enlighten the agent. The agent will receive more information for every connected agent. A constraint of one new network connection will slow the progress to create bigger networks. It will leave the agents with less information for more time. The constrain will result in fewer messages, and less time in the connection protocol. But it will increase the speed of cycles but not much.

The constrain can be the better choice if the cycles are too slow due to heavy calculations like A*. The duplicated messages are only present when agents complete the connection protocol. The number of times the connection protocol can be completed is constrained to the number of agents on a team. This is a small number compared to the number of simulation steps in the simulations used for the MAPC 2020 qualification.

8.2.4 LTL Formulas

The test presented in section 7.2 were made to ensure that the agents do something good instead of doing something wrong. More liveness properties could have been tested. It was difficult to find the correct LTL formulas due to problems with the `leadsto` operator and state queries to the percept base.

Big tests shouldn't be performed concurrently if one of them is using the `leadsto` operator might be a problem. One-cycle beliefs like `receivedInformation`, `shareInfo` and `agentShare` will be gone after one cycle and beliefs about obstacles can change in any cycle. The problem could be caused by multiple big tests are stealing CPU time from each other such that every state may not be checked for a state condition. One of the states not being checked might be the unique state that satisfies a state condition. If a test misses that unique state, it will continue to search for the state until the agent terminates and the test fails.

All the formulas succeeded when performed one at a time. If they succeed on their own it doesn't matter if they don't succeed when performing concurrently with other formulas. The tests don't add anything to the agent's belief base. They can't affect each other indirectly through the agent.

8.3 Centralized Preparation

The preparation is a centralized implementation because most of the computations are done by the master-agent. The master-agent is doing the computations for every team, without being on any of them. This ensures that unnecessary computations of type allocations don't happen. Furthermore, the algorithm has another centralized node, the team leader.

8.3.1 Why Two Centralized Nodes?

The team leader is used to decrease the number of requests sent to the master agent. Without the team leader, every agent on a team will send an identical request to the master-agent. Without the team leader and master-agent, they should perform the type allocation by themselves. These allocations of types might give different results and the team will not be coordinated correctly. Thus at least one of the master-agent or team leader is needed.

Why can't the team leader do the allocation of types? If this was the case problems will occur when two agents from different teams connect by following the connection protocol shown at fig. 5.5. One of the agents is a team leader and the other agent is not. The types of both teams are out of balance before the connection protocol starts. Imagine that the team leader is on team B and the other agent is on team A. If they connect synchronously the team leader will know if it's the team leader of the merged team. The team leader will also know the types of the agents on A in the previous cycle. It's not certain that these types are the most updated version. Three problems will occur.

First Problem The team leader of B is selected as the team leader of the merged team, and in the same cycle, the agents of A have gotten new types. The team leader is not aware of these types. In the next step, the team leader will allocate new types based on old information about types of A. This results in agents of A get new types that may not correspond to the agent type they received in the previous cycle. This is not optimal because agents of A are wasting one or more cycles on trying to calculate a path from A* and achieve an unnecessary goal.

Second Problem The team leader of B is selected as the team leader of the merged team, and the agents of A haven't got new types. The other team leader of A will first get knowledge about the connection between the two agents in the next cycle. Both team leaders know that the types are out of balance so the types must be changed. Both team leaders calculate new types and send the result to their team. Only one agent should calculate the new types to minimize unnecessary computations.

Third Problem The third problem is a continuation of the second problem. Agents of A who haven't got knowledge about the connection yet will receive three messages. One message about the connection and two messages about new types. If the message about the connection is read last then the types received from the team leader of A will be inserted. The message from the team leader of the merged team will be discarded. This will leave some of the agents with the wrong knowledge. The team leader won't calculate new types again because it believes that the types are in balance.

By using the master-agent as a central agent the first problem will never happen. The master-agent always has the most updated version of an agent's type. The allocation algorithm will never start with outdated types. But two extra cycles will be used when using the master-agent. Agents are moving around unaware of what to do next in the two cycles, but they will only compute A* once. Agents use one more cycle on doing nothing but compute A* one time less. It's a trade-off. On big maps, multiple computations of A* will take more time than a simulation step due to longer distances. Especially if multiple agents are computing it at the same time. Then it might be better to use one cycle more and calculate A* once. On smaller maps, the computation of A* is faster and agents may be done with the computations before the simulation step ends. But the agents also need to stall for one cycle after computing an A* due to the problem in section 8.1.3.

The second problem contradicts the idea of only one agent should calculate the new types. This is solved by using the master-agent. The computation is especially heavy when initializing a team. It may happen that both team leaders will initialize a team at the same time. This happens if the team leader of A perceives its first task board such that it's ready to team up. Meanwhile, the team leader of the merged team connects to the last agents needed to be ready to team up.

The third problem can be solved easily by handling new information received from the connection protocol before handling the remaining messages. But it will result in an extra match statement on the mailbox. Only necessary messages are sent because of the implementation of the communication protocols but the mail box may still have a lot of messages. The computation time on matching a message in the mailbox is fast so the third problem shouldn't weight that much in the decision whether to use master-agent or not.

The solution with the master-agent is chosen because of the following:

- The map used for the MAPC 2020 is a torus so it can be infinitely big. The wait time for A* may delay the agents with more than one cycle on this kind of map.
- The agents have to stall a cycle every time they compute A*.
- Initializing a team uses $\theta(!n)$. n is often below 10 but it may increase to larger numbers, so unnecessary calculations of new types should be avoided.

8.3.2 Initializing the team

Initializing the team scales factorial with the number of explore-agents. How a team is initialized is described in section 6.3.3. The reason is that all possible distributions must be taken into account. The number of possible distributions can be described by the binomial coefficient.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Where n is the number of explore-agents and k is the number of agents needed for the smallest team.

This will be a problem in maps with many agents, many dispensers, and only one task board. A scenario can happen where all agents but the master-agent are connected. They only need to find a task board then they are ready to team up. They finally find it and they have to go through an algorithm that scales factorially with the number of connected explore-agents. This will result in many simulation steps with `No_Action`.

The qualification rounds for the MAPC 2020 included a simulation with fifty agents and three different types of dispensers. If all types of dispensers are perceived and all agents but the master-agent are connected then the binomial coefficient is:

$$\binom{50-1}{3+1} = 211.876$$

That is a lot of possible distributions. A solution could be similar to GOAL-DTU's solution about active and inactive agents, where only the active agents may connect. With fifteen active agents the binomial coefficient is:

$$\binom{15-1}{3+1} = 1.001$$

That is an improvement of factor 211. A better algorithm should be found in the future so the initialization of a team is faster. An idea is to take a similar approach taken in Dijkstra's Algorithm. Expand the smallest edge until a solution is found. Dijkstra's Algorithm is explained in details in Cormen et al. 2009, Chapter 24.

8.3.3 Distances

The preparation implementation allocates types to agents. Agents prepare themselves by moving to the nearest dispenser or taskboard of their location. Moving to the nearest dispenser or taskboard is not optimal. Agents can be very far from each other when they have to complete a task together if they prepare like this. Instead, the team leader should calculate the average location of the agents. Find the nearest goal cell of the average location. The nearest dispensers of each type and nearest task board of this goal cell should be used when calculating the distances that are used in the allocation algorithm. This will result in agents on the same team being near each other when starting a new task.

The next decision has to be taken when two teams merge. Should a new average location be found where all agents move towards? Should one of the two teams move towards the other team such that at least one team can complete tasks? This won't be discussed any further.

Euclidean distance is used to calculate distances in the current version of the allocation algorithm. This is a fast and simple approximate. To get a better approximate A* could have been used to calculate how many cycles are needed for all agents to be ready. A* will only be possible if the master-agent is connected to the team leader or if all the obstacles known by the team leader are attached to the request. Calculating all distances by A* will use too much time for the master-agent. If a team is to be initialized with five explore-agents and three different types of dispensers then the master-agent has to compute A* twenty times. This is too much for a single agent to compute. And that's why euclidean distance is used for the distances in the allocation algorithm.

8.3.4 Future Strategies

The preparation is meant to be a foundation for future strategies. These are some of the strategies that can be used on top of the foundation.

Mining the Dispensers In this strategy block-agents continuously move blocks from dispensers to a specific goalzone agreed by the team. A submit-agent should wait near a task board until it finds a task the team can complete with the blocks near the goalzone. Then it will move to the goalzone and ask some of the block-agents to help. This strategy is exposed to `Clear` events; all the blocks collected around the goalzone can be cleared by one event. By changing the preparation goal of block-agents and adding few extra lines in the `aStar` module they will act like this. Under the assumption that all agents on the team are located around the same goalzone as discussed in section 8.3.3.

Simple Strategy In a more simple strategy, the block-agents will move away from the dispenser when the submit-agent tells them to do so. A submit-agent waits near a task board until a task is found that it can complete with the dispensers next to the block-agents. The submit-agent and the block-agents will move towards the same goal cell and complete the tasks. The block-agents will bring a block for the submit-agent.

Blocking A more defensive strategy is to let the majority of block-agents block dispensers throughout the simulation. A block-agent can block a dispenser by having a block on top of it or standing on top of it. They should only move when a member of the team wants a block from it. The better idea is to stand on top of it because `Clear` events won't remove agents but they will remove blocks and disable agents for multiple simulation steps. A disabled agent on top of a dispenser is still blocking. The minority of agents form a small team and follow *Simple Strategy*. This will only work if a team consists of more agents than the number of dispensers in the simulation. Only the goal of the block-agents has to change to make the block-agents block the dispensers by standing on top of them. The predicate `typesInBalance` can be changed such that a team only has one submit-agent, and the remaining agents are block-agents. Extra code has to be written to make the block-agents move when agents of the same team want a block.

8.4 Are They Improvements?

In section 7.4 the communication protocols and A* were tested on GOAL-DTU's MAS.

GOAL-DTU With Improved Communication The communication protocols didn't do anything good for GOAL-DTU's MAS. The implementation gave the MAS an average of 95.0 simulation steps with `No_Action` in Qualification Simulation 1 and scored 0 points. Qualification Simulation 2 went better. The MAS had an average of 3.4 simulation steps with `No-Action` and scored a total of 4 points in the five simulations.

The simulations of Qualification Simulation 1 were profiled such that it was possible to see how much time was used in each module. It showed that the master-agent used a lot of time on creating taskplans. The reason was that agents shared to much information. All of them could be submit-agents and they were all connected. Recall the strategy used by GOAL-DTU's MAS from section 4.2. This will exhaust the master-agent in cycles where tasks are available but none of them are possible to complete for the agents. The communication protocols won't work with the current algorithm for creating taskplans. The master-agent will delay the remaining agents and the team won't achieve any points. A solution could be to introduce dynamic agent types. The dynamic allocation of types will ensure that a minority of agents will request the master-agent for a taskplan. If the master-agent is still exhausted by all agents being connected then the submit-agents must decide on a task first and request a corresponding taskplan from the master-agent. The reason for the decrease in simulations steps with `No_Action` was that inactive agents can't connect to other agents. The map was large and with only fifteen active agents it was hard for them to find each other without help from inactive agents. They can't connect if they can't find each other. This means that the connection protocol couldn't scale and create a network of agents.

GOAL-DTU With A* The results for Qualification Simulation 1 showed that A* improved the MAS of GOAL-DTU by 2.8 points on average. The average score with A* was 12.6 points while the average score for GOAL-DTU was 9.8. Furthermore, A* was consistent in its performance. It scored 13 points in 4 out of 5 simulations and got 11 points in the last simulation. On the other hand, A* had on average 3.6 simulation steps with `No_Action`. 3.6 simulation steps correspond to 1.2% of the simulations 300 simulations steps. This is far below the 30% hard cap for the qualification. The tasks that were completed in the four simulations were the same. The agents that completed the tasks were different. This indicates that the four simulations were not identical. It was chosen that agent's shouldn't stall after computing A* because of the `No_Action` hard cap. The problem regarding performing actions twice did occur multiple times throughout the simulation but it didn't prevent the agent's from scoring more points than GOAL-DTU's MAS.

The results for Qualification Simulation 2 showed that agents may crash due to A*. GOAL-DTU's MAS with A* didn't score a single point and had on average 118.8 simulation steps with `No_Action`. In three of the five simulations, one or more agents crashed due to A* using too much time. The cause of the crashes is unknown, but it might be caused by A* not being able to find a path. A* will continue to expand due to the infinite state space of a torus. To solve this problem a limit for time or step cost can be used to limit A*. When A* hits the limit the path until the last expanded state is returned.

Neither of the MASs with A* or the communication protocols would have qualified for the MAPC 2020 due to not scoring points in both Qualification Simulation 1 and Qualification Simulation 2. GOAL-DTU's MAS did qualify. Are they then improvements? The MAS with A* performed best in Qualification Simulation 1. By adding a limit to A* agents shouldn't crash anymore and A* should be an improvement for the agents' movement. The MAS with the communication protocols had a high average of simulation steps with `No_Action`. But it was caused by agents communicating too well. This means that the communication protocols are an improvement for the communication of the MAS while not being an improvement for the entire MAS. This can be changed by implementing dynamic agent types.

9 Future work

The improvements are not complete. Multiple problems were described in chapter 8. In the future, more attention must be devoted to solve these problems.

Tasks For A* The implementation of A* includes only three actions. To make the agents more effective the remaining actions should be added. In the current version, two paths must be calculated if an agent wants to get in range of a task board, accept a task, and move to a goal cell. This can be changed to one path if `Accept` is implemented. The list for expanded states should be changed into a hash-map this will improve the time complexity from $\Theta(n)$ to $\Theta(1)$ when checking if a state already has been expanded. The most important improvement for A* is to constrain the algorithm with a limit. A limit could be a maximum step cost or a time-limit. It will ensure that no agent will crash due to the search algorithm not find a path.

Tasks For Preparation To complete the implementation of preparation a central location for the team must be implemented. The nearest dispensers and task boards of the central location must be used in the allocation algorithm, and agents must prepare near these dispensers and task boards. It will ensure that agents are near each other when a task is accepted. The allocation algorithm uses $\Theta(n!)$ when initializing a team. n is the number of agents. A better algorithm for initializing a team must be found. Simulations with more than fifteen agents will appear in the future.

Tasks For the Communication Protocols The implementation of the communication protocols is almost complete. To complete it a better integration to GOAL-DTU's MAS must be made. The problem occurs because all agents are connected and can be submit-agents. The implementation of preparation brings dynamic agent types. These will solve the problem. Otherwise a new algorithm for creating the taskplans must be made. Communication channels might be an improvement for the protocols. It will ensure that an agent won't send an identical message multiple times. The connection protocol can be used to find the dimensions of the map. It's possible when receiving and validating percepts from an already connected agent that should be somewhere else. The misplacement of the connected agent is a multiple of the width or height of the map.

10 Conclusion

In this thesis, three implementations of GOAL-DTU's MAS have been made. The focuses of the improvements were movement, communication, and preparation. Agents didn't use their knowledge about the environment to choose an action. A search algorithm, A*, was implemented to make the agents find a path before moving. Agents had problems when connecting asynchronously to each other. This caused agents to miss valuable information about the environment. Agents didn't share much information about the environment before and didn't remember percepts about obstacles. New communication protocols and changes to beliefs about the environment made the agents connect and communicate better. Agents were moving semi-randomly around collecting blocks until they were chosen for a task. Agents were detaching blocks at unlucky places and had to move a long distance to meet up with other agents. The third improvement aimed to solve this by introducing new agent types such that agents can prepare themselves for tasks.

A* has been implemented to solve the problem of the movement of agents. A* made the agents use a heuristic function that uses step cost from the start state and distance to the target state to calculate a heuristic value for each expanded state. Compared to earlier, an agent does now find a path to the target state before moving. The implementation has issues. The computation of A* may crash if a path doesn't exist. The agent is constrained to only doing one rotation in open areas. Only three actions are included in the A*, `Move`, `Rotate` and `Clear`. Every action has been tested in an environment without other agents. `Move` and `Clear` passed their tests without problems. Actions were performed twice when testing `Rotate`. This problem is known from the MAPC 2020 qualification and was solved by stalling the agent for one cycle after computing A*.

New communication protocols were added to solve the problem with communication. The connection protocol solved the problem of agents missing out on valuable information. Agents do now send information after they have connected to another agent. Furthermore, the distance between the origins of the two agents is attached to the information. This ensures that the receiving agent can translate the information and connect to the other agent. The new information is now shared with all the connected agents of the receiver. Agents of GOAL-DTU's MAS didn't share much information about the environment because most of the environment consists of obstacles that may disappear. By adding a timestamp and a status to obstacles it was possible to remember and share this information. The timestamp ensured that agents could compare beliefs if they disagreed on the status of an obstacle. Another protocol is added to minimize the number of messages sent about the environment. The protocol ensured that agents only share new information. This made it possible for agents to share information about the environment without spamming each other with duplicated information. This means that all connected agents have approximately the same beliefs about the environment.

New agent types were implemented to solve the problem of agents not preparing for a task. Agents are now given types such that they can prepare themselves according to their type. Four different types exist. Master-agent, submit-agent, block-agent, and explore-agent. Master-agent is predefined and will never change. The master-agent is a centralized agent used in algorithms that need coordination with unconnected agents. Explore-agents explore until they join teams and evolve into submit-agents and block-agents. Submit-agents prepare themselves by being in a range of a task board. Block-agents will prepare themselves by being adjacent to a dispenser of a specific type and have a block of the same type attached to them. The teams grow dynamically by merging with other teams or by adding new explore-agents. Each team has a team leader that requests the master-agent for new types when the relationship between explore-agents, submit-agents, and block-agents doesn't match a predefined relationship. The implementation has some shortcomings. Agents are indeed preparing for a task but they are moving towards the nearest dispenser or task board from their location. Furthermore, the algorithm for initializing a team uses factorial time.

To keep a good overview, the code has been split into multiple modules, and predicates in the knowledge base have been put in order based on the usage. Each module has a specific purpose and is accessed directly or indirectly from the event module.

Linear temporal logic formulas were used to test the safety and liveness properties of the communication and preparation improvements. All the LTL formulas hold when one was executed at a time.

GOAL-DTU's MAS was used as the baseline in simulations that should test if A* and the communication protocols improved the MAS. In a small environment with few agents, the MAS with A* performed best and the MAS with the communication protocols performed worst. The master-agent of the MAS with communication protocols got exhausted while creating taskplans because the agents communicated too well. In a large environment with many agents, the MAS with A* performed worst, and the baseline performed best. One or more agents crashed in the MAS with A* due to an infinite state space. The MAS with communication protocols never got to use its potential because active agents couldn't find each other.

Overall the improvements all passed their tests showing that they work in test environments. The simulations showed that A* improved GOAL-DTU's MAS in a small environment but adds a risk for agents crashing. The communication protocols did improve communication of GOAL-DTU's MAS in a small environment but worsened the overall score.

This thesis has shown that agents can share information about dynamic environment and connect asynchronously. A* is a suitable solution for the movement of agents but must be constrained by a limit in infinite state spaces. Preparation and explicit agent types are said to be an improvement. But are yet to be tested in the qualification of the MAPC 2020.

Bibliography

Hindriks, Koen V. (2018). *Programming Cognitive Agents in Goal*. GOAL. URL: https://bintray.com/artifact/download/goalhub/GOAL/GOALProgrammingGuide.pdf.

Ahlbrecht, Tobias et al. (2020). "The Multi-Agent Programming Contest: A résumé". In: *CoRR* abs/2006.02739. arXiv: 2006.02739. URL: https://arxiv.org/abs/2006.02739.

Russell, Stuart J. and Peter Norvig (2010). *Artificial Intelligence: A Modern Approach*. Third edition. Pearson Education. ISBN: 9780136042594.

Fredman, Michael L. et al. (1986). "The Pairing Heap: A New Form of Self-Adjusting Heap". In: *Algorithmica* 1.1, pp. 111–129. DOI: 10.1007/BF01840439.

Cormen, Thomas H. et al. (2009). *Introduction to Algorithms, 3rd Edition*. MIT Press. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

A LTL Formulas for Information Sharing

```
never
bel(
  cellsInVision(Cells),
  member((RelativeX,RelativeY), Cells),
  findall(obstacle(RelativeX,RelativeY, Status,TimeStamp),
          obstacle(RelativeX,RelativeY, Status, TimeStamp),
          Obstacles
  ),
  length(Obstacles, Length),
  Length > 1
).
```

Figure A.1: Visible cells that have contained or contain an obstacle must be unique by their location at any time.

```
bel(
  step(Step),
  (
    not(shareInfo(obstacle(RelativeX, RelativeY, Status, Step), team));
    (
      shareInfo(obstacle(RelativeX, RelativeY, Status, Step), team),
      relativeToLocalCoordinates(RelativeX, RelativeY, LocalX, LocalY),
      (
        perceivedObstacle(LocalX,LocalY) ->
          Status = present;
          Status = cleared

      )
    )
  )
).
```

Figure A.2: If the agent has a `shareInfo` belief after leaving the module, then it must correspond to a percept.

```
bel(
  cellsInVision(Cells),
  step(Step),
  forall(
    (
      member((RelativeX,RelativeY), Cells),
      obstacle(RelativeX, RelativeY, Status, TimeStamp),
      relativeToLocalCoordinates(RelativeX, RelativeY, LocalX, LocalY)
    ),
    (
      Step = TimeStamp,
      perceivedObstacle(LocalX, LocalY) ->
        Status = present;
        Status = cleared
    )
  )
).
```

Figure A.3: After leaving the module all perceivable cells that contain or has contained an obstacle must have the timestamp of the current step. If an obstacle is perceived then it will have the status 'present' otherwise 'cleared'.

```
never
bel(
  obstacle(Xtranslated,Ytranslated, Status, Step),
  obstacle(Xtranslated,Ytranslated, Status1, Step1),
  (
    Step \= Step1;
    Status \= Status1
  )
).
```

Figure A.4: Only one belief about an obstacle must be present in the belief base at any time.

```
never
bel(
  receivedMsg(obstacle(X,Y,Status, Timestamp), Agent, PrevStep),
  step(Step),
  PrevStep < Step,
  translateFromAgentsOriginToMyOrigin(Agent,X,Y,Xtranslated,Ytranslated),
  obstacle(Xtranslated,Ytranslated, MyStatus, Timestamp),
  Status \= MyStatus
).
```

Figure A.5: Contradicting information about obstacles with the same timestamp must never occur at any time.

```
bel(
  not((
    receivedMsg(obstacle(X,Y,Status, TimeStamp), Agent, PrevStep),
    step(CurrStep),
    CurrStep > PrevStep,
    translateFromAgentsOriginToMyOrigin(Agent,X,Y,Xtranslated,Ytranslated),
    obstacleBeforeMessageReceive(Xtranslated,Ytranslated, _, MyTimeStamp),
    TimeStamp > MyTimeStamp
  ));
  (
    receivedMsg(obstacle(X,Y,Status, TimeStamp), Agent, PrevStep),
    translateFromAgentsOriginToMyOrigin(Agent,X,Y,Xtranslated,Ytranslated),
    obstacle(Xtranslated,Ytranslated, Status, TimeStamp)
  )
).
```

Figure A.6: An agent should update its knowledge about an obstacle in the module when more updated information arrives.

```
bel(
  not((
    receivedMsg(obstacle(X,Y,Status, TimeStamp), Agent, PrevStep),
    step(CurrStep),
    CurrStep > PrevStep,
    translateFromAgentsOriginToMyOrigin(Agent,X,Y,Xtranslated,Ytranslated),
    obstacleBeforeMessageReceive(Xtranslated,Ytranslated, MyStatus,
       MyTimeStamp),
    MyTimeStamp > TimeStamp,
    MyStatus \= Status
  ));
  (
    receivedMsg(obstacle(X,Y,Status, TimeStamp), Agent, PrevStep),
    translateFromAgentsOriginToMyOrigin(Agent,X,Y,Xtranslated,Ytranslated),
    obstacle(Xtranslated,Ytranslated, MyStatus, MyTimeStamp),
    shareInfo(obstacle(Xtranslated,Ytranslated, MyStatus, MyTimeStamp), team)
  )
).
```

Figure A.7: An agent must share information with connected agents if the agent has contradicting and more updated information.

```
bel(
    not(shareInfo(_, _))
).
```

Figure A.8: The agent must not have any `shareInfo` beliefs after leaving the module.

```
bel(
  shareInfo(Info, Channel),
  name(Name),
  step(Step)
)
leadsto
bel(
  msgSend(msg(Info, Name, Step), Channel)
).
```

Figure A.9: All messages must be sent to their corresponding channel with the sender's name and the current simulation step.

```
bel(
  shareInfo(Info, team),
  name(Name),
  step(Step),
  connectedTeammates(Team)
)
leadsto
bel(
  forall(
    (
      member(Teammate, Team),
      name\=Teammate
    ),
    msgSend(msg(Info, Name, Step), Teammate)
  )
).
```

Figure A.10: A message is sent to all connected agents when team is the channel.

```
bel(
  team(Team), thing(_, _, entity, Team)
).
```

Figure A.11: The agent must have perceived another agent from the same team to enter this module.

```
bel(
  team(Team),
  forall(
    shareInfo(agentShare(LocalX, LocalY, _, _, CommonEnvPercepts), allother),
    (
      localToRelativeCoordinates(LocalX, LocalY,RelativeX, RelativeY),
      thing(RelativeX, RelativeY, entity, Team),
      commonEnvironmentPercepts(CommonEnvPercepts,LocalX, LocalY)
    )
  )
).
```

Figure A.12: For every `shareInfo` belief a corresponding entity and shared percepts, `commonEnvironmentPercepts`, must be present after leaving the module.

```
bel(agentShare(_, _, _, _, _, _, _)).
```

Figure A.13: The agent should only enter the module if the agent has received another agent's percepts, `agentShare`.

```
bel(
  not(agentShare(_, _, _, _, _, _, _)),
  not(commonEnvironmentPercepts(_,_,_))
).
```

Figure A.14: The agent must not have beliefs about percepts that the agent has send, `commonEnvironmentPercepts`, and received, `agentShare`, after leaving the module.

```
bel(
  uniqueAgentShares(UniqueAgentShares),
  member(agentShare(Agent, X, Y, _, _, CommonEnvPercepts), UniqueAgentShares),
  LocalX is -X,
  LocalY is -Y,
  commonEnvironmentPercepts(CepSelf,LocalX,LocalY),
  compareCommonEnvironmentPercepts(CommonEnvPercepts, CepSelf, LocalX, LocalY)
)
leadsto
bel(
  translateOrigins(Agent,OriginX,OriginY),
  HisOriginX is - OriginX,
  HisOriginY is - OriginY,
  receivedInformation(Agent, HisOriginX, HisOriginY, _, _, _, _, _)
).
```

Figure A.15: An agent will connect to another agent and receive information from the agent if the percepts received is validated successfully.

```
never
bel(
  agentShare(Agent, X, Y, _, _, CommonEnvPercepts, _),
  agentShare(Agent1, X, Y, _, _, CommonEnvPercepts, _),
  Agent1 \= Agent,
  uniqueAgentShares(UniqueAgentShares),
  member(agentShare(_, X, Y, _, _, CommonEnvPercepts, _), UniqueAgentShares)
).
```

Figure A.16: Duplicates of `agentShare` beliefs must never be flagged as unique.

```
bel(
    receivedInformation(_, _, _, _, _, _, _, _)
).
```

Figure A.17: The agent should only enter this module if it has received information from another agent.

```
bel(
  not(receivedInformation(_, _, _, _, _, _, _, _))
).
```

Figure A.18: The agent must not have any beliefs about received information, `receivedInformation` after leaving the module.

```
bel(
  receivedInformation(Agent,  HisOriginX, HisOriginY, AgentOrigins, Obstacles,
       Dispensers, GoalCells, TaskBoards)
)
leadsto
bel(
  OriginX is - HisOriginX,
  OriginY is - HisOriginY,
  translateOrigins(Agent,OriginX,OriginY),
  forall(member(obstacle(X1, Y1, Status, TimeStamp), Obstacles),
    (
      translateFromAgentsOriginToMyOrigin(Agent,X1, Y1,Xtranslated,Ytranslated
          ),
      (
        obstacle(Xtranslated,Ytranslated, Status, TimeStamp);
        (
          obstacle(Xtranslated,Ytranslated, _, MyTimeStamp),
          MyTimeStamp > TimeStamp
        )
      )
    )
  ),
  forall(member(dispenser(X1, Y1, Type), Dispensers),
    (
      translateFromAgentsOriginToMyOrigin(Agent,X1, Y1,Xtranslated,Ytranslated
          ),
      dispenser(Xtranslated,Ytranslated, Type)
    )
  ),
  forall(member(goalCell(X1, Y1), GoalCells),
    (
      translateFromAgentsOriginToMyOrigin(Agent,X1, Y1,Xtranslated,Ytranslated
          ),
      goalCell(Xtranslated,Ytranslated)
    )
  ),
  forall(member(taskBoard(X1, Y1), TaskBoards),
    (
      translateFromAgentsOriginToMyOrigin(Agent,X1, Y1,Xtranslated,Ytranslated
          ),
      taskBoard(Xtranslated,Ytranslated)
    )
  ),
  forall(member(translateOrigins(OtherAgent, X1, Y1), AgentOrigins),
    (
      translateFromAgentsOriginToMyOrigin(Agent,X1, Y1,Xtranslated,Ytranslated
          ),
      translateOrigins(OtherAgent, Xtranslated,Ytranslated)
    )
  )
).
```

Figure A.19: The agent will know all the other agent's information after connecting.

```
bel(
  receivedInformation(Agent,  HisOriginX, HisOriginY, _, _, _, _, _)
)
leadsto
bel(
  OriginX is - HisOriginX,
  OriginY is - HisOriginY,
  translateOrigins(Agent, OriginX, OriginY)
).
```

Figure A.20: The agent will be connected to another agent after the agent has received information from the other agent.

B LTL Formulas for Preparation

```
bel(
  name("agentGOAL-DTU1"),allocationRequest(_, _, _, _, _, _, _)
).
```

Figure B.1: The agent must have a belief about a request if `allocateTypes` is entered.

```
bel(
  not(allocationRequest(_, _, _, _, _, _, _))
).
```

Figure B.2: The agent must not have any beliefs about requests when leaving `allocateTypes`.

```
always
bel(
  forall(allocationRequest(Agents, ReceivedExploreAgents, ReceivedBlockAgents,
      ReceivedSubmitAgents, Dispensers, TaskBoards, _),
    (
      checkAgentTypes(Agents, ReceivedExploreAgents, ReceivedBlockAgents,
          ReceivedSubmitAgents, ExploreAgents, BlockAgents, SubmitAgents),
      optimalTypeDistribution(ExploreAgents, BlockAgents, SubmitAgents,
          Dispensers, TaskBoards, _)
    )
  )
).
```

Figure B.3: A distribution must be found for every request.

```
bel(
  allocationRequest(Agents, ReceivedExploreAgents, ReceivedBlockAgents,
      ReceivedSubmitAgents, Dispensers, TaskBoards, _),
  checkAgentTypes(Agents, ReceivedExploreAgents, ReceivedBlockAgents,
      ReceivedSubmitAgents, ExploreAgents, BlockAgents, SubmitAgents),
  optimalTypeDistribution(ExploreAgents, BlockAgents, SubmitAgents, Dispensers
      , TaskBoards, Distribution)
)
leadsto
bel(
  step(CurrStep),
  forall((
      member((Agent,Type,Details), Distribution),
      member((Teammate,_,_,_), Agents)
    ),
    msgSend(msg(agentType(Agent, Type, Details, CurrStep),"agentGOAL-DTU1",
        CurrStep),agent(Teammate))
  )
).
```

Figure B.4: Changes to types must be sent to every agent included in the request.

```
never
bel(
  findall(Agent,
      agentType(Agent, _, _, _),
      AgentTypes),
  select(Agent, AgentTypes, RemainingAgentTypes),
  memberchk(Agent,RemainingAgentTypes)
).
```

Figure B.5: Any agent must never have more than one type.

```
bel(
  name("agentGOAL-DTU1");
  (
    not((
      readyToTeamUp,
      teamLeader,
      getAllocationBeliefs(Agents, ExploreAgents, BlockAgents, SubmitAgents,
          Dispensers, TaskBoards),
      not(typesInBalance(ExploreAgents, BlockAgents, SubmitAgents, Dispensers)
          )
    ));
    (
      getAllocationBeliefs(Agents, ExploreAgents, BlockAgents, SubmitAgents,
          Dispensers, TaskBoards),
      shareInfo(allocationRequest(Agents, ExploreAgents, BlockAgents,
          SubmitAgents, Dispensers, TaskBoards), agent("agentGOAL-DTU1"))
    )
  )
).
```

Figure B.6: The team leader will request the master-agent for new types when they are out of balance.

```
eventually
bel(
  name("agentGOAL-DTU1");
  (
    typesOfAgents(ExploreAgents, BlockAgents, SubmitAgents),
    findall(dispenser(X,Y,Type),dispenser(X,Y,Type),Dispensers),
    typesInBalance(ExploreAgents, BlockAgents, SubmitAgents,Dispensers)
  )
).
```

Figure B.7: Eventually the types will be in balance.

```
bel(
  name(Agent),
  receivedMsg(agentType(Agent, Type, Details, PrevStep), "agentGOAL-DTU1",
      PrevStep),
  step(CurrStep),
  CurrStep =:= PrevStep +1
)
leadsto
bel(
  (Type\= blockAgent ; goalBlockAgent(CurrStep, Details)),
  (Type\= submitAgent ; goalSubmitAgent(CurrStep))
)
```

Figure B.8: An agent should adopt a corresponding goal if it evolves into a block-agent or submit-agent