# Multi-Agent Programming in GOAL

Philip Bratt Ørum and Nicolai Christian Christensen


Kongens Lyngby 2013
B.Sc.-2013-19

Technical University of Denmark
DTU Compute
Matematiktorvet, building 303B, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253031, Fax +45 45881399
compute@compute.dtu.dk
www.compute.dtu.dk B.Sc.-2013-19

# Summary (English)

The goal of this thesis is to evaluate the ideas and strategies used by the multi-agent system that won the 2011 Multi-Agent Programming Contest. We aim to determine whether any of these strategies are useful in the 2012 contest scenario and whether they can be implemented as part of a competitive solution. We also want to evaluate the usefulness and performance of the multi-agent programming language GOAL, which was used to write the original multi-agent system.

Evaluation is based on running simulations of the 2012 contest scenario between the 2011 winners and the team from DTU which nearly won the 2012 contest. We evaluate GOAL based on material provided by its creators as well as by using the programming language for the duration of the project.

Results show that the team from 2011 is inferior to the 2012 team in almost every way. Their strategies are too specialized towards the contest structure in 2011 to be transferred to later editions. For this reason it would not be worth it to adapt this system to any future contests. The GOAL programming language is intuitive and easy to use. Several bugs are present but as only alpha releases exist this is to be expected. We believe that GOAL will be a strong tool for multi-agent systems in the future.

# Summary (Danish)

Målet for denne afhandling er at evaluere de ideer og strategier der blev benyttet i det multi-agent system, som vandt Multi-Agent Programming Contest i 2011. Vores formål er at bestemme om disse strategier er brugbare i 2012 scenariet for konkurrencen og om de kan implementeres som del af et konkurrencedygtigt program. Vi vil også evaluere anvendeligheden og ydeevnen af multi-agent programmeringssproget GOAL, som det pågældende multi-agent system blev skrevet i.

Evalueringen sker på baggrund af simulationer, hvor vinderne fra 2011 dyster mod holdet fra DTU, som næsten vandt konkurrencen i 2012. GOAL bliver evalueret på baggrund af det materiale, der er skrevet af dets skabere, samt ved at arbejde med programmeringssproget gennem hele projektet.

Resultaterne viser at holdet fra 2011 er dårligere end det fra 2012 på næsten alle punkter. Deres strategier er alt for specialiserede i forhold til strukturen på konkurrencen i 2011 til, at de kan overføres til senere udgaver. Af denne grund vil det ikke være tiden værd at prøve at tilpasse systemet til fremtidige konkurrencer. Programmeringssproget GOAL er intuitivt og nemt at arbejde med. Der er en række fejl i sproget, men siden det kun er udgivet i alpha-versioner, er dette forventet. Vi tror at GOAL bliver et vigtigt værktøj for multi-agent systemer i fremtiden.

# Preface

This thesis was prepared at the department of DTU Compute at the Technical University of Denmark in fulfilment of the requirements for acquiring a B.Sc. in Informatics.

The thesis deals with multi-agent systems in general and strategies used in the Multi-Agent Programming Contest specifically, as well as the agent programming language GOAL.

The thesis consists of theory about agent systems, GOAL and the Mult-Agent Programming Contest. It concerns the strategy used by the winning team of the 2011 MAPC and through testing determines whether or not the strategy is good enough to compete in the 2012 MAPC.

Lyngby, 01-July-2013

Philip Bratt Ørum and Nicolai Christian Christensen

# Acknowledgements

We would like to thank our supervisor Jørgen Villadsen for help and guidance throughout the project.

We would also like to thank Øyvind Grønland Woller and Andreas Viktor Hess for discussions about GOAL and the bugs we encountered in it.

Finally we would like to thank Mikko Berggren Ettienne and Steen Vester who created the original Python-DTU system and Andreas Frøsig and Kenneth Balsiger Andersen who have also worked with Python-DTU for helping us make sense of the original python code.

# Who Did What

Below we list who was mainly responsible for writing each chapter.

**Philip:**

- Chapter 2
- Section 4.1
- Chapter 5
- Chapter 6
- Chapter 7

**Nicolai:**

- Chapter 1
- Chapter 3
- Section 4.2
- Chapter 6
- Chapter 8
- Chapter 9
- Appendix A

# Contents

CHAPTER 1

# Introduction

The idea of artificial intelligence has existed for many years and the subjects studied in this field have varied over time. Artificial intelligence started with a desire to build actual cognitive agents that exhibited human-like intelligent behavior. When people realized implementing such agents was impossible with the knowledge available, focus shifted to better understanding the constituent parts needed for an intelligent agent[1]. As such, the field of artificial intelligence has evolved from focusing solely on agents with so called artificial general intelligence[2] into trying to solving subproblems that will eventually help to realize the original goal. Even though no agents with even near human intelligence have been created today, the complexity of agents has evolved a great deal.

Multi-agent systems are systems for problem solving which employ several intelligent, autonomous agents. These agents are designed to cooperate, enabling them to solve complex problems that are too challenging for a single agent. In the study of Multi-agent Systems an annual contest, the Multi-Agent Programming Contest[3], is held to provide an environment for increasing the knowledge on these systems.

[1] [Wik13a]
[2] Also called Strong AI, described in [Wik13d]
[3] [DKS13]

In this thesis, we study multi-agent systems in the context of the Multi-Agent Programming Contest. We analyze the HactarV2 team that won the 2011 contest and compare their strategies to the Python-DTU team that nearly won in 2012. The code for Python-DTU has since been updated and we now consider it to be the strongest team to have been created for the 2012 contest.

The HactarV2 multi-agent system is written in the relatively new agent programming language GOAL so we have chosen to focus on this programming language for our project. Since this is a new language, it has not been studied in detail before. This allows us to evaluate its usefulness as an agent programming language.

We start by providing theory on agent systems, GOAL and the Multi-Agent Programming Contest needed to understand our work. We then go on to explain the changes we made to the HactarV2 code, both to get it to communicate with the new server structure as well as for its strategies to function on the new map layout. We now analyze the final version of the HactarV2 code after all our implementations and move on to present our results gathered from testing. Finally, we briefly describe our experiences with GOAL as an agent programming language and finally we conclude on our project results.

CHAPTER 2

# Agent Systems

We will start by introducing some of the fundamentals of multi-agent systems, like the concept of an agent and the basic ideas behind multi-agent systems.

## 2.1 Rational agents

The most basic component of a multi-agent system is the *agent*. An 'agent' is commonly defined as any entity, real or virtual, that can perceive its environment and is capable of acting upon it[1]. This thesis deals with software agents in particular, but the term is often used to describe real world entities as well.

The type of agent we are interested in is called a *rational agent*. Such an agent has one or more goals which it will try to accomplish by manipulating its environment. The general idea is that an agent is able to perceive its environment, at least to some degree. The agent can then process the information it perceives and turn it into a set of beliefs about the current state of the environment. It uses these beliefs coupled with any new percepts the agent receives to select what actions it needs to perform in order

[1] [Hin11] pages 9-10

to achieve its goals. Such an agent is rational because it will pursue its goals to the best of its ability, based on its understanding of the environment around it[2].

An important property of agents which is often used to differentiate agent programs from other programs is *autonomy*[3]. Agents decide what actions to perform autonomously, i.e. based solely on the agents own beliefs and percepts and without interference from other agents or entities. Most agents are created to work within specific environments or solve specific types of problems, and as such they rely on prior knowledge of the environment when making decisions[4]. While such agents are not completely independently they still make all their decisions independent of other entities, and so it still makes sense to refer to them as autonomous agents.

An important advantage of using an agent based approach to problem solving is that rational agents are able to make decisions about what to do and how to react to various situations at run time. This means that the creator of the system does not need to consider every possible scenario when designing the system, a task that is often impossible when solving sufficiently complex problems[5].

## 2.2 Multi-agent Systems

While single agents can be useful in a number of circumstances, it is often necessary to be able to model more complex systems that contain multiple entities, each with their own set of goals and beliefs[6]. A group of cooperating agents that are working together to solve a problem is called a multi-agent system (MAS).

For such agents to be able to cooperate, it is important that the roles of the agents and the relationships between them are clearly defined. Each agent needs an understanding of its role in relation to the other agents, and needs to be able to make autonomous decisions about how to interact with other agents[7].

[2] [RN10] page 4
[3] [Hin11] page 10
[4] [RN10] page 39
[5] [Jen00] page 284
[6] [Jen00] page 280
[7] [Jen00] page 280

The advantage of multi-agent systems over a more centralized approach is that decisions are made locally, making the system much more robust. If an unforeseen situation occurs, an autonomous agent is able to react to it locally, making the system as a whole more flexible and responsive[8]. Additionally, it is natural to use a multi-agent approach for problems which contain a number of similar entities, since having a close correlation between the theoretical model and the concrete problem being modeled reduces the complexity of the system[9].

[8] [Jen00] page 285
[9] [Jen00] page 286

CHAPTER 3

# GOAL Programming Language

In this chapter we describe the GOAL programming language. We briefly discuss the thoughts behind it, then describe the features and the basic structure of programs.

## 3.1 Mental State

The GOAL programming language is a language developed for creating multi-agent systems where rational agents operate in a predefined environment. The language has been designed based on a strategy called the Intentional Stance[1]. As quoted in the GOAL manual, it is described by Dennett[2] as a strategy that "consists of treating the object whose behavior you want to predict as a rational agent with beliefs and desires (...)". This strategy provides an abstraction from data stored in a database by labeling it with more relatable names such as knowledge, beliefs and goals. These labels make the language easily accessible and straightforward to use.

[1] [Hin11] page 11
[2] [Wik13b]

The agent's knowledge base, its beliefs about the environment and the goals it wants to achieve are collectively called the mental state of the agent[3]. They determine what the agent knows about the state of the environment, as well as what it desires the environment state to be like. In addition to having a mental state, as mentioned earlier, an agent can also handle percepts, communicate with other agents and take actions. GOAL is designed to facilitate the representation of these aspects of an agent by providing a simple interface where each of these elements can be represented. Collectively this part of a GOAL program is called the init module. The init module is shown in Figure 3.1.

```
1  init module {
2   knowledge{
3    % insert knowledge here, if any, or remove section.
4   }
5
6   beliefs{
7    % insert initial beliefs here, if any, or remove section.
8   }
9
10  goals{
11   % insert initial goals here, if any, or remove section.
12  }
13
14  program {
15          % insert one-time rules here, or remove section
16      }
17
18  actionspec{
19   % insert global action specification here, if any, or remove section.
20  }
21 }
```

**Figure 3.1:** Init module skeleton of a newly created MAS in GOAL

To describe an agent's mental state, as well as its action specifications, a Knowledge Representation[4] language is required. GOAL does not need any specific language, but uses SWI-Prolog as its standard language.

The knowledge of an agent is represented by a set of predicates in the Knowledge Representation language. The knowledge base is static, and prior knowledge which the agent might possess must be entered here before running the MAS that the agent is part of. It is recommended to put any rules or definitions into the knowledge base[5]. This is the logical thing to do in most cases, as we do not usually want rules and definitions to change during runtime. Any entries in the knowledge base are considered together

[3] [Hin11]
[4] [Hin11] page 19
[5] [Hin11] page 22

with the agent's beliefs when evaluating mental state conditions in the belief base (see below). This allows an agent to have a complete "view" of the environment built from knowledge of rules and definitions, as well as beliefs about the current world state that it has perceived, provided its belief base is complete. For an agent's belief base to be complete, the environment has to be fully observable for the agent. It is also the case that any goals are evaluated in combination with the knowledge base. Care should be taken, when letting agents adopt goals, that entries in the knowledge base do not cause undesired goals to be inferred, and rules should not be defined without considering how they might impact the agent's goals[6].

The beliefs section of an agent is dynamic, and predicates can be inserted and removed while the MAS is running. This is used to update the mental state of the agent as it receives percepts from the environment or messages from other agents. The belief base usually starts out empty, as any knowledge that the agent has prior to running is usually certain knowledge rather than a belief. When perceiving the environment, it is crucial to know whether or not the environment state is fully observable for the agent, as the current beliefs are used to evaluate rules the agent must follow. If we assume that the state of the environment, from the point of view of the agent, is complete, then we can take advantage of the fact that Prolog operates under a closed world assumption[7], meaning that anything not known to be true is assumed to be false. If the state of the environment is not fully observable, we have to define our rules around the fact that no assumptions can be made, making rules definitions more demanding.

The goal section of an agent holds definitions about what the agent wants to achieve. An agent can start with goals that it wants to achieve and never deviate from fulfilling those specific goals, or it can adopt new goals or drop old ones based on the logic that defines the agent's behavior. When a goal is believed to be achieved, i.e. it appears in the agent's belief base, it is removed from the goal section automatically. This is done because it would be irrational to pursue a goal that is already believed to be achieved[8].

[6] [Hin11] page 28 - "Where to Put Rules?"
[7] [Hin11] page 23 - "Closed world assumption"
[8] [Hin11] page 37

## 3.2 Reasoning Cycle

Every time an agent wants to act, it goes through a reasoning cycle[9] trying to determine the best course of action. To represent this reasoning, GOAL suggests using two special modules, each representing a specific part of the cycle. Only one of these modules is actually required in an agent, but if both are present they will be executed in a specific order.

The first one is the event module. This is designed for handling percepts from the environment and communication with other agents if necessary. The system goes through this module first, as it will bring the agent up to speed with all changes that might have occurred in reaction to last cycle's actions. The mental state of the agent is updated by adding and removing predicates from the belief base and possibly changing the goals of the agent based on new information.

The second module is the main module. This is supposed to contain the actual reasoning for picking the correct action. By following rules and utilizing it's current mental state, the agent should eventually arrive at an action call. If the precondition for an action is fulfilled the action will be executed. GOAL supports additional module creation for grouping code together. These modules act similarly to functions without return types and have the option of taking parameters, making it easy not to clutter up the event and main modules.

[9] [Hin11] page 44

CHAPTER 4

# Agents on Mars and the HactarV2 Team

The specific multi-agent scenario we study in this thesis is the 'Agents on Mars'-scenario from the 2011 - 2012 Multi-Agent Programming Contest[1]. We will give a basic explanation of the scenario here.

## 4.1 Contest scenario

In the 'Agents on Mars'-scenario, two teams of agents are fighting for control of a number of water wells on the surface of Mars. Each team consists of a number of independent agents with different roles (as detailed in the tables below). These agents must work together to control as many wells as possible, while at the same time preventing the other team from doing the same. Each team receives a number of points each turn based on the amount and value of the wells they control and the number of achievement points the team has. Achievement points are awarded for reaching certain milestones, like controlling an area of a certain size or inspecting a certain number of enemy agents, so teams are rewarded for being versatile and utilizing all of their op-

[1] [DKS13]

tions. A match consists of 750 rounds, and the winner is the team with the highest score at the end of the match.

A match is played out on a graph (called a map) consisting of a number of vertices connected by edges. The vertices represent the water wells, while the edges represent the paths between them. Each edge has a value which represents the cost, in energy, of traversing it.

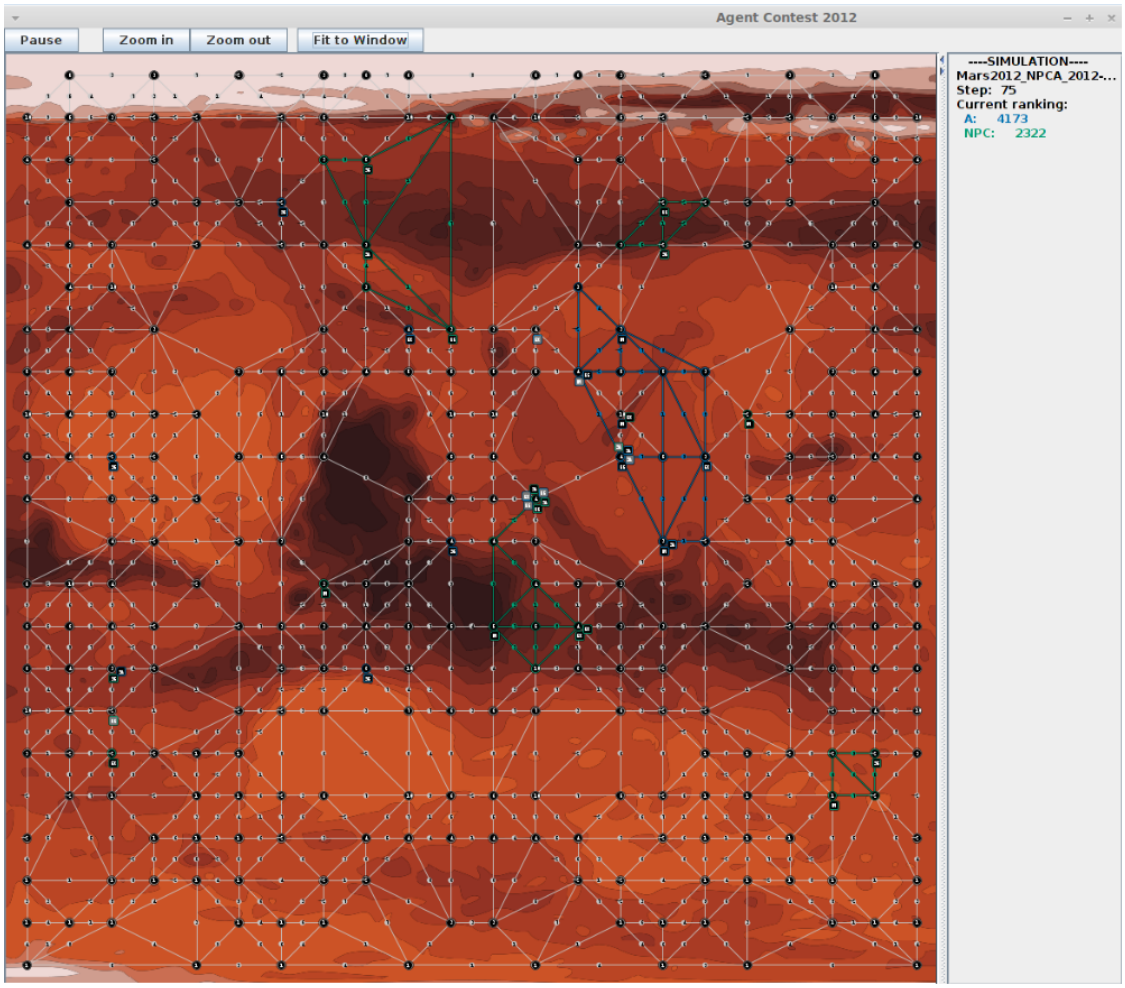


**Figure 4.1:** A screenshot from a running simulation

A special coloring algorithm is used to determine what areas of the graph (zones) are controlled by each team. The algorithm consists of the following four steps[2]:

1. The first phase of the calculation only involves the coloring of vertices that have agents standing on them. We say that a vertex *v* is dominated by a team *t* if *t* has the majority of agents on *v*.
2. The coloring is extended to empty vertices that are direct neighbors of dominated vertices. A team needs to dominate at least two neighboring vertices of an empty vertex to be able to color that empty vertex.
3. Some of the vertices that were colored with a team name *t* in the previous two steps might represent a frontier that isolates a part of the graph from all the other teams' agents. We say that an empty vertex has been isolated by a team *iff* for all agents *ag* belonging to the other team *t2* there is no path from *ag* to *v* that does not include a vertex colored by *t*. The coloring is extended to all vertices isolated by each team.
4. A node is not colored *iff* the other conditions are not satisfied.

This algorithm is run by the server each round and the result is sent to the agents as percepts, so an agent is always aware of what parts of the graph its team controls.

During each round all agents must choose an action to send to the server, and the server then determines the result of each agent performing its chosen action. The agents have a limited amount of time to decide what to do, and if the time limit is exceeded the agent forfeits its turn. The agents can share some information between them, and percepts are automatically shared between agents in the same zone of control, but each agent is responsible for telling the server what it wants to do.

[2] [BDH+11] page 5

A team consists of the agents seen in Table 4.1 and descriptions of all possible actions can be seen in Table 4.2.

**Explorer:** skip, goto, probe, survey, buy, recharge

**Repairer:** skip, goto, parry, survey, buy, repair, recharge

**Saboteur:** skip, goto, parry, survey, buy, attack, recharge

**Sentinel:** skip, goto, parry, survey, buy, recharge

**Inspector:** skip, goto, inspect, survey, buy, recharge

**Table 4.1:** The available actions for specific roles

**Attack:** attacks a vehicle.

**Buy:** buys an item, upgrading the agent.

**Goto:** moves to a vertex.

**Inspect:** inspects a visible entity to determine its role and status.

**Parry:** parries all attacks made against the agent this round.

**Probe:** probes the current vertex, determining its value.

**Recharge:** recharges the vehicle, restoring energy.

**Repair:** repairs a vehicle that has been damaged by an attack.

**Skip:** does nothing.

**Survey:** surveys a visible edge, determining its cost.

**Table 4.2:** Descriptions of all available actions

All actions except **recharge** cost some amount of energy, which the agents must periodically recharge. For a more detailed explanation of the actions and their effects, see [BDH+12] pages 6-7.

### 4.1.1 2011 vs. 2012

The parts of the contest discussed so far are true for all versions, but there are a few key differences between the Multi-Agent Programming Contest from 2011, which the HactarV2 team participated in, and the current[3] 2012 version. In the 2011 scenario, each team consisted of 10 agents (2 of each type). Each map had one global optimum containing the highest value nodes. In the 2012 scenario, each team consists of 20 agents (4 of each), and the maps contain no one global optimum, but instead a number of local optima, several of which have equal value.

## 4.2 HactarV2

This is a description of the original HactarV2 agent team, based on the document[4] that was provided by HactarV2's creators from TU Delft.

### 4.2.1 Design

HactarV2 is designed as a decentralized MAS where each agent acts autonomously on the map. Even though the agents make their own decisions they still cooperate to achieve common goals. Agents will send information about nodes that have been explored to the other agents. This ensures that the same work is not done twice by different agents, and this way an agent will quickly get a sense of the entire map which helps the agent move about more efficiently. Since each agent acts for itself all of them will navigate around the map computing their own routes using a set of pathing algorithms.

Messages between agents have been kept to a minimum to ensure efficiency in the program. As a result, agents will only send useful information but never discuss what actions to take with each other. The message inbox will only be checked at the start of a cycle, while messages can be sent throughout the cycle. Therefore messages received will be from the previous step and not the current one. Because of this, the agents have been designed to rely heavily on their own percepts of the surrounding area rather than on information from teammates.

[3] At the time of writing a new 2013 version is forthcoming, but it has not yet been released
[4] [DHH+12]

To ensure coding efficiency and to minimize bugs, the agents have been designed from the same basic agent template, but with special behavior based on the role of the agent. This design choice means less duplicate code and a program structure that is easy to follow. Shared behavior has been put into different modules which each agent can access when necessary. Each of the different roles also has a module that defines the special behavior of agents of that type.

### 4.2.2 Strategy

The HactarV2 team uses a strategy that is divided into two phases. In the first phase the agents roam around on their own, sending messages to each other about the state of the map and the opposing team but not directly cooperating. The explorers probe the nodes on the map looking for the optimum node to capture and hold. As mentioned, in the 2011 scenario only one global optimum existed on the map and HactarV2 uses this information to search for nodes of higher and higher value, effectively moving towards the global optimum.

Meanwhile the saboteurs are taking the fight to the enemy, disrupting their operations and hopefully keeping them from exploring the map. The saboteurs are made fairly effective by means of the buying strategy for upgrades that HactarV2 uses. The designers of HactarV2 deemed saboteurs to be the most important agent type because of the damage they can do to the enemy. For this reason they aggressively buy health and strength for the saboteurs from the start of the game. This costs them a lot of valuable achievement points early on and therefore they are often behind in the early stage of a game. They accept this penalty because they have a very strong technique for securing the optimum and will make up for the lack of total points later in the game. They are still aware of the importance of achievement points, however, and have decided not to buy upgrades for any of the other agents to save points. To generate the most achievement points they focus heavily on the most worthwhile achievements, like parry, and as an example their agents will more often use the parry action when enemy saboteurs are near rather than flee from them.

After the optimum is found the initial phase of hactarV2's strategy is done. Since the explorers can search for more and more valuable nodes, this usually does not take long. The explorers will broadcast the optimum node to the team and the agents will now all begin to *swarm* around this optimum node. Explorers will update the optimum if a higher value node is found later though.

In the swarming phase explorers will probe all nodes around the optimum. Assuming the found optimum has the maximum value 10, node values decrease gradually from here due to the map design in 2011. Nodes will then be probed in concentric "circles" outwards from the optimum until a specific value is reached. The agents will then spread out from the optimum node to cover what is called the optimum zone. The agents will maintain coherency and hold this most valuable position on the map. Saboteurs will no longer be aggressive but join the swarming and only attack opponents who come too close to the optimum zone. Holding this global optimum and the area around it is what led HactarV2 to victory in the 2011 MAPC.

CHAPTER 5

# Core Strategy

The main purpose of our program is to evaluate the strategy and performance of HactarV2 in the 2012 scenario in comparison to Python-DTU, so our final program still has the same core strategy as the original version. We want to create a system that works well in the 2012 scenario, while still maintaining the core parts of the HactarV2 strategy. We can then evaluate the viability of those parts in the new scenario. In this chapter we describe exactly what these core strategies are.

## 5.1 Buying strategy

HactarV2's strategy for buying upgrades is for the saboteurs to constantly ensure that they have higher strength than the highest health they have seen on an enemy and higher health than the strength of the strongest inspected enemy saboteur. In order to conserve achievement points, no other agents are ever upgraded. This strategy can cost a lot of achievement points (resulting in a lower score), but it allows the team's saboteurs to defeat any enemy in one attack, while also ensuring that they take at least two attacks to disable themselves.

## 5.2 Zone control

When an optimum[1] has been determined, all available agents will work together to establish a zone of control around it. All the teams explorers will immediately prioritize probing the area inside this optimum zone to ensure that we obtain the full amount of points from it each round. Saboteurs will still fight the enemies they encounter and reairers will still repair damaged allies, but otherwise all agents will move towards the optimum. When they reach it, the agents will try to position themselves in a ring around the optimum, making sure always to be close enough to each other to maintain a zone of control[2] around the optimum. Agents do this by ensuring that they are at the edge of the optimum zone while also maintaining a distance of 2 nodes to at least two other agents. We refer to this behaviour as ”swarming”.

Agents that are busy with other tasks such as defending the zone, probing or repairing are labeled as ”unreliable” in terms of maintaining the swarm, and are not considered when the swarm is established. If an enemy enters the optimum zone, the saboteurs will detach from the swarm to deal with them, but otherwise they will stay defensive and participate in swarming around the optimum. This makes a lot of sense in an environment with only one optimum, since maintaining control of the most valuable nodes is obviously a priority. It is less obvious how well it works in an environment with multiple possible optima, but that is what we hope to determine by using it here.

[1] A local maximum in terms of node value
[2] See [DHH+12] for an explanation of how zone control works

CHAPTER 6

# Adapting to the 2012 scenario

The HactarV2 code needed a number of modifications before it could be tested in the 2012 scenario, both in terms of connecting properly to the server and following the new communication protocols, as well as adapting their strategy to reflect the changes to the scenario that were introduced in the 2012 contest. In this chapter we describe this process, including some of the ideas we considered over the course of the project, and the solution we ended up with.

## 6.1 Connecting to the new setup

The first thing we had to do was to update the HactarV2 code so it could compete in the mars 2012 scenario. To differentiate our version of the code from the original, we changed the team name to NPC (Nicolai-Philip-Complete). The new team name is used in the code, but for simplicity we will not reference it in the thesis but keep using the team name HactarV2. Because of the changes in the competition from 2011 to 2012 as well as the new team name, we had to make several minor adjustments to different files. A file by file list of changes can be found in Appendix A.

We had to double the number of agents on the team to make it fit the rules of the 2012 scenario. This meant preparing the server for our team, adding new entities to the environment as well as adding agents in the goal code.

In the configuration for the environment we turned off the scheduling feature as HactarV2 has designed its own way of scheduling agents between rounds.

In the 2011 version of GOAL, agents were connected to the corresponding entities in the environment by their type. The type was simply *mars2011entity*. Unfortunately, this method of connecting no longer works in the 2012 scenario, even when the year is changed to 2012. A solution to this is to connect using the individual entity names instead of using the type, as seen in Figure 6.1.

```
1 launchpolicy{
2 % Launch all the agents with a type corresponding to the one they have in the simulation
3   when [type=mars2012entityunknown,max=1]@env do launch NPC1:mapc.
4   when [type=mars2012entityunknown,max=1]@env do launch NPC2:mapc.
5   when [type=mars2012entityunknown,max=1]@env do launch NPC3:mapc.
6   when [type=mars2012entityunknown,max=1]@env do launch NPC4:mapc.
```

**Figure 6.1:** Connecting to entities by name

This connection issue is due to an error in the implementation of the 2012 environment. The type of agents were meant to be role specific in the 2012 scenario, i.e. *mars2012entityexplorer* , but the specific role part was never added to the definition of type. Instead the default value unknown was used. Knowing this, it is possible to connect to entities using types as well, as seen in Figure 6.2.

```
1 launchpolicy{
2 %Launch all the agents with a name corresponding to the one they have in the simulation
3   when [name=NPC1]@env do launch NPC1:mapc.
4   when [name=NPC2]@env do launch NPC2:mapc.
5   when [name=NPC3]@env do launch NPC3:mapc.
6   when [name=NPC4]@env do launch NPC4:mapc.
```

**Figure 6.2:** Connecting to entities by type

## 6.2 Adapting to the new map structure

After getting the HactarV2 code working on the 2012 server, we started testing the strategy of HactarV2 against the updated Python-DTU team. The Python-DTU team had come close to winning the 2012 MAPC but ended up in a close second place. They lost because the buying strategy they were using could be exploited, causing them to spend too many achievement points. After fixing their strategy to minimize the risk of exploitation, the team won against every participant from the 2012 contest.

As soon as we pitted the unedited HactarV2 code against Python-DTU, it immediately became clear that Python-DTU was the strongest. In none of the matches we played was it ever a question of whether or not they would win, it was always how much would they win by. Early in a match Python-DTU gained more points from achievements than the HactarV2 team, and later when both teams tried to establish and control an area they were significantly more efficient. When the HactarV2 agents were actually able to set up a perimeter and hold it, they seemed to struggle to maintain control of their optimum zone.

We had deliberately tested the program after only changing what was needed to connect to and communicate with the 2012 server in order to see how the original HactarV2 performed in the new setting. When looking at the original code, it quickly became obvious why it performed so poorly in comparison to Python-DTU. Even though we had adapted it to the modifications made to the server, parts of the code still relied on assumptions that were no longer true. For example, we had added the 10 extra agents for the total of 20 in the 2012 MAPC, but some of the code relied on the number of agents on a team, but HactarV2 sometimes relied on the exact number of saboteurs on the enemy team to determine whether or not an agent was endangered by the presence of an opponent saboteur. When encountering an unknown opponent, they would rule it out as a saboteur if two other saboteurs had already been inspected. Since there were now four saboteurs on the enemy team, the predicate determining this was no longer compatible with the game rules and had to be updated.

### 6.2.1 Changing the strategy

The strategy, as described earlier, was to find an optimum node and have the agents swarm around it, giving them control over the area of highest values. We knew from the beginning that the map design in 2012 would make this kind of strategy less useful, but the implementation of the swarming part turned out to be practically useless. The nodes around the global optimum in the 2011 map have values that decrease gradually

as you get further away from it (see Figure 6.3). The HactarV2 team used this map design to define their optimum zone by sticking to nodes with fixed values ranging from 7 up to the maximum value of 10. In 2012 the map is created with several local optima, but surrounding nodes are not guaranteed to have gradually decreasing values, and as such it is impossible to define an area based on node values in this way. The idea was extremely efficient for the 2011 competition but unfortunately could not threaten the Python-DTU team with the new map layout.

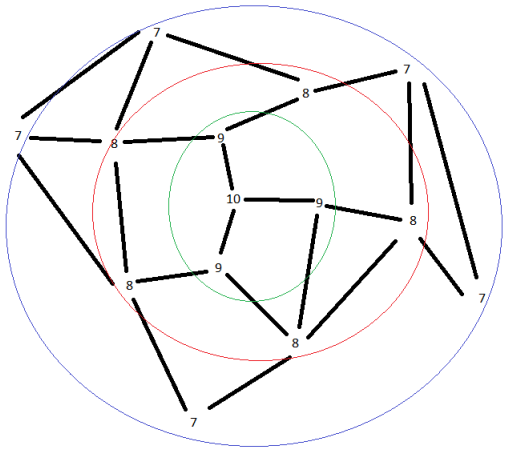


**Figure 6.3:** Decreasing node values around global optimum - 2011

To be able to test the strategy of the HactarV2 team we had to update a number of things about it. In the original code, the agents tried to gain control of the optimum zone as early as possible. Explorers used a greedy strategy, always moving towards higher value nodes when possible, in search of the optimum. This kind of strategy makes sense in an environment with only one optimum, since gaining control of the optimum zone not only ensures that the team get a lot of points but also starves the other team of points, as there are no other profitable locations on the map.

Because of the multiple optima in the 2012 scenario, however, it is much more difficult to deny the opponent a good location. In our opinion, this puts a greater emphasis on exploring and gaining achievement points in the early game, so inspired by the approach of Python-DTU[1], we have implemented an exploration phase in the first 150 steps of the competition. In this phase, agents try to gain achievement points by fulfilling their roles (saboteurs attack, sentinels parry, etc.), while exploring the map. Instead

[1] [AF12] page 4

of stopping as soon as an optimum is found, potential optimum locations are recorded and compared, and only when we reach step 150 do the agents decide on an optimum and move to secure it. This behavior is implemented in the *searchOptimal* module, which can be seen in Figure 6.4.

```
% Module that contains behavior for explorers to find the optimal value node
module searchOptimal {
 program[order=linearall]{
 if bel(currentPos(Here), !, vertexValue(Here, Value), optimumValue(Current), Current<Value)
  then {
   if bel(step(N), N<150) then insert(tempOptimum(Here), not(optimumValue(Current)), optimumValue(
        Value)) + send(allother, tempOptimum(Here)).

   if bel(step(N), N>=150, not(optimum(_)),calcOptimumZone(Here,OZone)) then insert(optimum(Here),
        not(optimumValue(Current)), optimumValue(Value)) + send(allother, optimum(Here)) + send(
        allother, optimumZone(OZone)).
  }

  if true then {
   % find an unprobed neighbouring vertex
   if bel(neighbour(There), needProbe(There), safePos(There)) then advancedGoto(There).

   % find an unprobed neighbouring vertex
   if bel(neighbour(There), needProbe(There), not((visibleEntity(_, There, Team, _), enemyTeam(Team))))
    then advancedGoto(There) + insert(noFlee).

   % find an unprobed neighbouring vertex
   if bel(neighbour(There), needProbe(There))
    then advancedGoto(There) + insert(noFlee).

   % Find closest unprobed vertex
   if bel(currentPos(Start), pathClosestNonProbed(Start, NonProbedVertex, [Here,Next|Path], Dist))
    then advancedGoto(Next).
  }
 }
}
```

**Figure 6.4:** Module allowing explorers to find the optimum node

When the simulation reaches step 150 this module is disabled and the best location on record is set as the optimum. This is a one-time action, meaning that it needs to be performed by exactly one agent at a certain point in time. In order to achieve this, we have extended the rank system found in the original HactarV2 code[2], see Figure 6.5.

```
1 % Returns the rank(based on its name) of an agent compared to all other agents of a specific role
2 agentRankRole(Rank, Role) :- me(Name), team(Team), !,
3  setof(Agent, (visibleEntity(Agent,_,Team,normal), role(Agent, Role)), Agents), agentRank(Agents,Name,
       Rank).
```

**Figure 6.5:** Predicate that allows ranking of agents of a specific role

Using this predicate, we can set $Rank =:= 0, Step =:= 150$ as preconditions[3], which allows us to create the one-time action described above.

It was also necessary to change how the optimum zone is determined. Originally, when the optimal optimum was found it would be a node with the value 10. Its neighbors would have values of either 9 or 10. Generally the values of neighbors of node $m$ were: $1 <= v(m) - 1 <= v(n_m) <= v(m) + 1 <= 10$, where $v(n_m)$ is the value of any neighbor of $m$.

This provided an easy way of expanding outwards from the optimum by having explorers probe nodes with gradually lower values and stop at a specified value, see Figure 6.3.

[2]This feature is not documented in the extended abstract for HactarV2, but the original version can be seen in appendix Section B.1

[3]In the actual implementation this is written as $Rank =:= 0, Step >= 150, not(Optimum(\_))$. This is due to a bug in GOAL which sometimes causes a step to be skipped

```
1  % When there are no more neighbouring vertices with higher values or unprobed values, the optimum is
       found
2  if bel(currentPos(Here), !, vertexValue(Here, Value), not((neighbour(N), vertexValue(N, NValue), (
       NValue == unknown; NValue > Value))),
3    Value2 is Value−1, Value3 is Value−2)
4   then insert(optimum(Here), needExploring(Value), needExploring(Value2), needExploring(Value3)) +
       send(allother, optimum(Here)) + swarmProbe.
```

**Figure 6.6:** Finding the optimum in 2011

As can be seen in Figure 6.6, when an explorer has found what it believes to be the optimum, it determines what node values correspond to the first three layers around the optimum and marks them as being in need of probing. All explorers will then focus on probing the corresponding nodes. Because very little probing is done before the optimum is discovered, the agents can then look at whether a node has been probed or not to determine where they need to be in order to control the entire area around the optimum.

In the 2012 scenario this is not possible since you cannot predict the values of nodes around the optimum (see Figure 6.7), so we had to create the perimeter of our optimum zone based on something else.

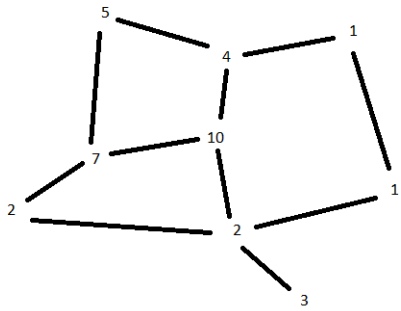


**Figure 6.7:** Decreasing node values from local optimum - 2012

The first approach considered focused on doing something entirely different from finding a node with value 10. It was based on the fact that nodes in the corners of the graphical display of the map have fever edges connecting them to neighboring nodes. The number of edges is two or three for nodes in the corners and three or four from nodes along the borders compared to four or more edges going out from nodes in the middle of the map. This means that it should be possible to locate a node in the corner or at least on the border of the map by counting the number of outgoing edges from it. Our idea, then, was to find such a corner node and move outwards from there. Taking advantage of the map coloring algorithm, we would be able to cut off and hold a huge section of the map (see Figure 6.8). This strategy would attempt to win solely on quantity of nodes instead of quality.

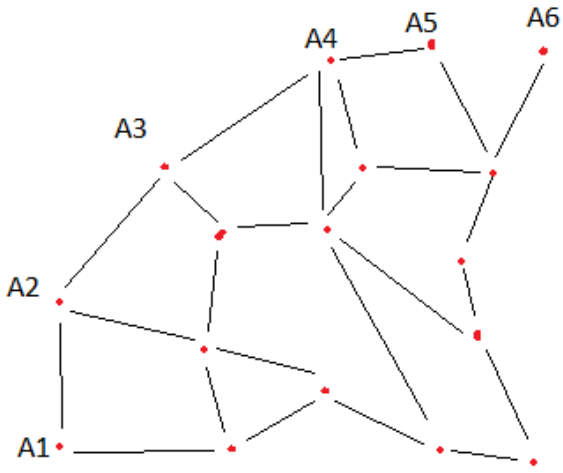


**Figure 6.8:** Quantity of nodes strategy

A big problem with this strategy was that it was too easy to disrupt. If the enemy team sent just one agent behind our perimeter we would lose control of almost all of our nodes. Another problem was reliably finding a node in a corner to designate as our optimum node. If we ended up with the optimum node being on the border of the map, our optimum zone would have a larger circumference to area ratio. This would make us even more vulnerable to attack and would require more agents to hold the perimeter. As if these problems were not discouraging enough, the old swarming strategy would not be particularly useful. It would have had to be rewritten so the agents could spread out and hold a line. We decided we were not interested in altering the strategy too much, as we would still like to know if the strategy at its core had any merit. All of these problems made us discard this idea before wasting too much time on it.

Our final solution was to simply calculate an optimum zone of a predetermined size around the optimum. When the optimum is chosen, one of the explorers (chosen using the rank system described earlier) determines the optimum zone using the formula seen in Figure 6.9:

```
calcOptimumZone(Opt, OptZone) :-
 neighbour(Opt,Primary),
 findall(X,(member(Y,Primary),neighbour(X,Y),not(member(X,Primary))),Secondary),
 merge(Primary,Secondary,OptZone).
```

**Figure 6.9:** Algorithm for determining the optimum zone

This creates a list of all nodes within two steps of the optimum, which is then distributed to all agents on the team. The algorithm is somewhat time-consuming, but since it is a one-time calculation performed by one agent on the team, it should not affect our performance in any meaningful way.

Explorers need to probe the entire optimum zone before they participate in the swarm, so they maintain another copy of the list which is updated to contain only the unprobed nodes in the optimum zone, which they then work together to probe (see Figure 6.10). They use the rank system to ensure that they work on different nodes.

```
%Insert optimum after 150 steps
if bel(not(optimum(Opt)), tempOptimum(TOpt), agentRankRole(Rank, 'Explorer'), Rank=:=0, step(N),
      N>=150, calcOptimumZone(TOpt, OptZone)) then
 insert(optimum(TOpt), optZone(OptZone)) + send(allother, optimum(TOpt)) + send(allother, optZone(
      OptZone)).

%Determine what needs to be probed and broadcast to all.
if bel(optZone(O), not(exploreZone(E)), findall(Pos,(member(Pos,O),needProbe(Pos)),ExploreZone))
      then insert(exploreZone(ExploreZone)) + send(allother, exploreZone(ExploreZone)).

%If current explore-objective is completed, delete the corresponding belief
if bel(currentPos(Here), exploring(Here)) then delete(exploring(Here)).

%Pick a node to explore, then tell others not to explore it
if bel(not(exploring(_)),exploreZone(E), E \= [], agentRankRole(Rank,'Explorer'), nth0(Rank, E, Node))
      then insert(exploring(Node)) + send(allother, exploreRemove(Node)).
```

**Figure 6.10:** Explorers finding and maintaining optimum zone

The remaining agents use their knowledge of the optimum zone to swarm around the optimum. They use the original swarming algorithm of HactarV2 to position themselves, which allows us to evaluate how it performs in the 2012 setting.

CHAPTER 7

# Testing

In order to evaluate the performance of the improved HactarV2 system, we have tested its performance against the Python-DTU system. As explained earlier, we consider the Python-DTU system to be the best existing solution to the 2012 competition even though it got second place in the actual competition. We have run our program against Python-DTU 60 times, 20 times on each of the three different map sizes. Unfortunately, the node values are randomized each time so no test run can be repeated exactly, which is why we are looking at an average of the final scores instead.

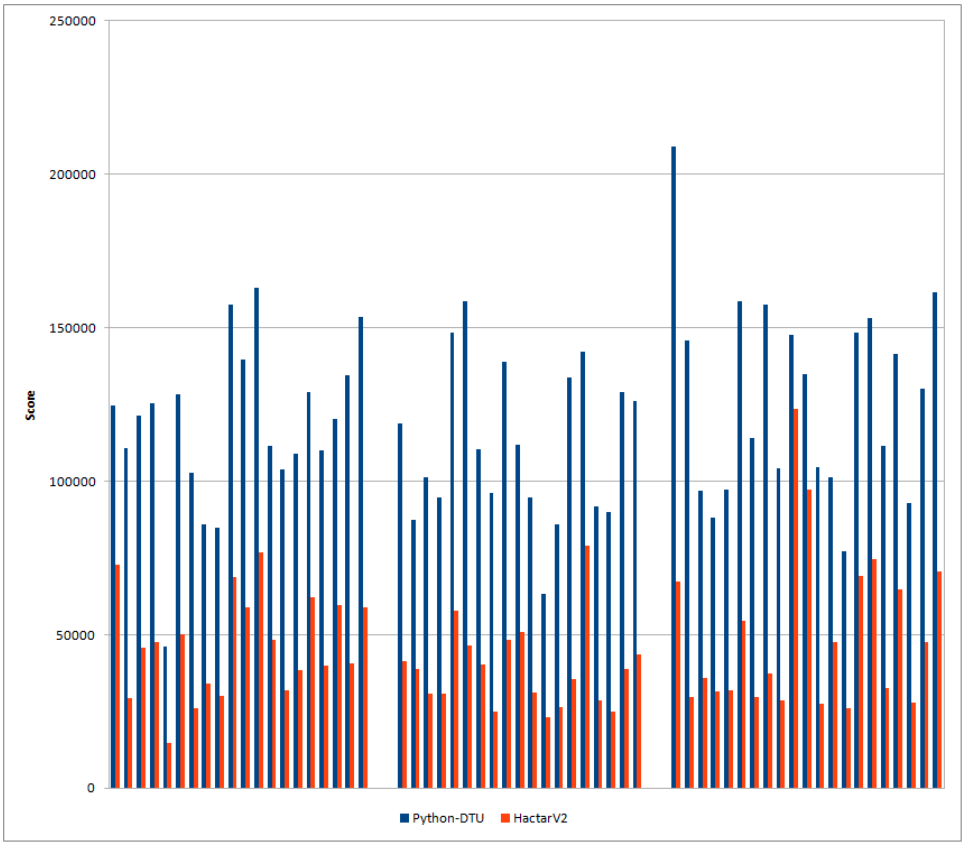


**Figure 7.1:** Comparison of scores for the two teams during 60 simulations

## 7.1 Analysis

Despite the high amount of variance in the test results, it is evident that Python-DTU is consistently outperforming our code on all three map sizes. We believe that this is because the core strategy of HactarV2 (see chapter 5) is inferior to that of Python-DTU.

### 7.1.1 Zone control

While reasonable, the swarming strategy of HactarV2 is held back by the fact that each agent is unable to predict the actions of the rest of its team. Communication between agents is only received at the start of each step, so the agents have to make

their decisions based on outdated information. This makes it very difficult to stay in formation around the optimum, as well as to distribute tasks between agents. Because the Python-DTU team coordinate all their actions rather than having each agent act autonomously, they do not suffer from this problem, and as a result they are much better at maintaining control of the most valuable areas and distributing their agents optimally across the map.

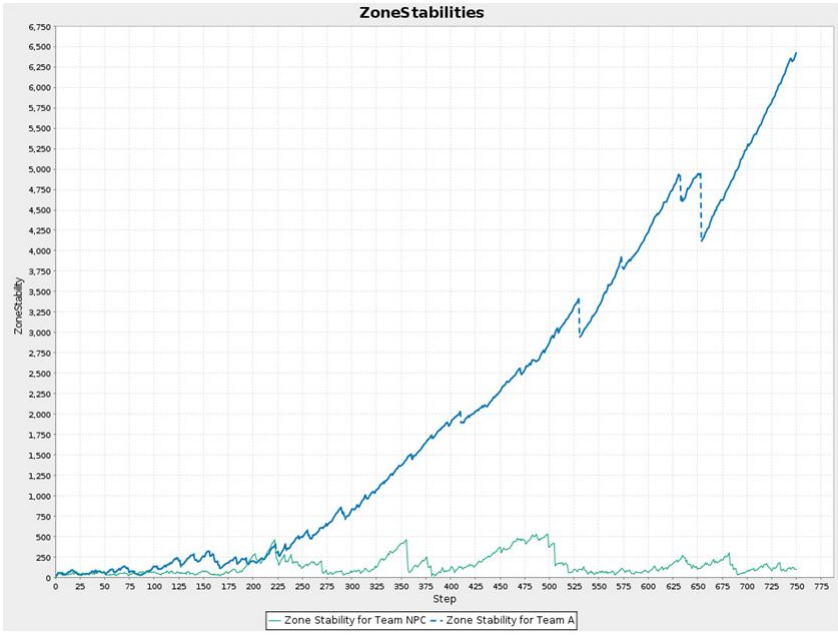


**Figure 7.2:** The Zone Stabilities of each team in each step of a sample run. The Zone Stability increases for one team, if the team can hold all conquered nodes over a longer period of time. If nodes are lost, the value decreases.

### 7.1.2 Buying Strategy

The buying strategy of HactarV2 is much more aggressive than that of Python-DTU. We spend all of our points early on, resulting in a deficit in achievement points that we never manage to recover from. This does mean that our agents are better upgraded than their opponents, so depending on the effectiveness of the rest of the strategy this may be an acceptable gambit, but it is worth noting that the amount of points lost is not insignificant, as can be seen below. It gives Python-DTU a significant advantage in terms of score, and because HactarV2 is significantly worse at zone control than Python-DTU, we are unable to capitalize on our upgrade advantage.

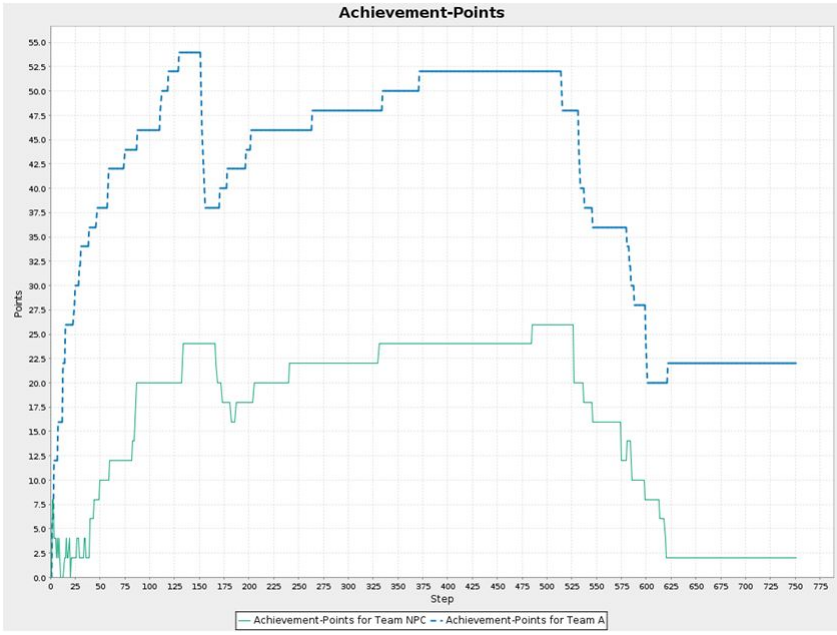


**Figure 7.3:** Step score gained from achievement points for the two teams during a sample run.

### 7.1.3 Pathing

When we initially tested our code, we found that it was often unable to respond to the server in time, so we eventually had to increase the maximum response time from 2 seconds to 10 seconds. This is due to the fact that the pathing algorithm used by HactarV2 is too inefficient to work properly in the 2012 setting. The HactarV2 system works by having each agent calculate a path to its target each round, so because the number of agents was increased from 10 to 20, the pathing algorithm of HactarV2 becomes too inefficient and is no longer able to keep up with the server.

## 7.2 In summation

We are forced to conclude that none of the core strategies and systems of HactarV2 are at the level of the Python-DTU system. Ultimately it is the inability of HactarV2's agents to coordinate their actions that prevents them from competing at the level of Python-DTU, as it renders them unable to take advantage of their aggressive buying

strategy. As a direct result of that, the team loses a large amount of points on a buying strategy they are unable to take advantage of, while at the same time being inferior to Python-DTU in terms of zone control. For the HactarV2 system to have a chance at winning, most or all of its core functionality would have to be replaced. In addition to that, the pathing algorithms would need to be optimized for the system to even meet the requirements of the original 2012 competition.

CHAPTER 8

# Evaluating GOAL

In this chapter, we evaluate the GOAL programming language based on our experience with it over the course of the project. We also briefly describe some of the more serious bugs we encountered.

## 8.1 The GOAL language

One of the strategies that GOAL adopts is the Intentional Stance described in Chapter 3. This strategy makes it very simple to understand what the agent programs are designed to do. In addition to this strategy, the GOAL designers have focused on making their agent programming language practical, transparent and useful[1]. As Figure 3.1 in chapter 3 shows, the structure of a GOAL program is intuitive and easy to understand, accomplishing the desire for transparency. If you pick a Knowledge Representation language you are already comfortable with, it is easy to get started programming your agents for whatever task they need to accomplish, making the language practical.

[1] [Hin11] page 11

## 8.2 Debugging in GOAL

We spent many hours debugging the HactarV2 code in GOAL. At the beginning of the project, we were not impressed with the debugging facilities of the language. We wanted a way to print debug information to the console, to follow the flow of the program. It was not possible to use the built-in *write/1* predicate of SWI-Prolog, and even if it had been, it was impossible to follow any information printed in the console, as it scrolled by too quickly.

We never used the feature of breakpoints and the ability to step into the code. The use of breakpoints is not described in the GOAL manual and the implementation is not intuitive enough that we felt we could utilize them.

As we learned more of the language, we found the logging to file feature very useful in debugging (see Figure 8.2). This allowed us to inspect an agent's behavior line by line. The log files get cluttered up with several lines of data printed for each reasoning cycle, and this can seem unnecessary when you only need one entry within a record tag, but with the search feature included in most text-editors, this is a minor problem compared to the level of detail the logs provide in terms of agent behavior. Of course, logging can only be used after the program has finished execution.

To debug at runtime, GOAL has a feature called the agent introspector, which allows you to query an agent according to its mental state as well as force the agent to take specific actions. The query function allows you to insert debug statements in the belief base at critical parts of the code and query the agent to find out whether or not it has reached the debug code. The introspector was acceptable, but only because no other runtime debugging was available to us. It requires you to write entire Prolog queries to determine if an agent has a specific belief (see Figure 8.1). It is possible to look directly at an agent's beliefs but since our agents constantly received percepts that caused them to update the belief base, it was usually impossible to learn anything from this. The possibility of pausing an agent was still not useful to us, as this meant it could not send actions to the server.

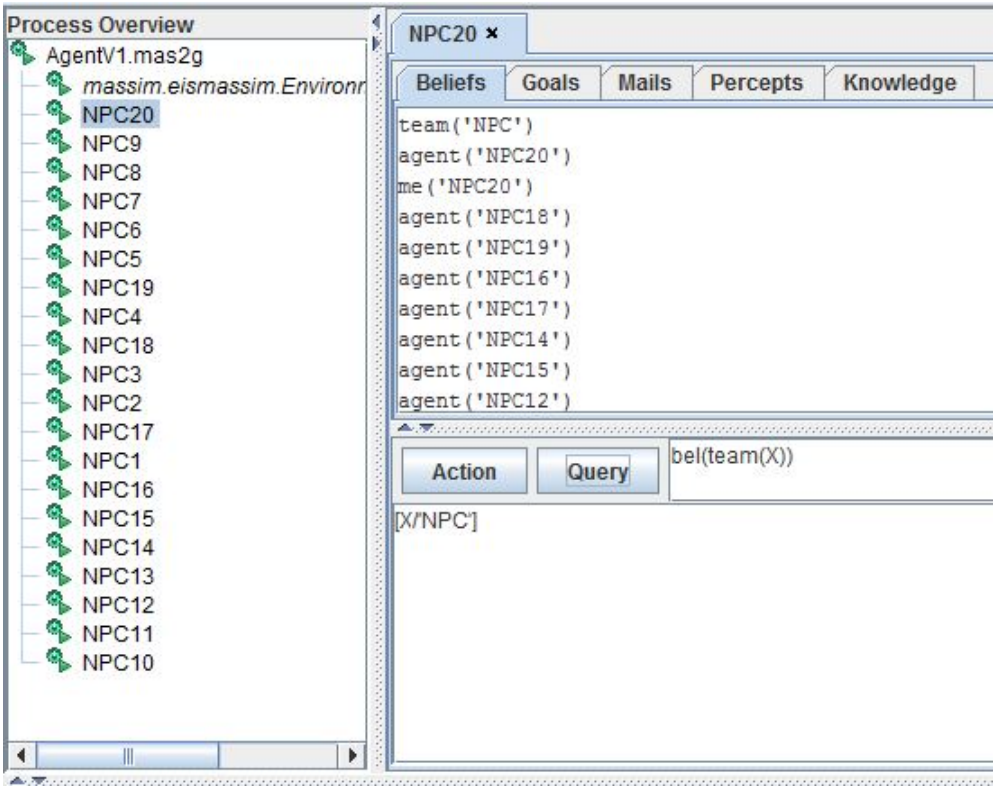


**Figure 8.1:** Introspector of an agent in GOAL with an answered query

## 8.3 GOAL bugs

While working with GOAL, we encountered several strange bugs which gave us a lot of problems in the first part of the project. After much testing we concluded that the GOAL versions released did not seem backwards compatible with older versions. This made it difficult to try out newer GOAL versions, as our program was not guaranteed to work as expected. Additionally, some of the more recent GOAL releases introduced new bugs that we could not find the source of.

The most problematic bug we encountered was that some modules created in GOAL were treated as action calls. Modules have the option of parameters, for passing variables to them. GOAL believed these modules with parameters to be action calls with uninstantiated variables, and stopped the agents trying to call them. As seen in line 9

in Figure 8.2, the module call *selectSurvey(Rank)* has an applicable instantiation, but instead of calling the module, GOAL just terminates the reasoning cycle because of the bug.

```
<record>
  <date>2013−03−10T13:18:55</date>
  <millis>1362917935659</millis>
  <sequence>11137</sequence>
  <level>INFO</level>
  <class>goal.tools.logging.GOALLogger</class>
  <method>log</method>
  <thread>31</thread>
  <message>Rule if bel(not(disabled) , currentPos(Here) , needSurvey(Here) , agentRankHere(Rank)) then
      selectSurvey(Rank)
 has 1 potentially applicable instantiation(s): [Here/vertex19, Rank/0]</message>
</record>
<record>
  <date>2013−03−10T13:18:57</date>
  <millis>1362917937425</millis>
  <sequence>15648</sequence>
  <level>INFO</level>
  <class>goal.tools.logging.GOALLogger</class>
  <method>log</method>
  <thread>31</thread>
  <message> +++++++ Cycle 2 +++++++ </message>
</record>
```

**Figure 8.2:** Module call interpreted as action ends the reasoning cycle

To avoid the bug we reverted to the latest working version available, GOAL release 4941. Even though this specific bug was later fixed, no bug free version has been released, and we have used version 4941 for the remainder of the project.

Another bug we encountered was that our agents would sometimes continue to receive the same percept each turn, even though that percept was only supposed to be sent once. For example, the *simStart* and *simEnd* percepts, which are supposed to be sent exactly once when the simulation starts or ends respectively, both continued to be sent to our agents every turn after the first time they were sent. This meant that our agents did not know when one simulation ended and the next began, which made it very difficult for us to test our program, since each test would have to be started and stopped manually. As far as we can tell it is a problem with either GOAL itself or the environment used in the competition, and so we could not do anything to fix it, but we eventually found a workaround that allowed us to test properly.

CHAPTER 9

# Conclusion

We have studied the MAS HactarV2 created for the 2011 MAPC. We have updated the system to run it on the 2012 version of the MAPC server. To test the merits of the HactarV2 team's strategy in the MAPC, we have pitted it against the Python-DTU team, which we believe to be the strongest team designed for the 2012 MAPC.

In working with the HactarV2 code, we have learned what defines a MAS and experienced its benefits as well as its disadvantages. Apart from studying the theory of Multi-agent Systems, we have earned this experience by working with the agent programming language GOAL, a language designed specifically for creating software agents and Multi-agent Systems.

The winners of the 2011 MAPC are no match for the DTU team. This is due both to the fact that the efficiency of their code is not optimized as well as the fact that their strategy is too dependent on the map structure from 2011 to be effective on the 2012 server. Our tests have shown they are inferior in every part of a match. An early aggressive buying strategy for saboteurs sees their achievement points plummet and they do not capitalize enough on their stronger agents to validate this choice. The later swarming strategy was efficient on the 2011 server but it is no longer useful to hold a fixed area around an optimum as a 2012 map has several local optima and node values are generated at random.

Another disadvantage HactarV2 has compared to Python-DTU is in communication and the degree of autonomy of the agents. When planning actions, the agents of Python-DTU use a shared environment to decide the best possible action for each agent at every step of a simulation. HactarV2's agents have a higher degree of autonomy; each agent decides its own actions and only relays its percepts to the other agents with a one step delay. This makes coordination a lot harder, and the team ends up being much less efficient, as HactarV2 agents perform actions based primarily upon what is most efficient from their own point of view, as opposed to the point of view of the entire team.

The GOAL programming language is very intuitive and straightforward to use. The GOAL IDE contains bugs and is missing some desirable features. As we have only worked with alpha releases, we expect that both the IDE and the language itself will be a much stronger tool in the future, as it is built on a promising design philosophy.

We have determined that even though the HactarV2 MAS was the best in 2011 it is too rigid and specialized to provide any challenge for systems developed for later versions of the MAPC. It is unfortunate we could not develop a MAS based on the HactarV2 team that was able to defeat the Python-DTU system, but since none of the core strategies and ideas of HactarV2 turned out to be suited for the 2012 scenario, we do not consider it worthwhile to use the HactarV2 code as the basis of a new solution.

In regard to evaluating GOAL, we think it would be interesting to compare it to an identical Multi-agent System implemented in a different programming language, in order to test whether or not the GOAL language offers improved performance as well as its easy understanding of an agent program.

APPENDIX A

# Changes to HactarV2

The specific changes required to adapt HactarV2 to the 2012 scenario are described in this section. For ease of reference, the changes will be broken down and described for each individual file.

## A.1 Agent.mas2g

The first file to change was the agent.mas2g file in the actual GOAL code. It is in this file that the connection to the environment is made and it is important that it reflects what the server expects. We made the following changes:

1. Changed team name to NPC
2. Added 10 agents for a total of 20
3. Changed agent names to reflect our new team name
4. Changed environment reference to reflect syntax change in GOAL
5. Changed detection of available entities from type to name

Some of these changes are self-explanatory. The fourth change refers to a syntax change in GOAL. The HactarV2 code was written for a 2011 version of GOAL and we are using a newer build. In the newer versions of GOAL the correct code is 'env = "path/to/environment/here"', whereas in older versions of GOAL you must put '"path/-to/environment/here"' when referencing the environment.

The fifth change was made because of the definition of type for entities in the mars 2012 scenario. In 2011 the type of an entity was *mars2011entity*. In 2012 it was not as simple as changing the year of the competition. The simple workaround for this was to have our system connect to individual entities based on their name, instead of their type. This works just fine but is not as convenient if the entities' names are changed. We decided find the root of the problem in the environment where the entities were handled.

It turned out that they had wanted more specific types for entities in the 2012 scenario. More specifically they wanted the type to include the role the entity was to play in the competition, for example *mars2012entityexplorer*. We discovered that this feature was not fully implemented. The environment never updated the role but instead it remained at its default, unknown value leaving all entities with the type *mars2012entityunknown*.

## A.2 Eismassimconfig.xml

With a new team name, as well as new names for our agents we had to ensure that the entities in the environment also had proper names in order for our agents to connect to them. We used the standard environment configuration file provided for the 2012 competition and modified it with the following changes:

1. Entity name to NPC#
2. Entity username to NPC#
3. Scenario name to mars2012
4. Scheduling disabled

Scheduling is disabled because the HactarV2 code implements its own scheduling.

## A.3 Accounts-NPC.xml

There are several files that provide information for the server setup, and one type of file is an account file that specifies a team of agents. We added an account file under 'massim-2012-2.0/massim/scripts/conf/helpers/2012' named accounts-NPC.xml, providing information about our team. We copied one of the other account files and changed the following:

1. Account team to NPC#
2. Account username to NPC#

## A.4 Config.dtd

This Document Type Definition[Wik13c] file specifies markup language entities that are used to reference account files in other documents. We added an entity 'teamNPC' that references our own account file.

## A.5 2012-3sims-NPC.xml

We added a server configuration file to allow our team to play any team with the team name 'A'. This file was added under 'massim-2012-2.0/massim/scripts/conf'.

Appendix B

# Code

## B.1 Original HactarV2 code

code/navigationKnowledge_original.pl

```
% Returns the rank(based on its name) of an agent compared to all other agents on its node
agentRankHere(Rank) :- currentPos(Here), me(Name), team(Team), !,
 setof(Agent, visibleEntity(Agent,Here,Team,normal), Agents), agentRank(Agents,Name,Rank).

%
agentRank(List,Agent,Rank) :- nth0(Rank, List, Agent), !.
```

## B.2 Final version source code

**Listing B.1:** code/actionProcessing.mod2g

```
module surveyVertices{
 program[order=linear]{
  % Search for and update current vertex.
  if bel(currentPos(Id1), !,vertex(Id1,Value,List),
     findall([W, Id2], (percept(surveyedEdge(Id1,Id2,W)); percept(surveyedEdge(Id2,Id1,W))), Array))
   then insert(not(vertex(Id1,Value,List)), vertex(Id1,Value,Array)) + send(allother,vertex(Id1,Value,Array)
        ).
  % Other statement is false. Do not search. Insert new vertex
  if bel(currentPos(Id1), !,
```

```
      findall([W, Id2], (percept(surveyedEdge(Id1,Id2,W)); percept(surveyedEdge(Id2,Id1,W))), Array))
    then insert(vertex(Id1,unknown,Array)) + send(allother,vertex(Id1,unknown,Array)).
  }
}

module probeVertices{
 program[order=linear]{
  % Search for and update current vertex.
  if bel(percept(probedVertex(Id1,Value)),vertex(Id1,V,List)) then
   insert(not(vertex(Id1,V,List)), vertex(Id1,Value,List))
   + send(allother,vertexProbed(Id1,Value)).
  % Other statement is false. Do not search. Insert new vertex.
  if bel(percept(probedVertex(Id1,Value))) then
   insert(vertex(Id1,Value,[])) + send(allother,vertexProbed(Id1,Value)).
 }
}

module inspectEntityPercept{
 program[order=linearall]{
  % When you get a percept of an inspected enemy, replace the last inspection of that entity and send the
      percept to all other agents.
  forall bel(percept(inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth,
      Strength, VisRange)), enemyTeam(Team),
    inspectedEntity(Id, Team, Role, V2, E2, ME2, H2, MH2, S2, VS2))
   do insert(not(inspectedEntity(Id, Team, Role, V2, E2, ME2, H2, MH2, S2, VS2)),
     inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth, Strength, VisRange
       ))
    + send(allother, inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth,
      Strength, VisRange)).

  % When you get a percept of an inspected enemy, and it has never been inspected before, insert it and
      send the percept to all other agents.
  forall bel(percept(inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth,
      Strength, VisRange)), enemyTeam(Team),
    not(inspectedEntity(Id, _, _, _, _, _, _, _, _, _)))
   do insert(inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth, Strength,
      VisRange))
    + send(allother, inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth,
      Strength, VisRange)).

  % Insert last time I inspected an agent.
  if bel(percept(inspectedEntity(Id,_,'Saboteur',_,_,_,_,_,_,_)), lastInspect(Id,LI), step(S)) then insert(not(
      lastInspect(Id,LI)), lastInspect(Id,S)).
  if bel(percept(inspectedEntity(Id,_,'Saboteur',_,_,_,_,_,_,_)), step(S)) then insert(lastInspect(Id,S)).
 }
}
```

**Listing B.2:** code/AgentV1.goal

```
init module {
 knowledge{
  % Contains general reasoning rules
  #import "generalKnowledge.pl".

  % Contains some rules that allow the agent to extract information from the percepts
```

```
7    #import "perceptKnowledge.pl".
8
9    % Contains role specific knowledge rules
10   #import "roleKnowledge.pl".
11
12   % Contains algorithms used for pathfinding
13   #import "dijkstra.pl".
14
15   % Contains rules about navigational subjects
16   #import "navigationKnowledge.pl".
17  }
18
19  beliefs{
20   % Makes sure the agent doesnt try to execute actions while the server is not started on startup
21   doneAction.
22   donePercepts.
23   doneMailing.
24
25   % Change this to our team name!
26   team('NPC').
27
28   ready.
29  }
30
31  goals{
32   % goals are dynamically inserted in the percept rules later on
33  }
34
35  % Define actions that can be sent to the environment interface
36  % Also specify what needs to be true in order to perform the actio
37  % and what should be inserted into the belief base afterwards
38  actionspec{
39   % Insert doneAction after each action to make sure no new actions are performed in this step(manual
         scheduling)
40   % All actions check if the agent meets the energy requirements, and for actions that require the agent to
         be enabled it will check if they are not disabled
41
42   recharge {
43    pre { true }
44    post { doneAction }
45   }
46   buy(Upgrade) {
47    pre { not(disabled), moneyGE(2), energyGE(2) }
48    post { doneAction }
49   }
50   probe {
51    pre { not(disabled), energyGE(1) }
52    post { doneAction }
53   }
54   parry {
55    pre { not(disabled), energyGE(2) }
56    post { doneAction }
57   }
58   survey {
59    pre { not(disabled), energyGE(1) }
```

```
60      post { doneAction }
61     }
62     % Only move over an edge when you actually have enough energy to do so
63     % Sometimes the edge you want to cross is not surveyed yet, but do make sure you try to move to a
           neighbour
64     goto(There) {
65      pre { currentPos(Here), shouldGoTo(Here,There) }
66      post{ doneAction }
67     }
68     skip {
69      pre { true }
70      post { doneAction }
71     }
72     % Only repair agents of the same team, on your location, and not yourself!
73     repair(Agent) {
74      pre { energyGE(3), currentPos(Here), team(Team), me(Me), visibleEntity(Agent, Here, Team, _), not(
            Agent == Me) }
75      post { doneAction }
76     }
77     % Only attack enemies on your location. Keep track of who you last attacked for strategic purposes
78     attack(Agent) {
79      pre { not(disabled), energyGE(2), currentPos(Here), visibleEntity(Agent, Here, Team, _), enemyTeam(
            Team), lastAttacked(X) }
80      post { not(lastAttacked(X)), lastAttacked(Agent), doneAction }
81     }
82     inspect {
83      pre { not(disabled), energyGE(2) }
84      post { doneAction }
85     }
86    }
87   }
88
89   % Main module which is executed every cycle, rules are considered linearly by default
90   main module{
91    program[order=linearall] {
92     % Only try to find a new action when one was not chosen in this step yet
93     if bel(not(doneAction)) then {
94
95      % If disabled get yourself fixed as soon as possible
96      if bel(disabled, not(role('Repairer'))) then disabled.
97
98      % Otherwise enter your role specific module to do something useful with your role
99      if bel(role('Repairer')) then repairerAction.
100     if bel(role('Inspector')) then inspectorAction.
101     if bel(role('Explorer')) then explorerAction.
102     if bel(role('Saboteur')) then saboteurAction.
103     if bel(role('Sentinel')) then sentinelAction.
104
105     % Aparently you had nothing role specific to do, so do some exploring
106     if bel(true) then explore.
107
108     % If no action could be found just send a skip to 'no valid action received in time'
109     if bel(true) then skip.
110    }
111   }
```

```
112 }
113
114 % Importing all the modules that are used for choosing an action
115
116 % This module contains behavior for when we have (almost) the entire map
117 #import "superiority.mod2g".
118 % This is a module that contains common behavior that each agent should perform
119 #import "common.mod2g".
120 % This module contains role specific behavior for the explorers
121 #import "explorer.mod2g".
122 % This module contains role specific behavior for the saboteurs
123 #import "saboteur.mod2g".
124 % This module contains role specific behavior for the repairers
125 #import "repairer.mod2g".
126 % This module contains role specific behavior for the sentinels
127 #import "sentinel.mod2g".
128 % This module contains role specific behavior for the inspectors
129 #import "inspector.mod2g".
130 % This module contains general behavior for disabled agents, but not repairers
131 #import "disabled.mod2g".
132 % This module contains behavior for agents that have to swarm to secure an area
133 #import "swarm.mod2g".
134 % This module contains some administrative rules that have to be performed after specific actions
135 #import "actionProcessing.mod2g".
136 % This module contains rules that allow for pathfinding and moving
137 #import "pathing.mod2g".
138 % This module contains rules required by an agent to defend itself in times of danger
139 #import "defense.mod2g".
140
141 % Event module which is called every GOAL cycle and is used for handling percepts, as well as updating
        the belief and goal base before an action is selected
142 event module{
143  program[order=linearall]{
144    % When a new step is detected allow the program to process the percepts, mails from other agents and
           choose a new action
145    if bel(percept(step(Current)), step(Old), !, Old \= Current) then {
146     if bel(Old == unknown)
147      then insert(not(step(Old)), not(donePercepts), not(doneMailing), not(doneAction), step(Current)).
148     %the integer part is to keep unknown from getting in the arithmetic.. should be catched by the rule
           above but sometimes isn't
149     if bel((integer(Old), Current >= Old + 1))% =:= changed to >=
150      then insert(not(step(Old)), not(donePercepts), not(doneMailing), not(doneAction), step(Current)).
151    }
152
153    % if the percepts and mails are not handled do so, and make sure it doesn't happen again before the next
           step
154    if bel(not(donePercepts)) then selectPercepts + insert(donePercepts).
155    if bel(donePercepts, not(doneMailing)) then selectReceiveMail + insert(doneMailing).
156
157    % simStart percepted and ready, handle the simStartpercepts and allow the program to send actions again
158    if bel(percept(simStart), ready) then delete(ready) + simStartPercepts.
159  }
160 }
```

**Listing B.3:** code/AgentV1.mas2g

```
% the agent teams mas2g file
% this file contains several parameters required for launching the GOAL agent team

environment {
 "eismassim-2.0.jar".
}

agentfiles {
 "AgentV1.goal" [name=mapc].
}

launchpolicy{
% Launch all the agents with a type corresponding to the one they have in the simulation
 when [type=mars2012entityunknown,max=1]@env do launch NPC1:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC2:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC3:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC4:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC5:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC6:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC7:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC8:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC9:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC10:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC11:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC12:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC13:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC14:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC15:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC16:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC17:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC18:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC19:mapc.
 when [type=mars2012entityunknown,max=1]@env do launch NPC20:mapc.
}
```

**Listing B.4:** code/common.mod2g

```
% Makes sure agents process percepts that are relevant to their role
module selectPercepts{
 program[order=linearall]{
  % Handle percepts that everyone uses.
  if true then commonPercepts.

  % Handle percepts that are specific to actions
  if bel(lastAction(survey), lastActionResult(successful)) then surveyVertices.

  % Handle percepts specific for your role.
  if bel(role('Explorer')) then explorerPercepts.
  if bel(role('Saboteur')) then saboteurPercepts.
  if bel(role('Repairer')) then repairerPercepts.
  if bel(role('Inspector')) then inspectorPercepts.
  if bel(role('Sentinel')) then sentinelPercepts.
 }
}

```

```
% Makes sure agents process mail that is relevant to their role
module selectReceiveMail{
 program[order=linearall]{
  % Handle mails that everyone uses.
  if true then commonReceiveMail.

  % Handle mails specific for your role.
  if bel(role('Explorer')) then explorerReceiveMail.
  if bel(role('Saboteur')) then saboteurReceiveMail.
  if bel(role('Repairer')) then repairerReceiveMail.
  if bel(role('Inspector')) then inspectorReceiveMail.
  if bel(role('Sentinel')) then sentinelReceiveMail.

  % Handle mails that disabled agents need.
  if bel(disabled) then disabledReceiveMail.

  % Clean up mailbox.
  if true then clearMailbox.
 }
}

% Module that performs some initial percept handling and allows the agent to start sending actions
module simStartPercepts{
 program [order=linearall] {
  % Insert some dummy values for certain predicates, to allow updating them
  if true then insert(oldZone(unknown), lastPos(unknown), step(unknown), optimumInfo([], [], []),
        currentPos(unknown), zoneScore(unknown), health(unknown), optimumValue(5)).

      % Tell the others your role
  if bel( percept(role(R)), me(Id), not(role(Id, _)) )
   then insert(role(Id, R)) + send(allother,role(R)).

  % Insert some info about the match and the map
  if bel( percept(steps(X)) ) then insert(steps(X)).
  if bel( percept(edges(X)) ) then insert(edges(X)).
  if bel( percept(vertices(X))) then insert(vertices(X)).

  % Dummyvalue for lastattacked for saboteur
  if bel(role('Saboteur')) then insert(lastAttacked('')).

  % Explore should have a goal to find an optimal node
  if bel(role('Explorer'))
   then adopt(optimum).


 }
}

% Module that can be called to reset the agent to a clean state ready to start a new match
module resetBeliefs{
 program[order=linearall]{
  % Delete some role specific information(deleting takes a bit of time, hence the role check)
  if bel(role('Saboteur'), lastAttacked(X)) then delete(lastAttacked(X)).
  if bel(role('Inspector')) then {
   forall bel(lastInspect(Id, X)) do delete(lastInspect(Id, X)).
```

```
74      }
75
76      if bel(role('Repairer'), repairing(X)) then delete(repairing(X)).
77
78      % Throw out information from the previous match
79      if bel( health(H) ) then delete( health(H) ).
80      if bel(steps(X)) then delete(steps(X)).
81      if bel(vertices(X)) then delete(vertices(X)).
82      if bel(optimumInfo(X,Y,Z)) then delete(optimumInfo(X,Y,Z)).
83      if bel(edges(X)) then delete(edges(X)).
84      if bel( optimum(X) ) then delete( optimum(X) ).
85
86      % Forget your mates roles(in case of a new random assignment)
87      forall bel(role(Id, Role)) do delete(role(Id, Role)).
88
89      % More garbage deleting
90      forall bel(enemyStatus(Id, Vertex, State)) do delete(enemyStatus(Id, Vertex, State)).
91      if bel(currentPos(X)) then delete(currentPos(X)).
92      if bel(lastPos(X)) then delete(lastPos(X)).
93      if bel(step(X)) then insert(not(step(X))).
94      if bel(zoneScore(X)) then delete(zoneScore(X)).
95      if bel(oldZone(X)) then delete(oldZone(X)).
96      forall bel( vertex(Id, Value, List) ) do delete( vertex(Id, Value, List) ).
97      forall bel(repairerLoc(Agent,Loc)) do delete(repairerLoc(Agent,Loc)).
98      forall bel(inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth, Strength,
            VisRange))
99       do delete(inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth, Strength,
             VisRange)).
100
101
102     % After deleting all garbage make sure no new actions are sent, and the agent is ready for a new simstart
103     if true then insert(donePercepts, doneMailing, doneAction, ready).
104   }
105 }
106
107 % Module that processes percepts that are received from the environment
108 module commonPercepts{
109   program[order=linearall]{
110     %Keep track of zoneScore
111     if bel(percept(zoneScore(Z)), zoneScore(X), oldZone(Y)) then insert( not(zoneScore(X)), not(oldZone(Y
            )), oldZone(X), zoneScore(Z) ).
112
113     % Keep track of the vertex you were on before you got here.
114     if bel(percept(position(Cur)), currentPos(Old), !, Old \= Cur) then {
115       if bel(lastPos(P)) then insert (not(lastPos(P)), lastPos(Old)).
116     }
117
118     % Update current location
119     if bel(percept(position(Cur)), currentPos(Old))
120       then insert(not(currentPos(Old)), currentPos(Cur)).
121
122     % Updates the info about the optimum swarm
123     if bel(optimumInfo(A,B,C), allInformationOptimumZone(Agents, Vertices, FirstDegree))
124       then insert(not(optimumInfo(A,B,C)), optimumInfo(Agents, Vertices, FirstDegree)).
125
```

```
126     % Swarm goal managing, when in optimum zone your goal is to swarm along
127     if not(goal(swarm)), bel(inOptimumZone) then adopt(swarm).
128     if goal(swarm), bel(not(inOptimumZone)) then drop(swarm).
129
130     % Check if the found optimum wasn't wrong
131     if bel(optimum(O), currentPos(Here), vertex(Here, Value, _), vertex(O, OValue, _), Value \= unknown,
              Value > OValue, optimumZone(OldZone), calcOptimumZone(Here,OZone), findall(Pos,(member(
              Pos,OZone),needProbe(Pos)),ExploreZone))
132      then insert(not(optimum(O)), optimum(Here), not(optimumZone(OldZone)), optimumZone(OZone)) +
               send(allother, optimum(Here)) + send(allother,optimumZone(OZone)) + send(allother,
               exploreZone(ExploreZone)).
133
134     % Request repairs when broken and notify repairer when fixed, also update the agents health
135     if bel(percept(health(H)), health(Current), !, H \= Current) then {
136      if bel( H == 0 ) then insert(not(health(Current)), health(H)).
137      forall bel( Current == 0, role(Agent,'Repairer') ) do insert(not(health(Current)), health(H)) + send(
               Agent,fixed).
138      if true then insert(not(health(Current)), health(H)).
139     }
140
141     % Keep track of the status of enemy agents, disabled agents are not scary
142     if bel(enemyStatus(ID,StoredVertex,StoredStatus), visibleEntity(ID,ActualVertex,_,ActualStatus))
143      then insert(not(enemyStatus(ID,StoredVertex,StoredStatus)), enemyStatus(ID,ActualVertex,
               ActualStatus)).
144
145     % If you find an enemy send that a saboteur can't see himself, inform the saboteur of the enemy and it's
              location
146     forall bel( visibleEntity(EID,Vertex,Team,X), enemyTeam(Team), role(ID,'Saboteur'), not(visibleEntity(
              ID,_,_,_)) )
147      do send(ID,visibleEntity(EID,Vertex,Team,X)).
148
149     % The enemy is no longer at the reported location, make sure the saboteur does not have incorrect info
150     forall bel( percept(visibleVertex(Vertex,Team)), enemyStatus(EID,Vertex,X), enemyTeam(Team), not(
              visibleEntity(EID,_,_,_)),
151        role(ID,'Saboteur'), not(visibleEntity(ID,_,_,_)) )
152      do send(ID,not(visibleEntity(EID,Vertex,Team,X))).
153    }
154   }
155
156   % Module that processes messages from other agents
157   module commonReceiveMail{
158    program[order=linearall]{
159     % Update edge/node values for (non)existing vertices
160     forall bel(received(A,vertex(Id,Value,NewList)), not(vertex(Id,_,_))) do
161      delete(received(A,vertex(Id,Value,NewList))) + insert(vertex(Id,Value,NewList)).
162     forall bel(received(A,vertex(Id,Value,NewList)), vertex(Id,Value,OldList)) do
163      delete(received(A,vertex(Id,Value,NewList))) + insert(not(vertex(Id,Value,OldList)), vertex(Id,Value,
               NewList)).
164
165     % Update probe values for (non)existing vertices
166     forall bel(received(A,vertexProbed(Id,Value)), not(vertex(Id,_,_)), not(role('Explorer')))
167      do delete(received(A,vertexProbed(Id,Value))) + insert(vertex(Id,Value,[])).
168     forall bel(received(A,vertexProbed(Id,Value)), vertex(Id,unknown,List), not(role('Explorer')))
169      do delete(received(A,vertexProbed(Id,Value))) + insert(not(vertex(Id,unknown,List)), vertex(Id,Value,
               List)).
```

```
170
171     % Optimum location receiving
172     if bel(received(Agent, optimum(Opt)), optimumValue(OldVal), vertexValue(Opt,NewVal), not(role('
            Explorer'))) then {
173       if bel(optimum(O)) then insert(not(optimum(O)), optimum(Opt), not(optimumValue(OldVal)),
              optimumValue(NewVal)) + delete(received(Agent, optimum(Opt))).
174       if true then insert(optimum(Opt), not(optimumValue(OldVal)), optimumValue(NewVal)) + delete(
              received(Agent, optimum(Opt))).
175     }
176
177     % Receiving data about the optimum zone
178     if bel(received(Agent, optZone(OptZone))) then {
179       if bel(optZone(O)) then insert(not(optZone(O)), optZone(OptZone)) + delete(received(Agent, optZone(
              OptZone))).
180       if true then insert(optZone(OptZone)) + delete(received(Agent, optZone(OptZone))).
181     }
182
183     % Agent roles
184     forall bel(received(Agent,role(Role))) do insert(role(Agent,Role)) + delete(received(Agent,role(Role))).
185
186     % inspectedEntities
187     % When you get a percept of an inspected enemy, replace the last inspection of that entity.
188     forall bel(received(_, inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth,
              Strength, VisRange)),
189       inspectedEntity(Id, Team, Role, V2, E2, ME2, H2, MH2, S2, VS2))
190       do insert(not(inspectedEntity(Id, Team, Role, V2, E2, ME2, H2, MH2, S2, VS2)),
191       inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth, Strength, VisRange))
                .
192     % When you get a percept of an inspected enemy, and it has never been inspected before, insert it.
193     forall bel(received(_, inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth,
              Strength, VisRange)),
194       not(inspectedEntity(Id, _, _, _, _, _, _, _, _)))
195       do insert(inspectedEntity(Id, Team, Role, Vertex, Energy, MaxEnergy, Health, MaxHealth, Strength,
              VisRange)).
196
197     % When send location info about enemies save it.
198     forall bel(received(Sender,visibleEntity(EID,Vertex,Team,X)),enemyStatus(EID,OVertex,Y))
199       do delete(received(Sender,visibleEntity(EID,Vertex,Team,X))) +
200       insert(not(enemyStatus(EID,OVertex,Y)), enemyStatus(EID,Vertex,X)).
201
202     forall bel(received(Sender,visibleEntity(EID,Vertex,Team,X)),not(enemyStatus(EID,_,_)))
203       do delete(received(Sender,visibleEntity(EID,Vertex,Team,X))) +
204       insert(enemyStatus(EID,Vertex,X)).
205
206     % Disapearing enemies should be removed so we don't go on wild chases.
207     forall bel( received(Sender, not(visibleEntity(EID,Vertex,Team,X))),enemyStatus(EID,Vertex,Y) )
208       do delete( received(Sender, not(visibleEntity(EID,Vertex,Team,X))), enemyStatus(EID,Vertex,Y) ).
209   }
210 }
211
212 % Clears out received messages and sent messages, these are now processed and irrelevant, hence slowing
        down the queries for no reason
213 module clearMailbox{
214   program[order=linearall]{
215     forall bel(received(Agent,Message)) do delete(received(Agent,Message)).
```

```
    forall bel(sent(Agent,Message)) do delete(sent(Agent,Message)).
  }
}

% The common explore module that works for every agent and explores the graph and its edges
module explore {
  program {
    % if there are edges with unknown weight around the current node survey them
    if bel(currentPos(Here), !, needSurvey(Here), agentRankHere(Rank))
      then selectSurvey(Rank).

    % Find closest unsurveyed vertex
    if bel(foreverAlone, currentPos(Start), pathClosestNonSurveyed(Start, NonSurveyedVertex, [Here,Next|
        Path], Dist))
      then advancedGoto(Next).

    % When multiple agents are on the node and there is an unsurveyed neighbour, try to split up.
    if bel(not(foreverAlone), agentRankHere(Rank), neighbourNeedSurvey(Any)) then gotoNeighbour(Rank
        , true, false).

    % find a better(higher value) node to chill on
    if bel(currentPos(Here), !, neighbour(There), safePos(There),
      vertexValue(Here,Value1), vertexValue(There,Value2), Value2 >= Value1)
      then advancedGoto(There).

    % lack of better node, go to an unprobed one.
    if bel(neighbour(There), vertexValue(There,unknown), safePos(There))
      then advancedGoto(There).

    % find a safe place to stand
    if bel(neighbour(There), safePos(There))
      then advancedGoto(There).

    % keep moving
    if bel(currentPos(Here), not(safePos(Here)), neighbour(Here, There))
      then advancedGoto(There).

    if bel(neighbour(There))
      then advancedGoto(There).
  }
}
```

**Listing B.5:** code/defense.mod2g

```
module defense{
  program{
    % Enemy on your position and the agent can parry
    if bel(not(role('Explorer')), not(role('Inspector')), needToParry) then defenseParry.

    % Wait for the saboteur to beat the shit out of you for parry achievements
    if bel(not(role('Explorer')), not(role('Inspector')), maxEnergy(E), not(energy(E)), !, neighbour(There),
      visibleEntity(Id, There, Team, _), enemyTeam(Team), inspectedEnemy(Id, 'Saboteur')) then recharge.

    % If you cant parry then just run away
    if true then defenseFlee.
```

```
12    }
13   }
14
15   module defenseParry{
16    program{
17     % randomly pick flee or parry when last parry was useless.
18     if bel(lastAction(parry), lastActionResult(useless)) then randomDefense.
19     if true then parry.
20     if true then recharge.
21    }
22   }
23
24   module randomDefense{
25    program{
26     % Keep 50% chance to Parry or Flee, not 33% chance to parry, recharge or continue role.
27     if bel(random(0.0, 1.0, R), R > 0.5) then {
28      if true then parry.
29      if true then recharge.
30     }
31     if true then defenseFlee.
32    }
33   }
34
35   module defenseFlee{
36    program{
37     if goal(swarm), bel(team(Team)) then {
38      % run away if needed.
39      if bel( currentPos(Here), not(needSurvey(Here))) then {
40       % try to move to the edge safely
41       if bel( safeEdgeDest(List), agentRankHere(Rank) ) then moveSplit(Rank, List).
42
43       % try to expand safely
44       if bel( safeExpandDest(List), agentRankHere(Rank) ) then gotoSplit(Rank, List).
45
46       % otherwise just move to a safe spot, probably collapsing the swarm
47       if bel( neighbour(N), vertexOwner(N,Team), safePos(N)) then advancedGoto(N).
48      }
49
50      % max edge Weight is 9.
51      if bel( energyGE(9), currentPos(Here) ) then {
52       % to a safe spot.
53       if bel( visibleEdge(Here,N), vertexOwner(N,Team), safePos(N)) then advancedGoto(N).
54       % to a safer spot which isn't where I was last step.
55       if bel( visibleEdge(Here,N), vertexOwner(N,Team), not((visibleEntity(_, N, ETeam, _), enemyTeam(
              ETeam))), not(lastPos(N))) then advancedGoto(N).
56       % to a safer spot.
57       if bel( visibleEdge(Here,N), vertexOwner(N,Team), not((visibleEntity(_, N, ETeam, _), enemyTeam(
              ETeam)))) then advancedGoto(N).
58      }
59     }
60
61    % run away if needed.
62    if bel( currentPos(Here), not(needSurvey(Here))) then {
63     % to a safe spot.
64     if bel( neighbour(N), safePos(N)) then advancedGoto(N).
```

```
65      % to a safer spot which isn't where I was last step.
66      if bel( neighbour(N), not((visibleEntity(_, N, Team, _), enemyTeam(Team))), not(lastPos(N))) then
            advancedGoto(N).
67      % to a safer spot.
68      if bel( neighbour(N), not((visibleEntity(_, N, Team, _), enemyTeam(Team)))) then advancedGoto(N).
69    }
70
71    % max edge Weight is 9.
72    if bel( energyGE(9), currentPos(Here) ) then {
73      % to a safe spot.
74      if bel( visibleEdge(Here,N), safePos(N)) then advancedGoto(N).
75      % to a safer spot which isn't where I was last step.
76      if bel( visibleEdge(Here,N), not((visibleEntity(_, N, Team, _), enemyTeam(Team))), not(lastPos(N)))
            then advancedGoto(N).
77      % to a safer spot.
78      if bel( visibleEdge(Here,N), not((visibleEntity(_, N, Team, _), enemyTeam(Team)))) then
            advancedGoto(N).
79    }
80
81    if true then recharge.
82   }
83 }
```

**Listing B.6:** code/dijkstra.pl

```
1  % Dijkstra from S to T
2  % Code is adapted from: http://colin.barker.pagesperso-orange.fr/lpa/dijkstra.htm
3
4  % main predicate
5
6  % path(Vertex0, Vertex, Path, Dist) is true if Path is the shortest path from
7  % Vertex0 to Vertex, and the length of the path is Dist. The graph is defined
8  % by e/3.
9  % e.g. path(penzance, london, Path, Dist)
10 path(Start, Target, Path, Dist) :-
11   dijkstra2(Start, Target, s(Target,Dist,Path)), !.
12
13 % helping predicates
14
15 dijkstra2(Start, Target, ResultingS):-
16   create(Start, [Start], Ds),
17   recharge(ERecharge),
18   dijkstra_2(Ds, ERecharge, [s(Start,0,[])], Target, ResultingS).
19
20 dijkstra_2([], _, _, _, _) :- !, fail.
21 dijkstra_2([D|Ds], ERecharge, _, Target, s(Target,Distance2,Path1)):-
22   best(Ds,D,s(Target,Distance,Path)),
23   delete2([D|Ds], [s(Target,Distance,Path)], _),
24   reverse([Target|Path], Path1),
25   Distance2 is Distance + ERecharge, !. % the '!' makes sure only 1 solution (the shortest) is correct.
26
27 dijkstra_2([D|Ds], ERecharge, Ss0, Target, ResultingS):-
28   best(Ds, D, S),
29   delete2([D|Ds], [S], Ds1),
30   S=s(Vertex,Distance,Path),
```

```
31    reverse([Vertex|Path], Path1),
32    Distance2 is Distance + ERecharge,
33    merge2(Ss0, [s(Vertex,Distance2,Path1)], Ss1),
34    create(Vertex, [Vertex|Path], Ds2),
35    delete2(Ds2, Ss1, Ds3),
36    incr(Ds3, Distance2, Ds4),
37    merge2(Ds1, Ds4, Ds5),
38    dijkstra_2(Ds5, ERecharge, Ss1, Target, ResultingS).
39
40  % Algorithm for calculating optimum zone
41
42  calcOptimumZone(Opt, OptZone) :-
43   neighbour(Opt,Primary),
44   findall(X,(member(Y,Primary),neighbour(X,Y),not(member(X,Primary))),Secondary),
45   merge(Primary,Secondary,OptZone).
46
47  % Dijkstra for closest non-probed vertex
48  %% Code is adapted from: http://colin.barker.pagesperso-orange.fr/lpa/dijkstra.htm
49
50  % main predicate
51
52  pathClosestNonProbed(Start, NonProbedVertex, Path, Dist) :-
53    dijkstra3(Start, s(NonProbedVertex, Dist, Path)), !.
54
55  % helping predicates
56
57  dijkstra3(Start, ResultingS):-
58    create(Start, [Start], Ds),
59    recharge(ERecharge),
60    dijkstra_3(Ds, ERecharge, [s(Start,0,[])], ResultingS).
61
62  dijkstra_3([], _, _, _) :- !, fail.
63  dijkstra_3([D|Ds], ERecharge, _, s(Vertex,Distance2,Path1)):-
64    best(Ds,D,s(Vertex,Distance,Path)),
65    needProbe(Vertex),
66    delete2([D|Ds], [s(Vertex,Distance,Path)], _),
67    reverse([Vertex|Path], Path1),
68    Distance2 is Distance + ERecharge, !.
69
70  dijkstra_3([D|Ds], ERecharge, Ss0, ResultingS):-
71    best(Ds, D, S),
72    delete2([D|Ds], [S], Ds1),
73    S=s(Vertex,Distance,Path),
74    reverse([Vertex|Path], Path1),
75    Distance2 is Distance + ERecharge,
76    merge2(Ss0, [s(Vertex,Distance2,Path1)], Ss1),
77    create(Vertex, [Vertex|Path], Ds2),
78    delete2(Ds2, Ss1, Ds3),
79    incr(Ds3, Distance2, Ds4),
80    merge2(Ds1, Ds4, Ds5),
81    dijkstra_3(Ds5, ERecharge, Ss1, ResultingS).
82
83  % Dijkstra for closest non-probed vertex, with some additional checks
84  % Code is adapted from: http://colin.barker.pagesperso-orange.fr/lpa/dijkstra.htm
85
```

```
86  % main predicate
87
88  pathClosestNonProbedWithExtraChecks(Start, NonProbedVertex, Path, Dist) :-
89    dijkstra9(Start, s(NonProbedVertex, Dist, Path)), !.
90
91  % helping predicates
92
93  dijkstra9(Start, ResultingS):-
94    create(Start, [Start], Ds),
95    recharge(ERecharge),
96    dijkstra_9(Ds, ERecharge, [s(Start,0,[])], ResultingS).
97
98  dijkstra_9([], _, _, _) :- !, fail.
99  dijkstra_9([D|Ds], ERecharge, _, s(Vertex,Distance2,Path1)):-
100   best(Ds,D,s(Vertex,Distance,Path)),
101   needProbe(Vertex),
102   e(Vertex, NeedExploring, _), team(Team), vertexOwner(NeedExploring, Team),
103   delete2([D|Ds], [s(Vertex,Distance,Path)], _),
104   reverse([Vertex|Path], Path1),
105   Distance2 is Distance + ERecharge.
106
107 dijkstra_9([D|Ds], ERecharge, Ss0, ResultingS):-
108   best(Ds, D, S),
109   delete2([D|Ds], [S], Ds1),
110   S=s(Vertex,Distance,Path),
111   reverse([Vertex|Path], Path1),
112   Distance2 is Distance + ERecharge,
113   merge2(Ss0, [s(Vertex,Distance2,Path1)], Ss1),
114   create(Vertex, [Vertex|Path], Ds2),
115   delete2(Ds2, Ss1, Ds3),
116   incr(Ds3, Distance2, Ds4),
117   merge2(Ds1, Ds4, Ds5),
118   dijkstra_9(Ds5, ERecharge, Ss1, ResultingS).
119
120 % Dijkstra for closest non-surveyed vertex
121 % Code is adapted from: http://colin.barker.pagesperso-orange.fr/lpa/dijkstra.htm
122
123 % main predicate
124
125 pathClosestNonSurveyed(Start, NonSurveyedVertex, Path, Dist) :-
126   dijkstra4(Start, s(NonSurveyedVertex, Dist, Path)), !.
127
128 % helping predicates
129
130 dijkstra4(Start, ResultingS):-
131   create(Start, [Start], Ds),
132   recharge(ERecharge),
133   dijkstra_4(Ds, ERecharge, [s(Start,0,[])], ResultingS).
134
135 dijkstra_4([], _, _, _) :- !, fail.
136 dijkstra_4([D|Ds], ERecharge, _, s(Vertex,Distance2,Path1)):-
137   best(Ds,D,s(Vertex,Distance,Path)),
138   needSurvey(Vertex),
139   delete2([D|Ds], [s(Vertex,Distance,Path)], _),
140   reverse([Vertex|Path], Path1),
```

```
141     Distance2 is Distance + ERecharge, !. % the '!' makes sure only 1 solution (the shortest) is correct.
142
143 dijkstra_4([D|Ds], ERecharge, Ss0, ResultingS):-
144     best(Ds, D, S),
145     delete2([D|Ds], [S], Ds1),
146     S=s(Vertex,Distance,Path),
147     reverse([Vertex|Path], Path1),
148     Distance2 is Distance + ERecharge,
149     merge2(Ss0, [s(Vertex,Distance2,Path1)], Ss1),
150     create(Vertex, [Vertex|Path], Ds2),
151     delete2(Ds2, Ss1, Ds3),
152     incr(Ds3, Distance2, Ds4),
153     merge2(Ds1, Ds4, Ds5),
154     dijkstra_4(Ds5, ERecharge, Ss1, ResultingS).
155
156 % Dijkstra for closest Repairer
157 % Code is adapted from: http://colin.barker.pagesperso-orange.fr/lpa/dijkstra.htm
158
159 % main predicate
160
161 pathClosestRepairer(Start, LocationRepairer, NameAgent, Path, Dist) :-
162     dijkstra5(Start, s(LocationRepairer, Dist, Path), NameAgent), !.
163
164 % helping predicates
165
166 dijkstra5(Start, ResultingS, NameAgent):-
167     create(Start, [Start], Ds),
168     recharge(ERecharge),
169     dijkstra_5(Ds, ERecharge, [s(Start,0,[])], ResultingS, NameAgent).
170
171 dijkstra_5([], _, _, _, _) :- !, fail.
172 dijkstra_5([D|Ds], ERecharge, _, s(Vertex,Distance2,Path1), NameAgent):-
173     best(Ds,D,s(Vertex,Distance,Path)),
174     repairerLoc(NameAgent, Vertex),
175     delete2([D|Ds], [s(Vertex,Distance,Path)], _),
176     reverse([Vertex|Path], Path1),
177     Distance2 is Distance + ERecharge, !. % the '!' makes sure only 1 solution (the shortest) is correct.
178
179 dijkstra_5([D|Ds], ERecharge, Ss0, ResultingS, NameAgent):-
180     best(Ds, D, S),
181     delete2([D|Ds], [S], Ds1),
182     S=s(Vertex,Distance,Path),
183     reverse([Vertex|Path], Path1),
184     Distance2 is Distance + ERecharge,
185     merge2(Ss0, [s(Vertex,Distance2,Path1)], Ss1),
186     create(Vertex, [Vertex|Path], Ds2),
187     delete2(Ds2, Ss1, Ds3),
188     incr(Ds3, Distance2, Ds4),
189     merge2(Ds1, Ds4, Ds5),
190     dijkstra_5(Ds5, ERecharge, Ss1, ResultingS, NameAgent).
191
192
193 % Dijkstra for closest Enemy
194 % Code is adapted from: http://colin.barker.pagesperso-orange.fr/lpa/dijkstra.htm
195
```

```
196 % main predicate
197
198 pathClosestEnemy(Start, LocationEnemy, NameEnemy, Path, Dist) :-
199   dijkstra6(Start, s(LocationEnemy, Dist, Path), NameEnemy), !.
200
201 % helping predicates
202
203 dijkstra6(Start, ResultingS, NameAgent):-
204   create(Start, [Start], Ds),
205   recharge(ERecharge),
206   dijkstra_6(Ds, ERecharge, [s(Start,0,[])], ResultingS, NameAgent).
207
208 dijkstra_6([], _, _, _, _) :- !, fail.
209 dijkstra_6([D|Ds], ERecharge, _, s(Vertex,Distance2,Path1), NameAgent):-
210   best(Ds,D,s(Vertex,Distance,Path)),
211   enabledEnemy(NameAgent, Vertex), %Enemy must be active by last info.
212   delete2([D|Ds], [s(Vertex,Distance,Path)], _),
213   reverse([Vertex|Path], Path1),
214   Distance2 is Distance + ERecharge, !. % the '!' makes sure only 1 solution (the shortest) is correct.
215
216 dijkstra_6([D|Ds], ERecharge, Ss0, ResultingS, NameAgent):-
217   best(Ds, D, S),
218   delete2([D|Ds], [S], Ds1),
219   S=s(Vertex,Distance,Path),
220   reverse([Vertex|Path], Path1),
221   Distance2 is Distance + ERecharge,
222   merge2(Ss0, [s(Vertex,Distance2,Path1)], Ss1),
223   create(Vertex, [Vertex|Path], Ds2),
224   delete2(Ds2, Ss1, Ds3),
225   incr(Ds3, Distance2, Ds4),
226   merge2(Ds1, Ds4, Ds5),
227   dijkstra_6(Ds5, ERecharge, Ss1, ResultingS, NameAgent).
228
229 % List all the information in the optimum zone
230
231 allInformationOptimumZone([],[],[]) :- not(optimum(_)), !.
232 allInformationOptimumZone(Agents, Nodes, Neighbours) :- optimum(Opt), team(Team),
233  allInformationOptimumZone([Opt], [], Nodes, Agents, Neighbours, Team), !.
234
235 allInformationOptimumZone([], _, [], [], [], _).
236 allInformationOptimumZone([First|ToConsider], Visited, [First|Nodes], Agents, Neighbours, Team) :-
237  vertexOwner(First, Team),
238  findall([NameAgent, First], visibleEntity(NameAgent, First, Team, normal), FoundAgents),
239  findall(Node, (e4(First, Node, _), not(member(Node, Visited))), FoundNodesTemp), list_to_set(
         FoundNodesTemp, FoundNodes),
240  union(FoundNodes, ToConsider, NewToConsider),
241  allInformationOptimumZone(NewToConsider, [First|Visited], Nodes, NewAgents, Neighbours, Team),
242  union(NewAgents, FoundAgents, Agents).
243 allInformationOptimumZone([First|ToConsider], Visited, Nodes, Agents, [First|Neighbours], Team) :-
244  not(vertexOwner(First, Team)),
245  allInformationOptimumZone(ToConsider, [First|Visited], Nodes, Agents, Neighbours, Team).
246
247 % Dijkstra for closest Visible Enemy
248 % Code is adapted from: http://colin.barker.pagesperso-orange.fr/lpa/dijkstra.htm
249
```

```
250 % main predicate
251
252 pathClosestVisibleEnemy(Start, LocationEnemy, NameEnemy, Path, Dist) :-
253     dijkstra8(Start, s(LocationEnemy, Dist, Path), NameEnemy), !.
254
255 % helping predicates
256
257 dijkstra8(Start, ResultingS, NameAgent):-
258     create(Start, [Start], Ds),
259     recharge(ERecharge),
260     dijkstra_8(Ds, ERecharge, [s(Start,0,[])], ResultingS, NameAgent).
261
262 dijkstra_8([], _, _, _, _) :- !, fail.
263 dijkstra_8([D|Ds], ERecharge, _, s(Vertex,Distance2,Path1), NameAgent):-
264     best(Ds,D,s(Vertex,Distance,Path)),
265     enabledEnemy(NameAgent, Vertex),
266     visibleEntity(NameAgent,_,_,_), %Enemy must be visible in current step.
267     delete2([D|Ds], [s(Vertex,Distance,Path)], _),
268     reverse([Vertex|Path], Path1),
269     Distance2 is Distance + ERecharge, !. % the '!' makes sure only 1 solution (the shortest) is correct.
270
271 dijkstra_8([D|Ds], ERecharge, Ss0, ResultingS, NameAgent):-
272     best(Ds, D, S),
273     delete2([D|Ds], [S], Ds1),
274     S=s(Vertex,Distance,Path),
275     reverse([Vertex|Path], Path1),
276     Distance2 is Distance + ERecharge,
277     merge2(Ss0, [s(Vertex,Distance2,Path1)], Ss1),
278     create(Vertex, [Vertex|Path], Ds2),
279     delete2(Ds2, Ss1, Ds3),
280     incr(Ds3, Distance2, Ds4),
281     merge2(Ds1, Ds4, Ds5),
282     dijkstra_8(Ds5, ERecharge, Ss1, ResultingS, NameAgent).
283
284 % General Dijkstra helping predicates
285 % Code is adapted from: http://colin.barker.pagesperso-orange.fr/lpa/dijkstra.htm
286
287 % create(Start, Path, Edges) is true if Edges is a list of structures s(Vertex,
288 % Distance, Path) containing, for each Vertex accessible from Start, the
289 % Distance from the Vertex and the specified Path. The list is sorted by the
290 % name of the Vertex.
291 create(Start, Path, Edges):-
292     maxEnergy(E), setof(s(Vertex,Edge,Path), (e(Start,Vertex,Edge), Edge =< E ), Edges), !.
293 create(_, _, []).
294
295 % best(Edges, Edge0, Edge) is true if Edge is the element of Edges, a list of
296 % structures s(Vertex, Distance, Path), having the smallest Distance. Edge0
297 % constitutes an upper bound.
298 best([], s(A,B,C), s(A,B,C)).
299 best([s(A,B,C)|Edges], Best0, Best):-
300     shorter(s(A,B,C), Best0), !,
301     best(Edges, s(A,B,C), Best).
302 best([_|Edges], Best0, Best):-
303     best(Edges, Best0, Best).
304
```

```
305 shorter(s(_,X,_), s(_,Y,_)):−X < Y.
306
307 % delete2(Xs, Ys, Zs) is true if Xs, Ys and Zs are lists of structures s(Vertex,
308 % Distance, Path) ordered by Vertex, and Zs is the result of deleting from Xs
309 % those elements having the same Vertex as elements in Ys.
310 delete2([], _, []).
311 delete2([X|Xs], [], [X|Xs]):−!.
312 delete2([X|Xs], [Y|Ys], Ds):−
313   eq(X, Y), !,
314   delete2(Xs, Ys, Ds).
315 delete2([X|Xs], [Y|Ys], [X|Ds]):−
316   lt(X, Y), !, delete2(Xs, [Y|Ys], Ds).
317 delete2([X|Xs], [_|Ys], Ds):−
318   delete2([X|Xs], Ys, Ds).
319
320 % merge2(Xs, Ys, Zs) is true if Zs is the result of merging Xs and Ys, where Xs,
321 % Ys and Zs are lists of structures s(Vertex, Distance, Path), and are
322 % ordered by Vertex. If an element in Xs has the same Vertex as an element
323 % in Ys, the element with the shorter Distance will be in Zs.
324 merge2([], Ys, Ys).
325 merge2([X|Xs], [], [X|Xs]):−!.
326 merge2([X|Xs], [Y|Ys], [X|Zs]):−
327   eq(X, Y), shorter(X, Y), !,
328   merge2(Xs, Ys, Zs).
329 merge2([X|Xs], [Y|Ys], [Y|Zs]):−
330   eq(X, Y), !,
331   merge2(Xs, Ys, Zs).
332 merge2([X|Xs], [Y|Ys], [X|Zs]):−
333   lt(X, Y), !,
334   merge2(Xs, [Y|Ys], Zs).
335 merge2([X|Xs], [Y|Ys], [Y|Zs]):−
336   merge2([X|Xs], Ys, Zs).
337
338 eq(s(X,_,_), s(X,_,_)).
339
340 lt(s(X,_,_), s(Y,_,_)):−X @< Y.
341
342 % incr(Xs, Incr, Ys) is true if Xs and Ys are lists of structures s(Vertex,
343 % Distance, Path), the only difference being that the value of Distance in Ys
344 % is Incr more than that in Xs.
345 incr([], _, []).
346 incr([s(V,D1,P)|Xs], Incr, [s(V,D2,P)|Ys]):−
347   D2 is D1 + Incr,
348   incr(Xs, Incr, Ys).
349
350 % predicate that finds all surveyed edges, and checks both ways to make sure not an edge is missed
351 e(X, Y, Z):− vertex(X, _, List), member([Z,Y], List), not(Z == unknown).
352 e(X, Y, Z):− vertex(Y, _, List), member([Z,X], List), not(Z == unknown).
353 e(X, Y, 5):− vertex(X, _, List), member([unknown,Y], List), vertex(Y, _, List2), member([unknown, X],
        List2).
354 e(X, Y, 5):− vertex(Y, _, List), member([unknown,X], List), vertex(X, _, List2), member([unknown, Y],
        List2).
355
356 % predicate that finds all survyed edges, but requires at least one of the sides of the edge to be in our zone
```

```
357 e4(X, Y, Z):- vertex(X, _, List), member([Z,Y], List), not(Z == unknown), team(OurTeam), (vertexOwner
        (X,OurTeam);vertexOwner(Y,OurTeam)).
358 e4(X, Y, Z):- vertex(Y, _, List), member([Z,X], List), not(Z == unknown), team(OurTeam), (vertexOwner
        (X,OurTeam);vertexOwner(Y,OurTeam)).
359 e4(X, Y, 5):- vertex(X, _, List), member([unknown,Y], List), vertex(Y, _, List2), member([unknown, X],
        List2), team(OurTeam), (vertexOwner(X,OurTeam);vertexOwner(Y,OurTeam)).
360 e4(X, Y, 5):- vertex(Y, _, List), member([unknown,X], List), vertex(X, _, List2), member([unknown, Y],
        List2), team(OurTeam), (vertexOwner(X,OurTeam);vertexOwner(Y,OurTeam)).
```

**Listing B.7:** code/disabled.mod2g

```
1  module disabledReceiveMail{
2   program[order=linearall]{
3    % Process the locations of our repairers.
4    forall bel(repairerLoc(Agent,Loc)) do delete(repairerLoc(Agent,Loc)).
5    forall bel(received(Agent,currentLoc(Loc))) do insert(repairerLoc(Agent,Loc)) + delete(received(Agent,
           currentLoc(Loc))).
6   }
7  }
8
9  module disabled{
10  program{
11   % Wait for nearby repairer when you are a repairer and other repairer has a higher priority.
12   if bel( role('Repairer'), role(Agent,'Repairer'), me(Name), Agent \= Name, visibleEntity(Agent,Pos,_,_),
13    (neighbour(Pos) ; currentPos(Pos)), compareAgents(Name,Agent,Agent))
14    then recharge.
15
16   % Wait for nearby repairer when you are not a repairer.
17   if bel( not(role('Repairer')), role(Agent,'Repairer'), me(Name), Agent \= Name, visibleEntity(Agent,Pos,
           _,_),
18    (neighbour(Pos) ; currentPos(Pos)))
19    then recharge.
20
21   % Find nearest repairer.
22   if bel( currentPos(Here), pathClosestRepairer(Here,Loc,Agent,[Here,Next|Path],_), reverse([Here,Next|
           Path],Reversed))
23    then gotoRepairer(Here,Next,Path,Reversed,Agent).
24
25   % Goto random neighbour.
26   if bel( neighbour(There), not(lastPos(There)) ) then advancedGoto(There).
27   if bel( currentPos(Here), visibleEdge(Here,There), not(lastPos(There)) ) then advancedGoto(There).
28  }
29 }
30
31 module gotoRepairer(Here,Next,Path,Reversed,Agent){
32  program{
33   % Move towards closest repairer and signal repairer.
34   if true then advancedGoto(Next) + send(Agent,helpPath(Reversed,Here)).
35  }
36 }
```

**Listing B.8:** code/explorer.mod2g

```
1  % Belief base management specific to the explorer
2  module explorerPercepts{
```

```
 program[order=linearall]{
  % If our last goto failed we are potentially under attack, feeling might be nescessary
  if bel(noFlee, lastAction(goto), lastActionResult(failed)) then delete(noFlee).

  % Makes sure the graph administration is performed after a probe and other agents receive this new
      correct information
  if bel(lastAction(probe), lastActionResult(successful)) then probeVertices.

  if bel(currentPos(Here), safePos(Here), noFlee) then delete(noFlee).
 }
}

% Sending messages specific for the explorer
module explorerReceiveMail{
 program[order=linearall]{
  % makes sure the explorer doesnt do work the other explorer already did

  forall bel(received(A,vertexProbed(Id,Value)), not(vertex(Id,_,_)))
   do delete(received(A,vertexProbed(Id,Value))) + insert(vertex(Id,Value,[])).

  forall bel(received(A,vertexProbed(Id,Value)), vertex(Id,unknown,List))
   do delete(received(A,vertexProbed(Id,Value))) + insert(not(vertex(Id,unknown,List)), vertex(Id,Value,
       List)).

  % the other agent found an optimum, determine which nodes now need exploring(probing) for the next
      phase
  if bel(received(Agent, optimum(Opt))) then {
   if true
    then insert(optimum(Opt))
    + delete(received(Agent, optimum(Opt))).
  }
  if bel(received(Agent, tempOptimum(Opt))) then {
   if true
    then insert(tempOptimum(Opt))
    + delete(received(Agent, tempOptimum(Opt))).
  }

  if bel(received(Agent, exploreZone(EZone))) then {
   if bel(exploreZone(E)) then insert(not(exploreZone(E)), exploreZone(EZone)) + delete(received(Agent,
       exploreZone(EZone))).
   if true then insert(exploreZone(EZone)) + delete(received(Agent, exploreZone(EZone))).
  }

  if bel(received(Agent, exploreRemove(X)), exploreZone(O), select(X,O,O2)) then insert(not(
      exploreZone(O)), exploreZone(O2)) + delete(received(Agent, exploreRemove(X))).
 }
}

% Module that makes sure an action is chosen for the explorer
module explorerAction{
 program[order=linearall]{
  %Drop the optimum goal when we exit the initial phase
  if a-goal(optimum) then {
   if bel(step(N), N>=150) then drop(optimum).
  }
```

```
53
54    %Insert optimum after 150 steps
55    if bel(not(optimum(Opt)), tempOptimum(TOpt), agentRankRole(Rank, 'Explorer'), Rank=:=0, step(N),
            N>=150, calcOptimumZone(TOpt, OptZone)) then
56     insert(optimum(TOpt), optZone(OptZone)) + send(allother, optimum(TOpt)) + send(allother, optZone(
            OptZone)).
57
58    %Determine what needs to be probed and broadcast to all.
59    if bel(optZone(O), not(exploreZone(E)), findall(Pos,(member(Pos,O),needProbe(Pos)),ExploreZone))
            then insert(exploreZone(ExploreZone)) + send(allother, exploreZone(ExploreZone)).
60
61    %If current explore-objective is completed, delete the corresponding belief
62    if bel(currentPos(Here), exploring(Here)) then delete(exploring(Here)).
63
64    %Pick a node to explore, then tell others not to explore it
65    if bel(not(exploring(_)),exploreZone(E), E \= [], agentRankRole(Rank,'Explorer'), nth0(Rank, E, Node))
            then insert(exploring(Node)) + send(allother, exploreRemove(Node)).
66
67
68    if true then {
69     % Agent is not safe, defend yourself
70     if bel(not(noFlee), currentPos(Here), not(safePos(Here))) then defense.
71
72     % if there are edges with unknown weight around the current node survey them
73     if bel( not(disabled), currentPos(Here), needSurvey(Here), agentRankHere(Rank) )
74      then selectSurvey(Rank).
75
76     % probe your node if it is unprobed
77     if bel( not(disabled), currentPos(Here), needProbe(Here), me(Name), team(Team),
78        findall(Agent, (visibleEntity(Agent,Here,Team,_), role(Agent,'Explorer')), Agents), agentRank(
               Agents,Name,Rank) )
79      then selectProbe(Rank).
80
81     % If we are looking for an optimum enter the module that has optimum finding behavior
82     if a-goal(optimum) then searchOptimal.
83
84     % When optimum is found but certain nodevalues still need exploring enter the module that makes sure
            this happens
85     if true then swarmProbe.
86
87     % An optimum is found but im not in the swarm, move a step closer to the optimum
88     if bel(currentPos(Pos), not(swarmPos(Pos)), optimum(Opt), path(Pos, Opt, [Here,Next|_], _))
89      then advancedGoto(Next).
90
91     % When in the swarm swarm along with the rest
92     if a-goal(swarm) then swarm.
93    }
94   }
95  }
96
97  % Explorer behavior when we have the entire map, probe as much as possible for maximum zone score!
98  module explorerSuperiority{
99   program{
100    % Probe the current node if required
101    if bel( not(disabled), currentPos(Here), needProbe(Here), me(Name), team(Team),
```

```
102       findall(Agent, (visibleEntity(Agent,Here,Team,_), role(Agent,'Explorer')), Agents), agentRank(Agents
              ,Name,Rank) )
103     then selectProbe(Rank).
104
105    % Go towards to closest node that still needs probing
106    if bel(currentPos(Start), pathClosestNonProbed(Start, NonProbedVertex, [Here,Next|Path], Dist))
107     then advancedGoto(Next).
108   }
109  }
110
111  % Module that contains behavior for explorers to find the optimal value node
112  module searchOptimal {
113   program[order=linearall]{
114   if bel(currentPos(Here), !, vertexValue(Here, Value), optimumValue(Current), Current<Value)
115    then {
116     if bel(step(N), N<150) then insert(tempOptimum(Here), not(optimumValue(Current)), optimumValue(
              Value)) + send(allother, tempOptimum(Here)).
117
118     if bel(step(N), N>=150, not(optimum(_)),calcOptimumZone(Here,OZone)) then insert(optimum(Here),
              not(optimumValue(Current)), optimumValue(Value)) + send(allother, optimum(Here)) + send(
              allother, optimumZone(OZone)).
119    }
120
121    if true then {
122     % find an unprobed neighbouring vertex
123     if bel(neighbour(There), needProbe(There), safePos(There)) then advancedGoto(There).
124
125     % find an unprobed neighbouring vertex
126     if bel(neighbour(There), needProbe(There), not((visibleEntity(_, There, Team, _), enemyTeam(Team))))
127      then advancedGoto(There) + insert(noFlee).
128
129     % find an unprobed neighbouring vertex
130     if bel(neighbour(There), needProbe(There))
131      then advancedGoto(There) + insert(noFlee).
132
133     % Find closest unprobed vertex
134     if bel(currentPos(Start), pathClosestNonProbed(Start, NonProbedVertex, [Here,Next|Path], Dist))
135      then advancedGoto(Next).
136    }
137   }
138  }
139
140  % Module that makes sure probing is handled in swarmphase
141  module swarmProbe {
142   program {
143    % If this vertex needs to be explored then goto an unprobed neighbour to probe it.
144    if bel(currentPos(Here), vertexValue(Here, Value), neighbour(Here, There), needProbe(There), team(
              Team), vertexOwner(There, Team))
145     then advancedGoto(There).
146
147    if bel(exploring(X), path(Here,Vertex,[Here,Next|Path],_))
148     then advancedGoto(Next).
149    % Find the closest unprobed vertex which is a neighbour of a vertex which needs to be explored
150    % if bel(currentPos(Here), pathClosestNonProbedWithExtraChecks(Here, NonProbedVertex, [Here,
              Next | Path], _))
```

```
151     % then advancedGoto(Next).
152   }
153 }
```

**Listing B.9:** code/generalBeliefs.pl

```
1  % some variables so that they can be updated later
2  lastPos(unknown).
3  oldZone(unknown).
4  step(unknown).
5
6  % predicate that will contain info about the optimum zone, updated each round in common percepts
7  optimumInfo([], [], []).
8
9  % makes sure agents wait for mails from other agents in first round.
10 doneAction.
11 donePercepts.
12 doneMailing.
```

**Listing B.10:** code/generalKnowledge.pl

```
1  % energy/money checks
2  energyGE(Nr) :- energy(E), E >= Nr.
3  moneyGE(Nr) :- money(M), M >= Nr.
4  maxEnergy(E) :- disabled, maxEnergyDisabled(E), !.
5  maxEnergy(E) :- not(disabled), maxEnergyWorking(E).
6  recharge(Nr) :- not(disabled), maxEnergy(E), Nr is round(0.5*E).
7  recharge(Nr) :- disabled, maxEnergy(E), Nr is round(0.3*E).
8
9  % Short predicate for current position of agent
10 role(Role) :- me(Id), role(Id, Role).
11
12 % Predicate that defines when an agent can not be trusted within the swarm
13 % This means the agent will not take the swarm into account when making its moves
14 independableAgent(Agent) :- me(Agent).
15 independableAgent(Agent) :- currentPos(Agent, Pos), insideZone(Pos).
16 independableAgent(Agent) :- role(Agent, 'Saboteur'), visibleEntity(_,_,Team,normal), enemyTeam(Team)
       .
17 independableAgent(Agent) :- role(Agent, 'Repairer'), team(Team), visibleEntity(Id,Vertex,Team,disabled)
       , not(currentPos(Agent, Vertex)).
18
19 % Two agents are connected when there are one or two edges between them
20 connectedAgent(Agent1, Agent2) :- team(Team), visibleEntity(Agent1,Pos1,Team,normal), visibleEntity(
       Agent2,Pos2,Team,normal),
21  visibleEdge(Pos1, Pos2), not(independableAgent(Agent2)), vertexOwner(Pos1, Team), vertexOwner(Pos2
       , Team).
22 connectedAgent(Agent1, Agent2) :- team(Team), visibleEntity(Agent1,Pos1,Team,normal), visibleEntity(
       Agent2,Pos2,Team,normal),
23  visibleEdge(Pos1, Pos3), visibleEdge(Pos3,Pos2), not(Pos1 == Pos2), not(independableAgent(Agent2)),
24  vertexOwner(Pos1, Team), vertexOwner(Pos2, Team), vertexOwner(Pos3, Team).
25
26 % Some short predicates for information about our optimum zone(the swarm around the found optimum)
27 agentsInOptimumZone(A) :- optimumInfo(A, _, _).
28 neighboursOfOptimumZone(F) :- optimumInfo(_, _, F).
29
```

```
% Team determination
enemyTeam(T) :- inspectedEntity(_, T, _, _, _, _, _, _, _, _).
enemyTeam(T) :- not(team(T)), T \= none.
% defines when an agent is disabled
disabled :- health(0).

% predicates for determining when a node or it's neighbour needs surveying
needSurvey(Vertex) :- vertex(Vertex,_,[]),!.
needSurvey(Vertex) :- not(vertex(Vertex,_,_)).
neighbourNeedSurvey(ID) :- currentPos(Here), neighbourNeedSurvey(Here,ID).
neighbourNeedSurvey(Vertex,ID) :- vertex(Vertex,_,List), member([_,ID],List), needSurvey(ID).

% true when an optimum is found
optimum :- optimum(_).

% Defines whether an enemy is to be considered dangerous for sure
dangerousEnemy(Id) :- inspectedEnemy(Id, 'Saboteur'), !.
dangerousEnemy(Id) :- not(inspectedEnemy(Id, _)), !, not((inspectedEnemy(Id2, 'Saboteur'), !,
 inspectedEnemy(Id3, 'Saboteur'), Id2 \= Id3, !, inspectedEnemy(Id4, 'Saboteur'), Id2\=Id4, Id3\=Id4, !,
 inspectedEnemy(Id5, 'Saboteur'), Id2\=Id5, Id3\=Id5, Id4\=Id5)).
% Enemy is passive when disabled, can also be used on allies.
passiveEnemy(Id) :- visibleEntity(Id,_,_,disabled), !.
passiveEnemy(Id) :- inspectedEnemy(Id,Role), !, Role \= 'Saboteur'.
passiveEnemy(Id) :- not(inspectedEnemy(Id,_)), !, inspectedEnemy(Id2, 'Saboteur'), !,
 inspectedEnemy(Id3, 'Saboteur'), Id2 \= Id3, !, inspectedEnemy(Id4, 'Saboteur'), Id2\=Id4, Id3\=Id4, !,
 inspectedEnemy(Id5, 'Saboteur'), Id2\=Id5, Id3\=Id5, Id4\=Id5.

% Short predicate to extract the most useful information from an inspected enemy
inspectedEnemy(Id,Role) :- inspectedEntity(Id, _, Role, _, _, _, _, _, _, _).

% True when we have 90 percent or more of all nodes in our posession
allMapAreBelongToUs :- team(Team), findall(Vertex, vertexOwner(Vertex, Team), List), vertices(V), V2
      is 0.9*V, length(List, V3), V3 > V2.

% Rule that defines whether it is requires to parry, speaks for itself ;)
needToParry :- currentPos(Here), !, visibleEntity(Agent,Here,_,normal), dangerousEnemy(Agent).
```

**Listing B.11:** code/inspector.mod2g

```
module inspectorPercepts{
 program[order=linearall]{
  % Process inspect data.
  if bel(lastAction(inspect), lastActionResult(successful)) then inspectEntityPercept.
 }
}

module inspectorReceiveMail{
 program[order=linearall]{
  % No special inspector mails need to be handled.
  if true then exit-module.
 }
}

module inspectorAction {
 program {
```

```
    % inspect when possible
    if bel( uninspectedNear ) then {
     if true then inspect.
     if true then recharge.
    }

    % defend yourself when not safe
    if bel(currentPos(Here), not(safePos(Here))) then defense.

    % find someone to inspect
    if bel( currentPos(Here), !, visibleEntity(Agent, There, Team, _), enemyTeam(Team),
      (uninspectedEntity(Agent); (inspectedEnemy(Agent, 'Saboteur'),lastInspect(Agent,LI), step(S), LI2 is
            LI + 50, LI2 < S)), !,
      path(Here, There, [Here,Next|GotoPath],_), ! )
     then advancedGoto(Next).

    % swarm
    if a-goal(swarm) then swarm.

    % walk to the optimum
    if bel(optimum(X), currentPos(Pos), path(Pos,X,[Here,Next|Path],_))
     then advancedGoto(Next).

    % randomly explore
    if true then explore.
   }
  }

  module inspectorSuperiority{
   program{
    % Stay idle.
    if true then recharge.
   }
  }
```

**Listing B.12:** code/navigationKnowledge.pl

```
% Finds all neighbouring nodes of the current position
neighbour(Neighbour) :- currentPos(Id),!, neighbour(Id,_,Neighbour).

% Finds all neighbouring nodes of a given node
neighbour(Id,Neighbour) :- neighbour(Id,_,Neighbour).

% Finds all neighbouring nodes of a given node, and the weight of their connection
neighbour(Id,Weight,Neighbour) :- vertex(Id,_,List), member([Weight,Neighbour],List).

%Fix to earlier double Goto actionspecs
shouldGoTo(Here,There) :- (neighbour(Here,Weight,There), energyGE(Weight)) ; (not(neighbour(Here,
      There)), visibleEdge(Here,There)).

% This predicate determines when a node is to be considered safe to stand on, this means no unknown role
      agent or saboteur can be at this location
safePos(P) :- not((visibleEntity(A, P, T, normal), enemyTeam(T), not(passiveEnemy(A)))),
 not((neighbour(P, P2), visibleEntity(A2, P2, T, normal), enemyTeam(T), inspectedEnemy(A2,'Saboteur')
       )).
```

```
16
17 % This is true when the agent has the highest rank(based on its name) of all agents on this node
18 kingOfTheHill :- agentRankHere(0).
19
20 % Determines if the agent is the only agent on its position
21 foreverAlone :- not( (currentPos(Pos), me(Me), team(Team), !, visibleEntity(ID, Pos, Team ,_), Me \= ID )
       ).
22
23 % Compares agents names to find which name has a higher 'value'
24 compareAgents(Agent1,Agent2,Agent2) :- Agent1 @< Agent2.
25 compareAgents(Agent1,Agent2,Agent1) :- Agent1 @> Agent2.
26
27 % Returns the rank(based on its name) of an agent compared to all other agents on its node
28 agentRankHere(Rank) :- currentPos(Here), me(Name), team(Team), !,
29  setof(Agent, visibleEntity(Agent,Here,Team,normal), Agents), agentRank(Agents,Name,Rank).
30
31 % Returns the rank(based on its name) of an agent compared to all other agents on its node of a specific
       role
32 agentRankHere(Rank, Role) :- currentPos(Here), me(Name), team(Team), !,
33  setof(Agent, (visibleEntity(Agent,Here,Team,normal), role(Agent, Role)), Agents), agentRank(Agents,
       Name,Rank).
34
35 % Returns the rank(based on its name) of an agent compared to all other agents of a specific role
36 agentRankRole(Rank, Role) :- me(Name), team(Team), !,
37  setof(Agent, (visibleEntity(Agent,_,Team,normal), role(Agent, Role)), Agents), agentRank(Agents,Name,
       Rank).
38
39 %Ranks the agents using the nth0 predicate
40 agentRank(List,Agent,Rank) :- nth0(Rank, List, Agent), !.
41
42 % Predicate that selects a Neighbour on index Number from the list of Neighbours, useful in combination
       with agentrank for splitting up, agent with rank 0 will not get a neighbour
43 selectNeighbour(List, Number, Neighbour) :- length(List, Size), Num is mod(Number,Size), nth1(Num,
       List, Neighbour), !.
44
45 % Predicate that selects a Destination on index Number from the list of Destinations, useful for splitting up
       in combination with agentrank when multiple destinations are available
46 selectDestination(List, Number, Destination) :- length(List,Size), Num is mod(Number,Size), nth0(Num,
       List, Destination), !.
47
48 % Short predicates for vertex information
49 vertexValue(Id,Value) :- vertex(Id,Value,_).
50 vertexValue(Id,unknown) :- not(vertex(Id,_,_)).
51
52 % predicate that determines if a position results situation where the agent maintains connection with
       another agent
53 connectedPos(X, Agent) :- currentPos(Agent, Y), not(independableAgent(Agent)), visibleEdge(X,Y).
54 connectedPos(X, Agent) :- currentPos(Agent, Z), not(independableAgent(Agent)), X \== Z,
55  visibleEdge(Z,Y), team(Team), vertexOwner(Y, Team), visibleEdge(Y,X).
56
57 % a vertex that is a swarmpos is a position that makes sure the agent is still connected to 2 other agents
58 swarmPos(X) :- connectedPos(X, Agent1), connectedPos(X, Agent2), Agent1 \== Agent2, !.
59
60 % check if the agent is currently in the zone that contains the found optimum
61 inOptimumZone :- me(Id), agentsInOptimumZone(A), member([Id,_], A).
```

```
62
63 %Counts enemy saboteurs on this vertex
64 getEnemySaboteurCount(Sabs, Vertex) :- setof(ID,(enemyTeam(Team), visibleEntity(ID, Vertex, Team, _)
       , inspectedEnemy(ID,'Saboteur')),List), length(List, Sabs).
65
66 %Counts our saboteurs on this vertex
67 getSaboteurCount(Sabs, Vertex) :- setof(ID,(team(Team), visibleEntity(ID, Vertex, Team, _), role(ID, '
       saboteur')),List), length(List, Sabs).
```

**Listing B.13:** code/pathing.mod2g

```
1  module gotoSplit(Rank,List){
2   knowledge{
3    % Data reformatting
4    stripList([],[]).
5    stripList([[Value,Vertex]|List],[Vertex|SList]) :- stripList(List,SList).
6   }
7   program{
8    % List = [[Value,Vertex],[...]] Highest after!
9    if bel(stripList(List,SList), selectDestination(SList,Rank,Vertex)) then advancedGoto(Vertex).
10   % List = [Vertex,...,Vertex]
11   if bel(selectDestination(List,Rank,Vertex)) then advancedGoto(Vertex).
12  }
13 }
14
15 module moveSplit(Rank,List){
16  program{
17   % List = [[Value,Vertex],[...],[Value,Vertex]]
18   if bel(stripList(List,SList), selectDestination(SList,Rank,Vertex), currentPos(Here), path(Here,Vertex,[
         Here,Next|Path],_))
19    then advancedGoto(Next).
20   % List = [Vertex,...,Vertex]
21   if bel(selectDestination(List,Rank,Vertex), currentPos(Here), path(Here,Vertex,[Here,Next|Path],_))
22    then advancedGoto(Next).
23  }
24 }
25
26 module gotoNeighbour(Rank,Unknown,Safe){
27  program{
28   if bel( Unknown == true, Safe == true, maxEnergy(E), energyGE(E), currentPos(Here), setof(Neighbour,
         (visibleEdge(Here,Neighbour),
29    safePos(Neighbour)), Neighbours), selectNeighbour(Neighbours,Rank,Vertex))
30    then advancedGoto(Vertex).
31
32   if bel( Unknown == true, maxEnergy(E), energyGE(E), currentPos(Here),setof(Neighbour, visibleEdge(
         Here,Neighbour), Neighbours),
33    selectNeighbour(Neighbours,Rank,Vertex))
34    then advancedGoto(Vertex).
35
36   if bel( Safe == true, X is Rank + 1, setof(Neighbour, (neighbour(Neighbour), safePos(Neighbour)),
         Neighbours), selectNeighbour(Neighbours,X,Vertex) )
37    then advancedGoto(Vertex).
38
39   if bel( X is Rank + 1, setof(Neighbour, neighbour(Neighbour), Neighbours), selectNeighbour(
         Neighbours,X,Vertex))
```

```
    then advancedGoto(Vertex).
  }
}

module advancedGoto(Destination){
 program{
  % Goto pre condition checks if we can move over explored edges.
  if bel( currentPos(Here), not(needSurvey(Here)) ) then {
   if true then goto(Destination).
   if true then recharge.
  }

  % Recharge to at least 9 energy before moving over an unsurveyed edge.
  if bel( energyGE(9) ) then goto(Destination).
  if true then recharge.
 }
}

module selectProbe(Rank){
 program{
  % Use probe action when I am rank 0 (Highest)
  if bel( Rank == 0 ) then probe.
  % Go to a neighbour if I am not rank 0
  if true then gotoNeighbour(Rank,true,false).
  if true then recharge.
 }
}

module selectSurvey(Rank){
 program{
  % Use survey action when I am rank 0 (Highest)
  if bel( Rank == 0 ) then survey.
  % Go to a neighbour if I am not rank 0
  if true then gotoNeighbour(Rank,true,false).
  if true then recharge.
 }
}
```

**Listing B.14:** code/perceptKnowledge.pl

```
% Extraction some information about the agent itself from the percepts
money(M) :- percept(money(M)).
energy(E) :- percept(energy(E)).
maxEnergyWorking(E) :- percept(maxEnergy(E)).
maxEnergyDisabled(E) :- percept(maxEnergyDisabled(E)).
strength(S) :- percept(strength(S)).
maxHealth(H) :- percept(maxHealth(H)).

% Extracting visible entities, vertices and edges from the percepts
visibleEntity(Id,Vertex,Team,Status) :- percept(visibleEntity(Id,Vertex,Team,Status)).
vertexOwner(Vertex,Team) :- percept(visibleVertex(Vertex, Team)).
visibleEdge(Vertex1,Vertex2) :- percept(visibleEdge(Vertex1,Vertex2)).
visibleEdge(Vertex1,Vertex2) :- percept(visibleEdge(Vertex2,Vertex1)).

% Extracting the agents current position from the percepts
```

```
currentPos(Agent,Vertex) :- percept(visibleEntity(Agent,Vertex,_,_)).

% Extracting round information from the percepts
lastAction(Action) :- percept(lastAction(Action)).
lastActionResult(Result) :- percept(lastActionResult(Result)).
```

**Listing B.15:** code/repairer.mod2g

```
module repairerPercepts{
 program[order=linearall]{
  % Send own location to other agents.
  if bel(currentPos(Pos)) then send(allother,currentLoc(Pos)).
  forall bel(repairPath(Path)) do delete(repairPath(Path)).
  if bel(currentPos(Here),team(Team),!,visibleEntity(EID,Here,ETeam,_),enemyTeam(ETeam),
        inspectedEnemy(EID,'saboteur'),!,
   not((visibleEntity(AID,Here,Team,_), role(AID,'saboteur'))),!, role(SID,'saboteur')) then send(SID,
         neededHere(Here)).

 }
}

module repairerReceiveMail{
 program[order=linearall]{
  % Cancel moving to an agent that is repaired.
  if bel(received(Agent,fixed), repairing(Agent)) then delete(repairing(Agent), received(Agent,fixed)).

  % Plan correct path towards disabled agents.
  % Drop current help path and start helping another repairer.
  if bel(received(Agent,helpPath([First,Second|Path],Destination)), repairing(Someone), role(Agent,'
        Repairer'), Someone \= Agent)
   then delete(repairing(Someone)) + buildPath(First,Second,Path,Agent,Destination).
  % Keep moving towards the agent I am currently repairing.
  if bel(received(Agent,helpPath([First,Second|Path],Destination)), repairing(Agent))
   then delete(repairing(Agent)) + buildPath(First,Second,Path,Agent,Destination).
  % Move towards an agent that needs repair when I am not repairing anything.
  if bel(received(Agent,helpPath([First,Second|Path],Destination)), not(repairing(_)))
   then buildPath(First,Second,Path,Agent,Destination).
 }
}

module buildPath(First,Second,Path,Agent,Destination){
 program{
  if bel(currentPos(First)) then insert(repairPath(Second), repairing(Agent)).
  if bel(neighbour(First)) then insert(repairPath(First), repairing(Agent)).
  if bel(lastPos(Pos)) then insert(repairPath(Pos), repairing(Agent)).
 }
}

module repairerAction{
 program{
  % Fix a nearby ally.
  if bel(disabledAllyNear(ID,Vertex)) then repairerRepair(ID,Vertex).

  % Move to a not nearby ally that needs repairs.
  if bel(repairing(_), repairPath(Next)) then advancedGoto(Next).
```

```
45
46    % Find help, because I am disabled.
47    if bel(disabled) then disabled.
48
49    % Defend if my current location has dangerous enemies nearby.
50    if bel(currentPos(Here), not(safePos(Here))) then defense.
51
52    % Swarm if I am in the optimum zone.
53    if a-goal(swarm) then swarm.
54
55    % Move towards the optimum zone if I am not in it.
56    if bel(optimum(X), currentPos(Pos), path(Pos,X,[Here,Next|Path],_))
57     then advancedGoto(Next).
58
59    % Explore the map.
60    if true then explore.
61   }
62  }
63
64  module repairerSuperiority{
65   program{
66    % Fix a nearby ally.
67    if bel(disabledAllyNear(ID,Vertex)) then repairerRepair(ID,Vertex).
68
69    % Move to a not nearby ally that needs repairs.
70    if bel(repairing(_), repairPath([Next|_])) then advancedGoto(Next).
71
72    % Find help, because I am disabled.
73    if bel(disabled) then disabled.
74
75    % Do nothing.
76    if true then recharge.
77   }
78  }
79
80  module repairerRepair(ID,Vertex){
81   program{
82    % Repair target.
83    if bel( currentPos(Vertex) ) then {
84     if true then repair(ID).
85     if true then recharge.
86    }
87    % Goto vertex with disabled agent.
88    if true then advancedGoto(Vertex).
89   }
90  }
```

**Listing B.16:** code/roleKnowledge.pl

```
1  %% Explorer specific knowledge
2  needProbe(Vertex) :- vertex(Vertex,unknown,_).
3  needProbe(Vertex) :- not(vertex(Vertex,_,_)).
4
5  %% Saboteur specific knowledge
6
```

```
7  %Because a target cannor be permanently disabled your never done hunting it.
8  hunt(ID) :- !, fail.
9
10 % To determine which enemies are on the current position
11 enemyHere(ID) :- currentPos(Vertex),!, visibleEntity(ID,Vertex,Team,_), enemyTeam(Team).
12
13 % To deterine which enemies are close to the current position
14 enemyNear(ID,Vertex) :- enemyHere(ID).
15 enemyNear(Id,Pos) :- neighbour(Pos), visibleEntity(Id,Pos,Team,_), enemyTeam(Team).
16
17 % To determine when a non-disabled enemy is at your position
18 enabledEnemyHere(Id) :- currentPos(Vertex),!, visibleEntity(Id,Vertex,Team,normal), enemyTeam(Team).
19
20 % when an non-disabled enemy is at or next to your position
21 enabledEnemyNear(ID,Vertex) :- enabledEnemyHere(ID).
22 enabledEnemyNear(Id,Pos) :- neighbour(Pos), visibleEntity(Id,Pos,Team,normal), enemyTeam(Team).
23
24 % A list of all locations near where there are enemies and we do not have superiority
25 attackEnemiesNear(List) :- setof(Vertex, (enabledEnemyNear(ID,Vertex), not(vertexSuperiority(Vertex))),
       List).
26
27 %Determines if we have stricly more sabs than the enemy on a given vertex
28 vertexSuperiority(Vertex) :- getEnemySaboteurCount(ESabs, Vertex), getSaboteurCount(Sabs, Vertex),
       Sabs>=ESabs.
29
30 % Predicate to find all enemy repairers
31 enemyRepairerList(List) :- setof(ID,(inspectedEnemy(ID,'Repairer')),List).
32
33 % Short predicate for finding enemies worth attacking
34 enabledEnemy(ID,Vertex) :- enemyStatus(ID,Vertex,normal), not(vertexSuperiority(Vertex)).
35
36 %% Repairer specific knowledge
37
38 % Predicate that returns disabled allies near or on the current position
39 disabledAllyNear(ID,Here) :- currentPos(Here), team(Team), me(Me), visibleEntity(ID,Here,Team,
       disabled), ID \= Me, !.
40 disabledAllyNear(ID,Vertex) :- team(Team), neighbour(Vertex), visibleEntity(ID,Vertex,Team,disabled), !.
41 disabledAllyNear(ID,Vertex) :- currentPos(Here), team(Team), visibleEdge(Here,Vertex), visibleEntity(ID
       ,Vertex,Team,disabled), !.
42
43 %% Inspector specific knowledge
44
45 % Predicate that returns uninspected agents close to the inspector
46 % This also makes sure enemy saboteurs are suitable for inspection again when last inspection is older than
       50 steps
47 uninspectedNear :- visibleEntity(Agent,Vertex,Team,_), enemyTeam(Team), (currentPos(Vertex) ;
       neighbour(Vertex)),
48  (not(inspectedEntity(Agent,_,_,_,_,_,_,_,_,_)); ( inspectedEnemy(Agent, 'Saboteur'), lastInspect(Agent,LI
       ), step(S), LI2 is LI + 50, LI2 < S)).
49 uninspectedEntity(Agent) :- not(inspectedEntity(Agent,_,_,_,_,_,_,_,_,_)).
```

**Listing B.17:** code/saboteur.mod2g

```
1  %Saboteur specific Percept handeling
2  module saboteurPercepts{
```

```
 program[order=linearall]{
  if true then exit-module.
 }
}

module saboteurReceiveMail{
 program[order=linearall]{
  % When receiving location info about enemies save it if enemy is not currently seen by you.
  forall bel(received(Sender,visibleEntity(EID,Vertex,Team,X)),enemyStatus(EID,OVertex,Y), not(
      visibleEntity(EID,_,_,_)) )
   do delete(received(A,visibleEntity(EID,Vertex,Team,X))) +
    insert(not(enemyStatus(EID,OVertex,Y)), enemyStatus(EID,Vertex,X)).

  forall bel(received(Sender,visibleEntity(EID,Vertex,Team,X)),not(enemyStatus(EID,_,_)), not(
      visibleEntity(EID,_,_,_)) )
   do delete(received(A,visibleEntity(EID,Vertex,Team,X))) + insert(enemyStatus(EID,Vertex,X)).

  % Disapearing enemies should be removed so we don't go on wild chases.
  forall bel( received(Sender, not(visibleEntity(EID,Vertex,Team,X))),enemyStatus(EID,Vertex,Y), not(
      visibleEntity(EID,_,_,_)) )
   do delete( received(A, not(visibleEntity(EID,Vertex,Team,X))), enemyStatus(EID,Vertex,Y) ).

  % When help is demanded by a repairer go help.
  if bel(received(Sender,neededHere(Target)),currentPos(Target)) then delete(received(_,neededHere(
      Target))).
  if bel(received(Sender,neededHere(Target)),currentPos(Pos), path(Pos, Target, [Here,Next|Path],Dist))
   then delete(received(_,neededHere(Target))) + advancedGoto(Next).
 }
}

module saboteurAction{
 program{
  %Clean up for Superiority module
  if goal(hunt(ID)) then drop(hunt(ID)).

  %Just walk away if this vertex is covered by other sabs
  if bel(enabledEnemyHere(ID), inspectedEnemy(ID,'Saboteur'), currentPos(Vertex),
      getEnemySaboteurCount(Sabs, Vertex),
  Sabs=\=0, agentRankHere(Rank, 'Saboteur'), Rank >= Sabs) then gotoNeighbour(Rank,false,false).

  % Attack enemy on this vertex
  % Preference to hit Saboteur and Repairer above other targets
  if bel(enabledEnemyHere(ID), currentPos(Vertex), (inspectedEnemy(ID,'Saboteur') ; inspectedEnemy(
      ID,'Repairer')) )
   then saboteurAttack(ID,Vertex).
  if bel(enabledEnemyHere(ID), currentPos(Vertex)) then saboteurAttack(ID,Vertex).

  % if the other saboteur is also at your location split up. %AgentRankHere is fine, changed to avoid
      superior vertices
  if bel(currentPos(Vertex),enabledEnemyNear(_,Y),!, visibleEntity(ID,Vertex,_,_), role(ID,'Saboteur'),
      not(me(ID)),!,
   attackEnemiesNear(List), agentRankHere(Rank) ) then gotoSplit(Rank,List).

  %Attack enemy on nearby vertex
```

```
49    if bel(enabledEnemyNear(ID,Vertex), not(visibleEntity(ID,Vertex,Team,_)), team(Team), role(ID,'
          Saboteur')) then {
50     if bel(inspectedEnemy(ID,'Saboteur') ; inspectedEnemy(ID,'Repairer') ) then saboteurAttack(ID,Vertex
           ).
51     if true then saboteurAttack(ID,Vertex).
52    }
53
54    % buy strength upgrade according to highest inspected health
55    if bel(strength(S), !, inspectedEntity(_, _, _, _, _, _, _, MaxHealth, _, _), S < MaxHealth, money(M), !, M
          >= 4) then {
56     if true then buy(sabotageDevice).
57     if true then recharge.
58    }
59
60    % buy health upgrade according to highest inspect enemy strength
61    if bel(maxHealth(H), !, inspectedEntity(_, _,'Saboteur', _, _, _, _, _, Strength, _), H =< Strength, money(
          M), !, M >= 4)
62    then {
63     if true then buy(shield).
64     if true then recharge.
65    }
66
67    %attack nearest visible enemy (only works in zones because otherwise it would have already been
          handled above)
68    if bel( currentPos(Start), pathClosestVisibleEnemy(Start, LocationEnemy, NameEnemy, [Here,Next|Path
          ], Dist),! )
69     then advancedGoto(Next).
70
71    %Head to optimal zone if we are not yet in there, ie. we are not in the swarm.
72    if not(goal(swarm)), bel( currentPos(Pos), optimum(Opt), !, path(Pos, Opt, [Here,Next|List], _),!) then
          advancedGoto(Next).
73
74    % Go into swarming mode.
75    if a-goal(swarm) then swarm.
76
77    %attack nearest enemy. This works for all known enemies and is used before we have found an optimal
          position.
78    if bel( currentPos(Start), pathClosestEnemy(Start, LocationEnemy, NameEnemy, [Here,Next|Path], Dist)
          ,! )
79     then advancedGoto(Next).
80
81    %Fail save
82    if true then explore.
83   }
84  }
85
86  module saboteurAttack(ID,Vertex){
87   program{
88    % Attack target if on this location.
89    if bel( currentPos(Vertex) ) then {
90     %If your last attack action was at the same target who parried and there is another active target hit the
           other instead
91     if bel( lastActionResult('Parried'), lastAttacked(ID),!, enabledEnemyHere(AID), AID \== ID ) then
           attack(AID).
92     if true then attack(ID).
```

```
    if true then recharge.
   }
   % Goto vertex with enemy agent.
   if true then advancedGoto(Vertex).
  }
 }

 module saboteurSuperiority{
  knowledge{
   %pick a target per saboteur
   selectTarget(List, Number, Target) :- length(List,Size), Num is mod(Number,Size), nth0(Num, List,
        Target), !.

   %get the rank of the saboteur
   saboteurRank(Rank) :- me(Name),!,setof(ID, role(ID,'saboteur'), Agents), agentRank(Agents,Name,
        Rank).

  }
  program{
   %hunt if you can
   if goal(hunt(ID)) then hunt.
   if bel(saboteurRank(Rank), enemyRepairerList(List),selectTarget(List,Rank,Target)) then adopt(hunt(
        Target)) + hunt.
  }
 }

 %Chase after and attack your target.
 module hunt{
  program{
   if goal(hunt(ID)), bel(enemyNear(ID,Vertex)) then saboteurAttack(ID,Vertex).
   if goal(hunt(ID)), bel(enemyStatus(ID,Vertex,_),currentPos(Here),!,path(Here,Vertex,[Here,NextlList],_))
    then advancedGoto(Next).
   % if you can't find target walk around random to find him.
   if bel(neighbour(Next)) then advancedGoto(Next).
  }
 }
```

**Listing B.18:** code/sentinel.mod2g

```
1  module sentinelPercepts{
2   program[order=linearall]{
3    % No special sentinel percepts need to be handled.
4    if true then exit-module.
5   }
6  }
7
8  module sentinelReceiveMail{
9   program[order=linearall]{
10   % No special sentinel mails need to be handled.
11   if true then exit-module.
12  }
13 }
14
15 module sentinelAction{
16  program{
```

```
    % Defend if my current location has dangerous enemies nearby.
    if bel(currentPos(Here), not(safePos(Here))) then defense.

    % Swarm if I am in the optimum zone.
    if a-goal(swarm) then swarm.

    % Move towards the optimum zone if I am not in it.
    if bel(optimum(X), currentPos(Pos), path(Pos,X,[Here,Next|Path],_))
     then advancedGoto(Next).

    % Explore the map.
    if true then explore.
   }
}

module sentinelSuperiority{
 program{
   % Survey entire map for additional achievement points.
   if true then superiorityExplore.
  }
}
```

**Listing B.19:** code/superiority.mod2g

```
module superioritySelect{
 program{
   % Perform behavior that is specific for your role when we have most of the map
   if bel(role('Repairer')) then repairerSuperiority.
   if bel(role('Saboteur')) then saboteurSuperiority.
   if bel(role('Inspector')) then inspectorSuperiority.
   if bel(role('Explorer')) then explorerSuperiority.
   if bel(role('Sentinel')) then sentinelSuperiority.
  }
}

module superiorityExplore{
 program{
   % if there are edges with unknown weight around the current node survey them
   if bel(currentPos(Here), !, needSurvey(Here), agentRankHere(Rank))
    then selectSurvey(Rank).

   % Find closest unsurveyed vertex and move in that direction
   if bel(foreverAlone, currentPos(Start), pathClosestNonSurveyed(Start, NonSurveyedVertex, [Here,Next|
        Path], Dist))
    then advancedGoto(Next).

   % When multiple agents are on the node and there are an unsurveyed neighbours, try to split up over
        these neighbours
   if bel(not(foreverAlone), agentRankHere(Rank), neighbourNeedSurvey(Any)) then gotoNeighbour(Rank
        , true, false).

   % recharge when you got nothing better to do
   if true then recharge.
  }
}
```

**Listing B.20:** code/swarm.mod2g

```
module swarm{
 knowledge {
  expandPos(ID) :- vertexOwner(ID,none), swarmPos(ID).

  expandDest(List3, Pos) :- findall([Value, Neighbour], ( neighbour(Pos, Neighbour), expandPos(
      Neighbour), vertexValue(Neighbour, Value), Value \== unknown ), List), not(List == []), sort(List,
      List2), reverse(List2, List3).
  safeExpandDest(List3, Pos) :- findall([Value, Neighbour], ( neighbour(Pos, Neighbour), expandPos(
      Neighbour), safePos(Neighbour), vertexValue(Neighbour, Value), Value \== unknown), List), not(
      List == []), sort(List, List2), reverse(List2, List3).
  expandDest(List) :- currentPos(Pos), expandDest(List, Pos).
  safeExpandDest(List) :- currentPos(Pos), safeExpandDest(List, Pos).

  bestExpandDest(ID, Value, Pos):- expandDest(List, Pos), List = [[Value, ID]|_].

  edgeDest(List3) :- neighboursOfOptimumZone(F), !, findall([Value, Vertex], (member(Vertex, F),
      vertexValue(Vertex, Value), Value \== unknown), List), not(List == []), sort(List, List2), reverse(
      List2, List3).
  safeEdgeDest(List3) :- neighboursOfOptimumZone(F), !, findall([Value, Vertex], (member(Vertex, F),
      safePos(Vertex), vertexValue(Vertex, Value), Value \== unknown), List), not(List == []), sort(List,
      List2), reverse(List2, List3).

  insideZone(Pos) :- not((neighbour(Pos, There), vertexOwner(There, Team), not(team(Team)))).
  insideZone :- currentPos(Pos), insideZone(Pos).
 }

 program{
  % If surrounded by swarm move to the edges
  if bel(insideZone, edgeDest(List), agentRankHere(Rank)) then moveSplit(Rank, List).

  if bel(expandDest(List), List = [[Value, Vertex]|_], me(Id), agentRankHere(Rank)) then {
   if bel (not((connectedAgent(Id, Agent), currentPos(Agent, Pos), bestExpandDest(_, Value2, Pos),
       Value2 >= Value))) then gotoSplit(Rank, List).
   if bel (not(kingOfTheHill), Rank2 is Rank-1) then gotoSplit(Rank2, List).

   % im in the optimum zone but not on a swarm position, so im basically derping. Move back to opt and
       try again
   if bel (currentPos(Pos), not(swarmPos(Pos)), optimum(Opt), path(Pos, Opt, [Here,Next|Path], _)) then
       advancedGoto(Next).
  }

  if true then recharge.
 }
}
```

# Bibliography


[AF12] Kenneth Balsiger Andersen and Andreas Frøsig. On the multi-agent programming contest. 2012.

[BDH+11] Tristan Behrens, Jürgen Dix, Jomi Hübner, Michael Köster, and Federico Schlesinger. Multi-agent programming contest 2011 edition - documentation. 2011.

[BDH+12] Tristan Behrens, Jürgen Dix, Jomi Hübner, Michael Köster, and Federico Schlesinger. Multi-agent programming contest scenario description - 2012 edition. http://www.multiagentcontest.org/2012/, April 2012.

[DHH+12] Marc Dekker, Pieter Hameete, Michiel Hegemans, Sebastiaan Leysen, Joris Oever, Jeff Smits, and Koen V. Hindriks. *HactarV2: An Agent Team Strategy Based on Implicit Coordination*, volume 7217 of *Lecture Notes in Computer Science*, pages 173–184. Springer Berlin Heidelberg, 2012.

[DKS13] Jürgen Dix, Michael Köster, and Federico Schesinger. Multi-agent programming contest. http://www.multiagentcontest.org/, 2013.

[Hin11] Koen V. Hindriks. *GOAL manual*, May 2011. http://mmi.tudelft.nl/trac/goal.

[Jen00] Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(8):277–296, 2000.

[RN10] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach, 3rd ed.* Prentice Hall, 2010.

[Wik13a] Wikipedia. Artificial intelligence. http://en.wikipedia.org/wiki/Artificial_intelligence, 2013.

[Wik13b] Wikipedia. Daniel dennett. http://en.wikipedia.org/wiki/Daniel_Dennett, 2013.

[Wik13c] Wikipedia. Document type definition. http://en.wikipedia.org/wiki/Document_Type_Definition, 2013.

[Wik13d] Wikipedia. Strong ai. http://en.wikipedia.org/wiki/Strong_AI, May 2013.